Sage Reference Manual

Release 4.0.1

The Sage Development Team

June 06, 2009

CONTENTS

This is the manual for the Sage mathematical software system. Sage is free open source math software that supports research and teaching in algebra, geometry, number theory, cryptography, and related areas. Both the Sage development model and the technology in Sage itself are distinguished by an extremely strong emphasis on openness, community, cooperation, and collaboration: we are building the car, not reinventing the wheel.

This reference manual contains many examples that illustrate the usage of Sage. The examples are all tested with each release of Sage, and should produce exactly the same output as in this manual, except for line breaks.

The Sage command line is briefly described in *The Sage Command Line*, which lists the command line options. For more details about the command line, see the Sage tutorial.

The Sage graphical user interface is described in *The Sage Notebook*. This graphical user interface is unusual in that it operates via your web browser. It provides you with Sage worksheets that you can edit and evaluate, which contain scalable typeset mathematics and beautiful antialiased images.

Enjoy Sage!

CHAPTER

ONE

THE SAGE COMMAND LINE

The chapter lists the Sage command line options. For more details about how to use the Sage command line once it starts up, see the Sage tutorial and the documentation for IPython.

1.1 Attach a file to a running instance of Sage.

class Attach()

Attach a file to a running instance of Sage.

Note: `attach` is *not* a function and is not part of the Python language.

`load` is exactly the same as attach, but doesn't automatically reload a file when it changes.

You attach a file, e.g., `foo.sage` or `foo.py` or `foo.spyx`, to a running Sage session by typing

```
sage: attach foo.sage   # or foo.py   or foo.spyx  (not tested)
```

The contents of the file are then loaded, which means they are read into the running Sage session. For example, if `foo.sage` contains `x=5`, after attaching `foo.sage` the variable `x` will be set to 5. Moreover, any time you change `foo.sage`, the attached file will be re-read automatically (with no intervention on your part).

USAGE: `attach file1 file2 ...` - space-separated list of .py, .spyx, and .sage files.

EFFECT: Each file is read in and added to an internal list of watched files. The meaning of reading a file in depends on the file type:

- read in with no preparsing (so, e.g., `2^3` is 2 bit-xor 3),
- preparsed then the result is read in
- *not* preparsed. Compiled to a module `m` then `from m import *` is executed.

Type `attached_files()` for a list of all currently attached files. You can remove files from this list to stop them from being watched.

Note: `attach` is exactly the same as load, except it keeps track of the loaded file and automatically reloads it when it changes.

1.2 Interactively tracing execution of a command

trace(*code, preparse=True*)

Evaluate Sage code using the interactive tracer and return the result. The string `code` must be a valid expression enclosed in quotes (no assignments - the result of the expression is returned). In the Sage notebook this just raises a NotImplementedException.

INPUT:

- `code` - str
- `preparse` - bool (default: True); if True, run expression through the Sage preparser.

REMARKS: This function is extremely powerful! For example, if you want to step through each line of execution of, e.g., `factor(100)`, type

```
sage: trace("factor(100)")              # not tested
```

then at the (Pdb) prompt type `s` (or `step`), then press return over and over to step through every line of Python that is called in the course of the above computation. Type `?` at any time for help on how to use the debugger (e.g., `l` lists 11 lines around the current line; `bt` gives a back trace, etc.).

Setting a break point: If you have some code in a file and would like to drop into the debugger at a given point, put the following code at that point in the file:

```
import pdb; pdb.set_trace()
```

For an article on how to use the Python debuger, see http://www.onlamp.com/pub/a/python/2005/09/01/debugger.html

TESTS: The only real way to test this is via pexpect spawning a sage subprocess that uses IPython.

```
sage: import pexpect
sage: s = pexpect.spawn('sage')
sage: _ = s.sendline("trace('print factor(10)'); print 3+97")
sage: _ = s.sendline("s"); _ = s.sendline("c");
sage: _ = s.expect('100')
```

Seeing the ipdb prompt and the 2 * 5 in the output below is a strong indication that the trace command worked correctly.

```
sage: print s.before[s.before.find('---'):]
---...
ipdb> c
2 * 5
```

We test what happens in notebook embedded mode:

```
sage: sage.plot.plot.EMBEDDED_MODE = True
sage: trace('print factor(10)')
...
NotImplementedError: the trace command is not implemented in the Sage notebook; you must use the
```

1.3 Sage: Command Line Arguments

There are several flags you can pass to Sage via the command line:

- `-h, -?` Prints a help message.
- `-notebook` Starts the Sage notebook running in default mode. This does not open a browser; it just starts a Sage Notebook server running on http://localhost:8000 (or a subsequent port if that port is not available). Pressing {Control-c} stops the server.
- `-i [packages]` Installs the given Sage packages. If a package is listed at http://modular.math.washington.edu/sage/packages/optional/ and not available in your current directory, then Sage will try to download the package and install it.
- `-optional` Lists all optional packages that can be downloaded.

- `-t <files|dir>` Tests examples in .py, .pyx, .sage or .tex files.
- `-testall` Tests all examples in Sage.
- `-upgrade` Downloads the latest non-optional Sage packages from http://modular.math.washington.edu/sage/packages/standard/, then builds and installs them. In many cases this requires that your computer has the necessary software development tools (listed in the installation documentation). Optional packages have to upgraded manually using `sage -i`. (There are no binary packages yet.)
- `file1.sage file2.sage file.py ...` Starts Sage, compiles any .sage files to .py files, and runs files.
- `-advanced` Displays additional options that can be passed to Sage.

CHAPTER

TWO

THE SAGE NOTEBOOK

2.1 The Sage Notebook object

class Notebook (*dir, system=None, pretty_print=False, show_debug=False, address='localhost', port=8000, secure=True, server_pool=, []*)

DIR()

Return the absolute path to the directory that contains the Sage Notebook directory.

add_to_history(*input_text*)

add_to_user_history(*entry, username*)

add_user(*username, password, email, account_type='user', force=False*)

INPUT:

- `username` - the username
- `password` - the password
- `email` - the email address
- `account_type` - one of 'user', 'admin', or 'guest'
- `force` - bool

If the method get_accounts return False then user can only be added if force=True

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('Mark', 'password', '', force=True)sage:
nb.user('Mark')
Mark
sage: nb.add_user('Sarah', 'password', '')
Traceback (most recent call last):
ValueError: creating new accounts disabled.
sage: nb.set_accounts(True)
sage: nb.add_user('Sarah', 'password', '')sage:
nb.user('Sarah')
Sarah
```

backup_directory()

change_password(*username, password*)

INPUT:

- `username` - the username
- `password` - the password to change the user's password to

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('Mark', 'password', '', force=True)sage:
nb.user('Mark').password()
'aajfMKNH1hTm2'
sage: nb.change_password('Mark', 'different_password')
sage: nb.user('Mark').password()
'aaTlXok5npQME'
```

change_worksheet_key(*old_key, new_key*)

color()

conf()

copy_worksheet(*ws, owner*)

create_default_users(*passwd*)

Create the default users for a notebook.

INPUT:

- `passwd` - a string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.create_default_users('password')
Creating default users.
sage: list(sorted(nb.users().iteritems()))
[('_sage_', _sage_), ('admin', admin), ('guest', guest), ('pub', pub)]
sage: list(sorted(nb.passwords().iteritems()))
[('_sage_', 'aaQSqAReePlq6'), ('admin', 'aajfMKNH1hTm2'), ('guest', 'aaQSqAReePlq6'), ('pub'
sage: nb.create_default_users('newpassword')
Creating default users.
WARNING: User 'pub' already exists -- and is now being replaced.
WARNING: User '_sage_' already exists -- and is now being replaced.
WARNING: User 'guest' already exists -- and is now being replaced.
WARNING: User 'admin' already exists -- and is now being replaced.
sage: list(sorted(nb.passwords().iteritems()))
[('_sage_', 'aaQSqAReePlq6'), ('admin', 'aajH86zjeUSDY'), ('guest', 'aaQSqAReePlq6'), ('pub'
```

create_new_worksheet(*worksheet_name, username, docbrowser=False, add_to_list=True*)

create_new_worksheet_from_history(*name, username, maxlen=None*)

create_user_with_same_password(*user, other_user*)

INPUT:

- `user` - a string
- `other_user` - a string

OUTPUT: Changes password of `user` to that of `other_user`.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('bob', 'an**d', 'bob@gmail.com', force=True)
sage: nb.user('bob').password()
'aa4Q6Jbx/MiUs'
sage: nb.add_user('mary', 'ccd', 'mary@gmail.com', force=True)
sage: nb.user('mary').password()
'aaxr0gcWJMXKU'
sage: nb.create_user_with_same_password('bob', 'mary')
sage: nb.user('bob').password() == nb.user('mary').password()
True
```

default_user()
Return a default login name that the user will see when confronted with the Sage notebook login page.
OUTPUT: string
Currently this returns 'admin' if that is the *only* user. Otherwise it returns the string ''.
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.create_default_users('password')
Creating default users.
sage: nb.default_user()
'admin'
sage: nb.add_user('AnotherUser', 'password', '', force=True)sage:
nb.default_user()
''
```

del_user(*username*)
Deletes the given user
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('Mark', 'password', '', force=True)sage:
nb.user('Mark')
Mark
sage: nb.del_user('Mark')
sage: nb.user('Mark')
Traceback (most recent call last):
KeyError: "no user 'Mark'"
```

delete()
Delete all files related to this notebook.
This is used for doctesting mainly. This command is obviously *VERY* dangerous to use on a notebook you actually care about. You could easily lose all data.
EXAMPLES:

```
sage: tmp = tmp_dir()
sage: nb = sage.server.notebook.notebook.Notebook(tmp)
sage: sorted(os.listdir(tmp))
['backups', 'nb.sobj', 'objects', 'worksheets']
sage: nb.delete()
```

Now the directory is gone.

```
sage: os.listdir(tmp)
...
OSError: [Errno 2] No such file or directory: '...
```

delete_doc_browser_worksheets()

delete_worksheet(*filename*)
Delete the given worksheet and remove its name from the worksheet list.

deleted_worksheets()

directory()

empty_trash(*username*)
Empty the trash for the given user.
INPUT:

- `username` - a string

This empties the trash for the given user and cleans up all files associated with the worksheets that are in the trash.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.new_worksheet_with_title_from_text('Sage', owner='sage')
sage: W.move_to_trash('sage')
sage: nb.worksheet_names()
['sage/0']
sage: nb.empty_trash('sage')
sage: nb.worksheet_names()
[]
```

export_worksheet(*worksheet_filename, output_filename, verbose=True*)

Export a worksheet with given directory filenmae to output_filename.

INPUT:

- `worksheet_filename` - string
- `output_filename` - string
- `verbose` - bool (default: True) if True print some the tar command used to extract the sws file.

OUTPUT: creates a file on the filesystem

get_accounts()

Return `__accounts`

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.get_accounts()
False
sage: nb.set_accounts(True)
sage: nb.get_accounts()
True
```

get_all_worksheets()

get_server()

get_ulimit()

get_worksheet_names_with_collaborator(*user*)

get_worksheet_names_with_viewer(*user*)

get_worksheet_with_filename(*filename*)

Get the worksheet with given filename. If there is no such worksheet, raise a `KeyError`.

INPUT: string OUTPUT: a worksheet or KeyError

get_worksheet_with_name(*name*)

get_worksheets_with_collaborator(*user*)

get_worksheets_with_owner(*owner*)

get_worksheets_with_owner_that_are_viewable_by_user(*owner, user*)

get_worksheets_with_viewer(*user*)

history()

history_count()

history_count_inc()

history_html()

history_text()

history_with_start(*start*)

html(*worksheet_filename=None, username='guest', show_debug=False, admin=False*)

html_afterpublish_window(*worksheet, username, addr, dtime*)
Return the html code for a page dedicated to worksheet publishing after the publication of the given worksheet.

INPUT: worksheet - instance of Worksheet username - string addr - string dtime - instance of time.struct_time

html_banner()

html_banner_and_control(*user, entries*)

html_beforepublish_window(*worksheet, username*)
Return the html code for a page dedicated to worksheet publishing prior to the publication of the given worksheet.

INPUT: worksheet - instance of Worksheet username - string

html_debug_window()

html_doc(*username*)

html_download_or_delete_datafile(*ws, username, filename*)

html_edit_window(*worksheet, username*)
Return a window for editing `worksheet`.

INPUT:

- `worksheet` - a worksheet

html_notebook_settings()

html_plain_text_window(*worksheet, username*)
Return a window that displays a plain text version of the worksheet

INPUT:

- `worksheet` - a worksheet
- `username` - name of the user

html_pretty_print_check_form_element(*ws*)

html_settings()

html_share(*worksheet, username*)

html_slide_controls()

html_specific_revision(*username, ws, rev*)

html_system_select_form_element(*ws*)

html_topbar(*user, pub=False*)

html_upload_data_window(*ws, username*)

html_user_control(*user, entries*)

html_user_settings(*username*)

html_worksheet_page_template(*worksheet, username, title, select=None, backwards=False*)

html_worksheet_revision_list(*username, worksheet*)

html_worksheet_settings(*ws, username*)

html_worksheet_topbar(*worksheet, select=None, username='guest', backwards=False*)

import_worksheet(*filename, owner*)
Upload the worksheet with name `filename` and make it have the given owner.

INPUT:

- `filename` - a string

- `owner` - a string

OUTPUT:

- `worksheet` - a newly created worksheet

EXAMPLES: We create a notebook and import a plain text worksheet into it.

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: name = tmp_filename() + '.txt'
sage: open(name,'w').write('foo\n{{{\n2+3\n}}}')
sage: W = nb.import_worksheet(name, 'admin')
```

W is our newly-created worksheet, with the 2+3 cell in it:

```
sage: W.name()
'foo'
sage: W.cell_list()
[Cell 0; in=2+3, out=]
```

list_window_javascript(*worksheet_filenames*)

max_history_length()

migrate_old()

Migrate all old worksheets, i.e., ones with no owner, to `/pub`.

new_worksheet_with_title_from_text(*text, owner*)

number_of_backups()

object_directory()

object_list_html()

objects()

passwords()

Return the username:password dictionary.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.create_default_users('password')
Creating default users.
sage: nb.add_user('Mark', 'password', '', force=True)sage:
list(sorted(nb.passwords().iteritems()))
[('Mark', 'aajfMKNH1hTm2'), ('_sage_', 'aaQSqAReePlq6'), ('admin', 'aajfMKNH1hTm2'), ('guest
```

plain_text_worksheet_html(*name, prompts=True*)

pretty_print(*username=None*)

publish_worksheet(*worksheet, username*)

Publish the given worksheet.

This creates a new worksheet in the `pub` directory with the same contents as `worksheet`.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('Mark','password','',force=True)sage:
W = nb.new_worksheet_with_title_from_text('First steps', owner='Mark')
sage: nb.worksheet_names()
['Mark/0']
sage: nb.publish_worksheet(nb.get_worksheet_with_filename('Mark/0'), 'Mark')
<BLANKLINE>
[Cell 0; in=, out=]
sage: sorted(nb.worksheet_names())
['Mark/0', 'pub/0']
```

quit()

quit_idle_worksheet_processes()

save(*filename=None, verbose=False*)

scratch_worksheet()

server_pool()

set_accounts(*value*)

Changes `__accounts` to `value`

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.get_accounts()
False
sage: nb.set_accounts(True)
sage: nb.get_accounts()
True
sage: nb.set_accounts(False)
sage: nb.get_accounts()
False
```

set_color(*color*)

set_debug(*show_debug*)

set_directory(*dir*)

set_not_computing()

set_pretty_print(*pretty_print*)

set_server_pool(*servers*)

set_system(*system*)

set_ulimit(*ulimit*)

system(*username=None*)

user(*username*)

Return an instance of the User class given the username of a user in a notebook.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.create_default_users('password')
Creating default users.
sage: nb.user('admin')
admin
sage: nb.user('admin')._User__email
''
sage: nb.user('admin')._User__password
'aajfMKNH1hTm2'
```

user_conf(*username*)

Return a user's configuration.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.create_default_users('password')
Creating default users.
sage: config = nb.user_conf('admin')
sage: config['max_history_length']
500
sage: config['default_system']
```

```
'sage'
sage: config['autosave_interval']
180
sage: config['default_pretty_print']
False
```

user_exists(*username*)

Return whether or not a user exists given a username.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.create_default_users('password')
Creating default users.
sage: nb.user_exists('admin')
True
sage: nb.user_exists('pub')
True
sage: nb.user_exists('mark')
False
sage: nb.user_exists('guest')
True
```

user_history(*username*)

user_history_text(*username, maxlen=None*)

user_is_admin(*user*)

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('Administrator', 'password', '', 'admin', True)sage:
nb.add_user('RegularUser', 'password', '', 'user', True)sage:
nb.user_is_admin('Administrator')
True
sage: nb.user_is_admin('RegularUser')
False
```

user_is_guest(*username*)

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.create_default_users('password')
Creating default users.
sage: nb.user_is_guest('guest')
True
sage: nb.user_is_guest('admin')
False
```

user_list()

Return list of user objects.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.create_default_users('password')
Creating default users.
sage: sorted(nb.user_list(), key=lambda k: k.username())
[_sage_, admin, guest, pub]
```

usernames()

Return list of usernames.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.create_default_users('password')
Creating default users.
sage: sorted(nb.usernames())
['_sage_', 'admin', 'guest', 'pub']
```

users()

Return dictionary of users in a notebook.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.create_default_users('password')
Creating default users.
sage: list(sorted(nb.users().iteritems()))
[('_sage_', _sage_), ('admin', admin), ('guest', guest), ('pub', pub)]
```

valid_login_names()

Return list of users that can be signed in.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.create_default_users('password')
Creating default users.
sage: nb.valid_login_names()
['admin']
sage: nb.add_user('Mark', 'password', '', force=True)sage:
nb.add_user('Sarah', 'password', '', force=True)sage:
nb.add_user('David', 'password', '', force=True)sage:
sorted(nb.valid_login_names())
['David', 'Mark', 'Sarah', 'admin']
```

worksheet_directory()

worksheet_html(*filename, do_print=False*)

worksheet_list_for_public(*username, sort='last_edited', reverse=False, search=None*)

worksheet_list_for_user(*user, typ='active', sort='last_edited', reverse=False, search=None*)

worksheet_names()

Return a list of all the names of worksheets in this notebook.

OUTPUT: list of strings.

EXAMPLES: We make a new notebook with two users and two worksheets, then list their names:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.new_worksheet_with_title_from_text('Sage', owner='sage')
sage: nb.add_user('wstein','sage','wstein@sagemath.org',force=True)
sage: W2 = nb.new_worksheet_with_title_from_text('Elliptic Curves', owner='wstein')
sage: nb.worksheet_names()
['sage/0', 'wstein/0']
```

clean_name(*name*)

load_notebook(*dir, address=None, port=None, secure=None*)

Load the notebook from the given directory, or create one in that directory if one isn't already there.

INPUT:

- `dir` - a string that defines a directory name
- `address` - the address that the notebook server will listen on

- `port` - the port the server listens on
- `secure` - whether or not the notebook is secure

make_path_relative(*dir*)
If easy, replace the absolute path `dir` by a relative one.

sort_worksheet_list(*v, sort, reverse*)
INPUT:

- `sort` - 'last_edited', 'owner', 'rating', or 'name'
- `reverse` - if True, reverse the order of the sort.

2.2 A Cell.

A cell is a single input/output block. Worksheets are built out of a list of cells.

class Cell(*id, input, out, worksheet*)

cell_output_type()
Returns the cell output type.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.cell_output_type()
'wrap'
sage: C.set_cell_output_type('nowrap')
sage: C.cell_output_type()
'nowrap'
```

changed_input_text()
Returns the changed input text for the cell. If there was any changed input text, then it is reset to '' before this method returns.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.changed_input_text()
''
sage: C.set_changed_input_text('3+3')
sage: C.input_text()
'3+3'
sage: C.changed_input_text()
'3+3'
sage: C.changed_input_text()
''
sage: C.version()
0
```

cleaned_input_text()
Returns the input text with all of the percent directives removed. If the cell is interacting, then the interacting text is returned.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '%hide\n%maxima\n2+3', '5', None)
sage: C.cleaned_input_text()
'2+3'
```

computing()
Returns True if self is in its worksheet's queue.
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2")
sage: C.computing()
False

sage: import shutil; shutil.rmtree(nb.directory())
```

delete_files()
Deletes all of the files associated with this cell.
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, 'plot(sin(x),0,5)', ", W)sage:
C.evaluate()
sage: W.check_comp(wait=9999)
('d', Cell 0; in=plot(sin(x),0,5), out=
<html><font color='black'><img src='cell://sage0.png'></font></html>
<BLANKLINE>
)
sage: C.files()
['sage0.png']
sage: C.delete_files()
sage: C.files()
[]
```

delete_output()
Delete all output in this cell.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None); C
Cell 0; in=2+3, out=5
sage: C.delete_output()
sage: C
Cell 0; in=2+3, out=
```

directory()
Returns the directory associated to self. If the directory doesn't already exist, then this method creates it.
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
sage: C.directory()
'.../worksheets/sage/0/cells/0'

sage: import shutil; shutil.rmtree(nb.directory())
```

doc_html(*wrap=None, div_wrap=True, do_print=False*)
Modified version of `self.html` for the doc browser. This is a hack and needs to be improved. The problem is how to get the documentation html to display nicely between the example cells. The type setting (jsMath formating) needs attention too.

edit_text(*ncols=0, prompts=False, max_out=None*)

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.edit_text()
'{{{id=0|\n2+3\n///\n5\n}}}'
```

evaluate(*introspect=False, time=None, username=None*)

INPUT:

- `username` - name of user doing the evaluation
- `time` - if True return time computation takes
- `introspect` - either False or a pair [before_cursor, after_cursor] of strings.

EXAMPLES: We create a notebook, worksheet, and cell and evaluate it in order to compute 3^5:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: W.edit_save('Sage\n{{{\n3^5\n}}}')
sage: C = W.cell_list()[0]; C
Cell 0; in=3^5, out=
sage: C.evaluate(username='sage')
sage: W.check_comp(wait=9999)
('d', Cell 0; in=3^5, out=
243
)
sage: C
Cell 0; in=3^5, out=
243

sage: import shutil; shutil.rmtree(nb.directory())
```

evaluated()

Return True if this cell has been successfully evaluated in a currently running session.

This is not about whether the output of the cell is valid given the input.

OUTPUT:

- `bool` - whether or not this cell has been evaluated in this session

EXAMPLES: We create a worksheet with a cell that has wrong output:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: W.edit_save('Sage\n{{{\n2+3\n///\n20\n}}}')
sage: C = W.cell_list()[0]
sage: C
Cell 0; in=2+3, out=
20
```

We re-evaluate that input cell:

```
sage: C.evaluate()
sage: W.check_comp(wait=9999)
('d', Cell 0; in=2+3, out=
5
)
```

Now the output is right:

```
sage: C
Cell 0; in=2+3, out=
5
```

And the cell is considered to have been evaluated.

```
sage: C.evaluated()
True
```

```
sage: import shutil; shutil.rmtree(nb.directory())
```

files()

Returns a list of all the files in self's directory.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, 'plot(sin(x),0,5)', '', W)sage:
C.evaluate()
sage: W.check_comp(wait=9999)
('d', Cell 0; in=plot(sin(x),0,5), out=
<html><font color='black'><img src='cell://sage0.png'></font></html>
<BLANKLINE>
)
sage: C.files()
['sage0.png']
```

```
sage: import shutil; shutil.rmtree(nb.directory())
```

files_html(*out*)

has_output()

Returns True if there is output for this cell.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.has_output()
True
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '', None)
sage: C.has_output()
False
```

html(*wrap=None, div_wrap=True, do_print=False*)

html_in(*do_print=False, ncols=80*)

Returns the HTML code for the input of this cell.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: print C.html_in()
<div class="insert_new_cell" id="insert_new_cell_0"...</a>
```

html_new_cell_after()

Returns the HTML code for inserting a new cell after self.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: print C.html_new_cell_after()
<div class="insert_new_cell" id="insert_new_cell_0">...
```

html_new_cell_before()
Returns the HTML code for inserting a new cell before self.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: print C.html_new_cell_before()
<div class="insert_new_cell" id="insert_new_cell_0">...
```

html_out(*ncols=0, do_print=False*)

id()
Returns the id of self.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.id()
0
```

input_text()
Returns self's input text.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.input_text()
'2+3'
```

interrupt()
Record that the calculation running in this cell was interrupted.
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2")
sage: C.interrupt()
sage: C.interrupted()
True
sage: C.evaluated()
False

sage: import shutil; shutil.rmtree(nb.directory())
```

interrupted()
Returns True if the evaluation of this cell has been interrupted.
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2")
sage: C.interrupt()
sage: C.interrupted()
True

sage: import shutil; shutil.rmtree(nb.directory())
```

introspect()
TODO: Figure out what the __introspect method is for and write a better doctest.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.introspect()
False
sage: C.set_introspect("a", "b")
sage: C.introspect()
['a', 'b']
```

introspect_html()

is_asap()

Return True if this is an asap cell, i.e., evaluation of it is done as soon as possible.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_asap()
False
sage: C.set_asap(True)
sage: C.is_asap()
True
```

is_auto_cell()

Returns True if self is an auto cell.

An auto cell is a cell that is automatically evaluated when the worksheet starts up.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_auto_cell()
False
sage: C = sage.server.notebook.cell.Cell(0, '#auto\n2+3', '5', None)
sage: C.is_auto_cell()
True
```

is_html()

Returns True if this is an HTML cell. An HTML cell whose system is 'html' and is typically specified by `%html`.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, "%html\nTest HTML", None, None)
sage: C.system()
'html'
sage: C.is_html()
True
sage: C = sage.server.notebook.cell.Cell(0, "Test HTML", None, None)
sage: C.is_html()
False
```

is_interacting()

Returns True

is_interactive_cell()

Return True if this cell contains the use of interact either as a function call or a decorator.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "@interact\ndef f(a=slider(0,10,1,5):\n    print a^2")
sage: C.is_interactive_cell()
True
sage: C = W.new_cell_after(C.id(), "2+2")
```

```
sage: C.is_interactive_cell()
False

sage: import shutil; shutil.rmtree(nb.directory())
```

is_last()

Returns True if self is the last cell in the worksheet.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2"); C
Cell 1; in=2^2, out=
sage: C.is_last()
True
sage: C = W.get_cell_with_id(0)
sage: C.is_last()
False

sage: import shutil; shutil.rmtree(nb.directory())
```

is_no_output()

Return True if this is an no_output cell, i.e., a cell for which we don't care at all about the output.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_no_output()
False
sage: C.set_no_output(True)
sage: C.is_no_output()
True
```

next_id()

Returns the id of the next cell in the worksheet associated to self. If self is not in the worksheet or self is the last cell in the cell_list, then the id of the first cell is returned.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2")
sage: C = W.get_cell_with_id(0)
sage: C.next_id()
1
sage: C = W.get_cell_with_id(1)
sage: C.next_id()
0

sage: import shutil; shutil.rmtree(nb.directory())
```

notebook()

Returns the notebook object associated to self.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
```

```
sage: C.notebook() is nb
True

sage: import shutil; shutil.rmtree(nb.directory())
```

output_html()

output_text(*ncols=0, html=True, raw=False, allow_interact=True*)

parse_html(*s, ncols*)

parse_percent_directives()

Returns a string which consists of the input text of this cell with the percent directives at the top removed. As it's doing this, it computes a list of all the directives and which system (if any) the cell should be run under.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '%hide\n%maxima\n2+3', '5', None)
sage: C.parse_percent_directives()
'2+3'
sage: C.percent_directives()
['hide', 'maxima']
```

percent_directives()

Returns a list of all the percent directives that appear in this cell.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '%hide\n%maxima\n2+3', '5', None)
sage: C.percent_directives()
['hide', 'maxima']
```

plain_text(*ncols=0, prompts=True, max_out=None*)

Returns the plain text version of self.

TODO: Add more comprehensive doctests.

process_cell_urls(*x*)

sage()

TODO: Figure out what exactly this does.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.sage() is None
True
```

set_asap(*asap*)

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_asap()
False
sage: C.set_asap(True)
sage: C.is_asap()
True
```

set_cell_output_type(*typ='wrap'*)

Sets the cell output type.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.cell_output_type()
```

```
'wrap'
sage: C.set_cell_output_type('nowrap')
sage: C.cell_output_type()
'nowrap'
```

set_changed_input_text(*new_text*)

Note that this does not update the version of the cell. This is typically used for things like tab completion.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.set_changed_input_text('3+3')
sage: C.input_text()
'3+3'
sage: C.changed_input_text()
'3+3'
```

set_id(*id*)

Sets the id of self to id.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.set_id(2)
sage: C.id()
2
```

set_input_text(*input*)

Sets the input text of self to be the string input.

TODO: Add doctests for the code dealing with interact.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2")
sage: C.evaluate()
sage: W.check_comp(wait=9999)
('d', Cell 1; in=2^2, out=
4
)
sage: C.version()
0

sage: C.set_input_text('3+3')
sage: C.input_text()
'3+3'
sage: C.evaluated()
False
sage: C.version()
1

sage: import shutil; shutil.rmtree(nb.directory())
```

set_introspect(*before_prompt, after_prompt*)

TODO: Figure out what the __introspect method is for and write a better doctest.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.set_introspect("a", "b")
```

```
sage: C.introspect()
['a', 'b']
```

set_introspect_html(*html, completing=False*)

set_is_html(*v*)

Sets whether or not this cell is an HTML cell.

This is called by check_for_system_switching in worksheet.py.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_html()
False
sage: C.set_is_html(True)
sage: C.is_html()
True
```

set_no_output(*no_output*)

Sets whether or not this is an no_output cell, i.e., a cell for which we don't care at all about the output.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_no_output()
False
sage: C.set_no_output(True)
sage: C.is_no_output()
True
```

set_output_text(*output, html, sage=None*)

set_worksheet(*worksheet, id=None*)

Sets the worksheet object of self to be worksheet and optionally changes the id of self.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: W = "worksheet object"
sage: C.set_worksheet(W)
sage: C.worksheet()
'worksheet object'
sage: C.set_worksheet(None, id=2)
sage: C.id()
2
```

stop_interacting()

system()

Returns the system used to evaluate this cell. The system is specified by a percent directive like '%maxima' at the top of a cell.

If no system is explicitly specified, then None is returned which tells the notebook to evaluate the cell using the worksheet's default system.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '%maxima\n2+3', '5', None)
sage: C.system()
'maxima'
sage: prefixes = ['%hide', '%time', '']
sage: cells = [sage.server.notebook.cell.Cell(0, '%s\n2+3'%prefix, '5', None) for prefix in
sage: [(C, C.system()) for C in cells if C.system() is not None]
[]
```

time()

Returns True if the time it takes to evaluate this cell should be printed.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.time()
False
sage: C = sage.server.notebook.cell.Cell(0, '%time\n2+3', '5', None)
sage: C.time()
True
```

unset_introspect()

TODO: Figure out what the __introspect method is for and write a better doctest.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.set_introspect("a", "b")
sage: C.introspect()
['a', 'b']
sage: C.unset_introspect()
sage: C.introspect()
False
```

update_html_output(*output=''*)

Update the list of files with html-style links or embeddings for this cell.

For interactive cells the html output section is always empty, mainly because there is no good way to distinguish content (e.g., images in the current directory) that goes into the interactive template and content that would go here.

url_to_self()

Returns a notebook URL for this cell.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
sage: C.url_to_self()
'/home/sage/0/cells/0'
```

version()

Returns the version number of this cell.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.version()
0
sage: C.set_input_text('2+3')
sage: C.version()
1
```

word_wrap_cols()

Returns the number of columns for word wrapping. This defaults to 70, but the default setting for a notebook is 72.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.word_wrap_cols()
70
```

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
sage: C.word_wrap_cols()
72

sage: import shutil; shutil.rmtree(nb.directory())
```

worksheet()

Returns the workseet associated to self.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
sage: C.worksheet() is W
True

sage: import shutil; shutil.rmtree(nb.directory())
```

worksheet_filename()

Returns the filename of the worksheet associated to self.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
sage: C.worksheet_filename()
'sage/0'

sage: import shutil; shutil.rmtree(nb.directory())
```

class **Cell_generic**()

delete_output()

Delete all output in this cell. This is not executed - it is an abstract function that must be overwritten in a derived class.

EXAMPLES: This function just raises a NotImplementedError, since it most be defined in derived class.

```
sage: C = sage.server.notebook.cell.Cell_generic()
sage: C.delete_output()
...
NotImplementedError
```

is_interactive_cell()

Returns True if this cell contains the use of interact either as a function call or a decorator.

EXAMPLES:

```
sage: from sage.server.notebook.cell import Cell_generic
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', None)
sage: Cell_generic.is_interactive_cell(C)
False
```

class **ComputeCell**(*id, input, out, worksheet*)

cell_output_type()
Returns the cell output type.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.cell_output_type()
'wrap'
sage: C.set_cell_output_type('nowrap')
sage: C.cell_output_type()
'nowrap'
```

changed_input_text()
Returns the changed input text for the cell. If there was any changed input text, then it is reset to '' before this method returns.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.changed_input_text()
''
sage: C.set_changed_input_text('3+3')
sage: C.input_text()
'3+3'
sage: C.changed_input_text()
'3+3'
sage: C.changed_input_text()
''
sage: C.version()
0
```

cleaned_input_text()
Returns the input text with all of the percent directives removed. If the cell is interacting, then the interacting text is returned.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '%hide\n%maxima\n2+3', '5', None)
sage: C.cleaned_input_text()
'2+3'
```

computing()
Returns True if self is in its worksheet's queue.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2")
sage: C.computing()
False

sage: import shutil; shutil.rmtree(nb.directory())
```

delete_files()
Deletes all of the files associated with this cell.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, 'plot(sin(x),0,5)', '', W)sage:
```

```
C.evaluate()
sage: W.check_comp(wait=9999)
('d', Cell 0; in=plot(sin(x),0,5), out=
<html><font color='black'><img src='cell://sage0.png'></font></html>
<BLANKLINE>
)
sage: C.files()
['sage0.png']
sage: C.delete_files()
sage: C.files()
[]
```

delete_output()
Delete all output in this cell.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None); C
Cell 0; in=2+3, out=5
sage: C.delete_output()
sage: C
Cell 0; in=2+3, out=
```

directory()
Returns the directory associated to self. If the directory doesn't already exist, then this method creates it.
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
sage: C.directory()
'.../worksheets/sage/0/cells/0'

sage: import shutil; shutil.rmtree(nb.directory())
```

doc_html(*wrap=None, div_wrap=True, do_print=False*)
Modified version of `self.html` for the doc browser. This is a hack and needs to be improved. The problem is how to get the documentation html to display nicely between the example cells. The type setting (jsMath formating) needs attention too.

edit_text(*ncols=0, prompts=False, max_out=None*)
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.edit_text()
'{{{id=0|\n2+3\n///\n5\n}}}'
```

evaluate(*introspect=False, time=None, username=None*)
INPUT:

- `username` - name of user doing the evaluation
- `time` - if True return time computation takes
- `introspect` - either False or a pair [before_cursor, after_cursor] of strings.

EXAMPLES: We create a notebook, worksheet, and cell and evaluate it in order to compute 3^5:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: W.edit_save('Sage\n{{{\n3^5\n}}}')
```

```
sage: C = W.cell_list()[0]; C
Cell 0; in=3^5, out=
sage: C.evaluate(username='sage')
sage: W.check_comp(wait=9999)
('d', Cell 0; in=3^5, out=
243
)
sage: C
Cell 0; in=3^5, out=
243

sage: import shutil; shutil.rmtree(nb.directory())
```

evaluated()

Return True if this cell has been successfully evaluated in a currently running session. This is not about whether the output of the cell is valid given the input.

OUTPUT:

- `bool` - whether or not this cell has been evaluated in this session

EXAMPLES: We create a worksheet with a cell that has wrong output:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: W.edit_save('Sage\n{{{\n2+3\n///\n20\n}}}')
sage: C = W.cell_list()[0]
sage: C
Cell 0; in=2+3, out=
20
```

We re-evaluate that input cell:

```
sage: C.evaluate()
sage: W.check_comp(wait=9999)
('d', Cell 0; in=2+3, out=
5
)
```

Now the output is right:

```
sage: C
Cell 0; in=2+3, out=
5
```

And the cell is considered to have been evaluated.

```
sage: C.evaluated()
True

sage: import shutil; shutil.rmtree(nb.directory())
```

files()

Returns a list of all the files in self's directory.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, 'plot(sin(x),0,5)', '', W)sage:
C.evaluate()
sage: W.check_comp(wait=9999)
```

```
('d', Cell 0; in=plot(sin(x),0,5), out=
<html><font color='black'><img src='cell://sage0.png'></font></html>
<BLANKLINE>
)
sage: C.files()
['sage0.png']

sage: import shutil; shutil.rmtree(nb.directory())
```

files_html(*out*)

has_output()

Returns True if there is output for this cell.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.has_output()
True
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '', None)
sage: C.has_output()
False
```

html(*wrap=None, div_wrap=True, do_print=False*)

html_in(*do_print=False, ncols=80*)

Returns the HTML code for the input of this cell.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: print C.html_in()
<div class="insert_new_cell" id="insert_new_cell_0"...</a>
```

html_new_cell_after()

Returns the HTML code for inserting a new cell after self.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: print C.html_new_cell_after()
<div class="insert_new_cell" id="insert_new_cell_0">...
```

html_new_cell_before()

Returns the HTML code for inserting a new cell before self.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: print C.html_new_cell_before()
<div class="insert_new_cell" id="insert_new_cell_0">...
```

html_out(*ncols=0, do_print=False*)

id()

Returns the id of self.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.id()
0
```

input_text()

Returns self's input text.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.input_text()
'2+3'
```

interrupt()

Record that the calculation running in this cell was interrupted.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2")
sage: C.interrupt()
sage: C.interrupted()
True
sage: C.evaluated()
False

sage: import shutil; shutil.rmtree(nb.directory())
```

interrupted()

Returns True if the evaluation of this cell has been interrupted.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2")
sage: C.interrupt()
sage: C.interrupted()
True

sage: import shutil; shutil.rmtree(nb.directory())
```

introspect()

TODO: Figure out what the __introspect method is for and write a better doctest.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.introspect()
False
sage: C.set_introspect("a", "b")
sage: C.introspect()
['a', 'b']
```

introspect_html()

is_asap()

Return True if this is an asap cell, i.e., evaluation of it is done as soon as possible.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_asap()
False
sage: C.set_asap(True)
sage: C.is_asap()
True
```

is_auto_cell()

Returns True if self is an auto cell.

An auto cell is a cell that is automatically evaluated when the worksheet starts up.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_auto_cell()
False
sage: C = sage.server.notebook.cell.Cell(0, '#auto\n2+3', '5', None)
sage: C.is_auto_cell()
True
```

is_html()

Returns True if this is an HTML cell. An HTML cell whose system is 'html' and is typically specified by `%html`.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, "%html\nTest HTML", None, None)
sage: C.system()
'html'
sage: C.is_html()
True
sage: C = sage.server.notebook.cell.Cell(0, "Test HTML", None, None)
sage: C.is_html()
False
```

is_interacting()

Returns True

is_interactive_cell()

Return True if this cell contains the use of interact either as a function call or a decorator.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "@interact\ndef f(a=slider(0,10,1,5):\n    print a^2")
sage: C.is_interactive_cell()
True
sage: C = W.new_cell_after(C.id(), "2+2")
sage: C.is_interactive_cell()
False

sage: import shutil; shutil.rmtree(nb.directory())
```

is_last()

Returns True if self is the last cell in the worksheet.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2"); C
Cell 1; in=2^2, out=
sage: C.is_last()
True
sage: C = W.get_cell_with_id(0)
sage: C.is_last()
False

sage: import shutil; shutil.rmtree(nb.directory())
```

is_no_output()

Return True if this is an no_output cell, i.e., a cell for which we don't care at all about the output.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_no_output()
False
sage: C.set_no_output(True)
sage: C.is_no_output()
True
```

next_id()

Returns the id of the next cell in the worksheet associated to self. If self is not in the worksheet or self is the last cell in the cell_list, then the id of the first cell is returned.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2")
sage: C = W.get_cell_with_id(0)
sage: C.next_id()
1
sage: C = W.get_cell_with_id(1)
sage: C.next_id()
0

sage: import shutil; shutil.rmtree(nb.directory())
```

notebook()

Returns the notebook object associated to self.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
sage: C.notebook() is nb
True

sage: import shutil; shutil.rmtree(nb.directory())
```

output_html()

output_text(*ncols=0, html=True, raw=False, allow_interact=True*)

parse_html(*s, ncols*)

parse_percent_directives()

Returns a string which consists of the input text of this cell with the percent directives at the top removed. As it's doing this, it computes a list of all the directives and which system (if any) the cell should be run under.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '%hide\n%maxima\n2+3', '5', None)
sage: C.parse_percent_directives()
'2+3'
sage: C.percent_directives()
['hide', 'maxima']
```

percent_directives()
Returns a list of all the percent directives that appear in this cell.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '%hide\n%maxima\n2+3', '5', None)
sage: C.percent_directives()
['hide', 'maxima']
```

plain_text(*ncols=0, prompts=True, max_out=None*)
Returns the plain text version of self.
TODO: Add more comprehensive doctests.

process_cell_urls(*x*)

sage()
TODO: Figure out what exactly this does.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.sage() is None
True
```

set_asap(*asap*)
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_asap()
False
sage: C.set_asap(True)
sage: C.is_asap()
True
```

set_cell_output_type(*typ='wrap'*)
Sets the cell output type.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.cell_output_type()
'wrap'
sage: C.set_cell_output_type('nowrap')
sage: C.cell_output_type()
'nowrap'
```

set_changed_input_text(*new_text*)
Note that this does not update the version of the cell. This is typically used for things like tab completion.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.set_changed_input_text('3+3')
sage: C.input_text()
'3+3'
sage: C.changed_input_text()
'3+3'
```

set_id(*id*)
Sets the id of self to id.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.set_id(2)
```

```
sage: C.id()
2
```

set_input_text(*input*)

Sets the input text of self to be the string input.

TODO: Add doctests for the code dealing with interact.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = W.new_cell_after(0, "2^2")
sage: C.evaluate()
sage: W.check_comp(wait=9999)
('d', Cell 1; in=2^2, out=
4
)
sage: C.version()
0

sage: C.set_input_text('3+3')
sage: C.input_text()
'3+3'
sage: C.evaluated()
False
sage: C.version()
1

sage: import shutil; shutil.rmtree(nb.directory())
```

set_introspect(*before_prompt, after_prompt*)

TODO: Figure out what the __introspect method is for and write a better doctest.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.set_introspect("a", "b")
sage: C.introspect()
['a', 'b']
```

set_introspect_html(*html, completing=False*)

set_is_html(*v*)

Sets whether or not this cell is an HTML cell.

This is called by check_for_system_switching in worksheet.py.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_html()
False
sage: C.set_is_html(True)
sage: C.is_html()
True
```

set_no_output(*no_output*)

Sets whether or not this is an no_output cell, i.e., a cell for which we don't care at all about the output.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.is_no_output()
```

```
False
sage: C.set_no_output(True)
sage: C.is_no_output()
True
```

set_output_text(*output, html, sage=None*)

set_worksheet(*worksheet, id=None*)

Sets the worksheet object of self to be worksheet and optionally changes the id of self.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: W = "worksheet object"
sage: C.set_worksheet(W)
sage: C.worksheet()
'worksheet object'
sage: C.set_worksheet(None, id=2)
sage: C.id()
2
```

stop_interacting()

system()

Returns the system used to evaluate this cell. The system is specified by a percent directive like '%maxima' at the top of a cell.

If no system is explicitly specified, then None is returned which tells the notebook to evaluate the cell using the worksheet's default system.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '%maxima\n2+3', '5', None)
sage: C.system()
'maxima'
sage: prefixes = ['%hide', '%time', '']
sage: cells = [sage.server.notebook.cell.Cell(0, '%s\n2+3'%prefix, '5', None) for prefix in
sage: [(C, C.system()) for C in cells if C.system() is not None]
[]
```

time()

Returns True if the time it takes to evaluate this cell should be printed.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.time()
False
sage: C = sage.server.notebook.cell.Cell(0, '%time\n2+3', '5', None)
sage: C.time()
True
```

unset_introspect()

TODO: Figure out what the __introspect method is for and write a better doctest.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.set_introspect("a", "b")
sage: C.introspect()
['a', 'b']
sage: C.unset_introspect()
sage: C.introspect()
False
```

update_html_output(*output=''*)
Update the list of files with html-style links or embeddings for this cell.

For interactive cells the html output section is always empty, mainly because there is no good way to distinguish content (e.g., images in the current directory) that goes into the interactive template and content that would go here.

url_to_self()
Returns a notebook URL for this cell.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
sage: C.url_to_self()
'/home/sage/0/cells/0'
```

version()
Returns the version number of this cell.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.version()
0
sage: C.set_input_text('2+3')
sage: C.version()
1
```

word_wrap_cols()
Returns the number of columns for word wrapping. This defaults to 70, but the default setting for a notebook is 72.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', None)
sage: C.word_wrap_cols()
70

sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
sage: C.word_wrap_cols()
72

sage: import shutil; shutil.rmtree(nb.directory())
```

worksheet()
Returns the workseet associated to self.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
sage: C.worksheet() is W
True

sage: import shutil; shutil.rmtree(nb.directory())
```

worksheet_filename()
Returns the filename of the worksheet associated to self.
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: C = sage.server.notebook.cell.Cell(0, '2+3', '5', W)
sage: C.worksheet_filename()
'sage/0'

sage: import shutil; shutil.rmtree(nb.directory())
```

class TextCell(*id, text, worksheet*)

delete_output()
Delete all output in this cell. This does nothing since text cells have no output.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', None)
sage: C
TextCell 0: 2+3
sage: C.delete_output()
sage: C
TextCell 0: 2+3
```

edit_text()
EXAMPLES:

```
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', None)
sage: C.edit_text()
'2+3'
```

html(*ncols=0, do_print=False, do_math_parse=True, editing=False*)
Returns an HTML version of self as a string.
INPUT:

- do_math_parse - bool (default: True) If True, call math_parse (defined in cell.py) on the html.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', None)
sage: C.html()
'<div class="text_cell" id="cell_text_0">2+3...'
sage: C.set_input_text("$2+3$")
sage: C.html(do_math_parse=True)
'<div class="text_cell" id="cell_text_0"><span class="math">2+3</span>...'
```

html_inner(*ncols=0, do_print=False, do_math_parse=True, editing=False*)
Returns an HTML version of the content of self as a string.
INPUT:

- do_math_parse - bool (default: True) If True, call math_parse (defined in cell.py) on the html.

EXAMPLES:

```
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', None)
sage: C.html_inner()
'2+3...'
sage: C.set_input_text("$2+3$")
sage: C.html_inner(do_math_parse=True)
'<span class="math">2+3</span>...'
```

id()
EXAMPLES:

```
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', None)
sage: C.id()
0
```

is_auto_cell()
EXAMPLES:

```
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', None)
sage: C.is_auto_cell()
False
```

plain_text(*prompts=False*)
EXAMPLES:

```
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', None)
sage: C.plain_text()
'2+3'
```

set_cell_output_type(*typ='wrap'*)
This does nothing for TextCells.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', None)
sage: C.set_cell_output_type("wrap")
```

set_input_text(*input_text*)
Sets the input text of self to be input_text.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', None)
sage: C
TextCell 0: 2+3
sage: C.set_input_text("3+2")
sage: C
TextCell 0: 3+2
```

set_worksheet(*worksheet, id=None*)
Sets the worksheet object of self to be worksheet and optionally changes the id of self.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', None)
sage: W = "worksheet object"
sage: C.set_worksheet(W)
sage: C.worksheet()
'worksheet object'
sage: C.set_worksheet(None, id=2)
sage: C.id()
2
```

worksheet()
Returns the worksheet object associated to self.
EXAMPLES:

```
sage: C = sage.server.notebook.cell.TextCell(0, '2+3', 'worksheet object')
sage: C.worksheet()
'worksheet object'
```

format_exception(*s0, ncols*)

Make it so excpetions don't appear expanded by default.

INPUT:

- `s0` - string
- `ncols` - integer

OUTPUT: string

If s0 contains "notracebacks" then this function always returns s0

EXAMPLES:

```
sage: sage.server.notebook.cell.format_exception(sage.server.notebook.cell.TRACEBACK,80)
'\nTraceback (click to the left for traceback)\n...\nTraceback (most recent call last):'
sage: sage.server.notebook.cell.format_exception(sage.server.notebook.cell.TRACEBACK + "notraceb
'Traceback (most recent call last):notracebacks'
```

number_of_rows(*txt, ncols*)

Returns the number of rows needed to display the string in txt if there are a maximum of ncols columns per row.

EXAMPLES:

```
sage: from sage.server.notebook.cell import number_of_rows
sage: s = "asdfasdf\nasdfasdf\n"
sage: number_of_rows(s, 8)
2
sage: number_of_rows(s, 5)
4
sage: number_of_rows(s, 4)
4
```

2.3 A Worksheet.

A worksheet is embedded in a webpage that is served by the Sage server. It is a linearly-ordered collections of numbered cells, where a cell is a single input/output block.

The worksheet module is responsible for running calculations in a worksheet, spawning Sage processes that do all of the actual work and are controlled via pexpect, and reporting on results of calculations. The state of the cells in a worksheet is stored on the filesystem (not in the notebook pickle sobj).

AUTHORS:

- William Stein

class Worksheet(*name, dirname, notebook_worksheet_directory, system, owner, docbrowser=False, pretty_print=False, auto_publish=False*)

DIR()

Return the absolute path to the directory that contains the Sage Notebook directory for the notebook that contains this worksheet.

OUTPUT: string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.DIR()    # random output
'/Users/was/.sage/temp/teragon_2.local/19129'
```

add_collaborator(*user*)

Add the given user as a collaborator on this worksheet.

INPUT:

- `user` - a string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('diophantus','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Collaborator test', 'admin')
sage: W.collaborators()
[]
sage: W.add_collaborator('diophantus')
sage: W.collaborators()
['diophantus']
```

add_viewer(*user*)

Add the given user as an allowed viewer of this worksheet.

INPUT:

- `user` - string (username)

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('diophantus','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Viewer test', 'admin')
sage: W.add_viewer('diophantus')
sage: W.viewers()
['diophantus']
```

append(*L*)

append_new_cell()

Create and append a new cell to the list of cells.

OUTPUT: a new empty cell

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Test Edit Save', 'admin')
sage: W
[Cell 0; in=, out=]
sage: W.append_new_cell()
Cell 1; in=, out=
sage: W
[Cell 0; in=, out=, Cell 1; in=, out=]
```

attach(*filename*)

attached_data_files()

Return a list of the filenames of files in the worksheet data directory.

OUTPUT: list of strings

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.attached_data_files()
[]
sage: open('%s/foo.data'%W.data_directory(),'w').close()
sage: W.attached_data_files()
['foo.data']
```

attached_files()

attached_html()

autosave(*username*)

best_completion(*s, word*)

cell_id_list()

Return a new list of the id's of cells in this worksheet.

OUTPUT: a new list

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Test Edit Save', 'admin')
```

Now we set the worksheet to have two cells with the default id of 0 and another with id 10.

```
sage: W.edit_save('Sage\n{{{\n2+3\n///\n5\n}}}\n{{{id=10|\n2+8\n///\n10\n}}}')
sage: W.cell_id_list()
[0, 10]
```

cell_list()

Return a reference to the list of the all the cells in this worksheet.

OUTPUT:

- `list` - a list of cells

Note: This function loads the cell list from disk (the file worksheet.txt) if it isn't available in memory.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Test Edit Save', 'admin')
sage: W.edit_save('Sage\n{{{\n2+3\n///\n5\n}}}\n{{{\n2+8\n///\n10\n}}}')
sage: v = W.cell_list(); v
[Cell 0; in=2+3, out=
5, Cell 1; in=2+8, out=
10]
sage: v[0]
Cell 0; in=2+3, out=
5
```

cells_directory()

Return the directory in which the cells of this worksheet are evaluated.

OUTPUT: string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.cells_directory()
'.../worksheets/admin/0/cells/'
```

check_cell(*id*)

Check the status on computation of the cell with given id.

INPUT:

- `id` - an integer

OUTPUT:

- `status` - a string, either 'd' (done) or 'w' (working)
- `cell` - the cell with given id

check_comp(*wait=0.20000000000000001*)

Check on currently computing cells in the queue.

INPUT:

- `wait` - float (default: 0.2); how long to wait for output.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: W.edit_save('Sage\n{{{\n3^20\n}}}')
sage: W.cell_list()[0].evaluate()
sage: W.check_comp()      # random output -- depends on computer speed
('d', Cell 0; in=3^20, out=
3486784401
)
sage: nb.delete()
```

check_for_system_switching(*input, cell*)

Check for input cells that start with `%foo`, where `foo` is an object with an eval method.

INPUT:

- `s` - a string of the code from the cell to be executed
- `C` - the cell object

EXAMPLES: First, we set up a new notebook and worksheet.

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
```

We first test running a native command in 'sage' mode and then a GAP cell within Sage mode.

```
sage: W.edit_save('Sage\nsystem:sage\n{{{\n2+3\n}}}\n\n{{{\n%gap\nSymmetricGroup(5)\n}}}')
sage: c0, c1 = W.cell_list()
sage: W.check_for_system_switching(c0.cleaned_input_text(), c0)
(False, '2+3')
sage: W.check_for_system_switching(c1.cleaned_input_text(), c1)
(True, "print _support_.syseval(gap, ur'''SymmetricGroup(5)''', '...')")

sage: c0.evaluate()
sage: W.check_comp()  #random output -- depends on the computer's speed
('d', Cell 0; in=2+3, out=
5
)
sage: c1.evaluate()
sage: W.check_comp()  #random output -- depends on the computer's speed
('d', Cell 1; in=%gap
SymmetricGroup(5), out=
Sym( [ 1 .. 5 ] )
)
```

Next, we run the same commands but from 'gap' mode.

```
sage: W.edit_save('Sage\nsystem:gap\n{{{\n%sage\n2+3\n}}}\n\n{{{\nSymmetricGroup(5)\n}}}')
sage: c0, c1 = W.cell_list()
sage: W.check_for_system_switching(c0.cleaned_input_text(), c0)
(False, '2+3')
sage: W.check_for_system_switching(c1.cleaned_input_text(), c1)
(True,
 "print _support_.syseval(gap, ur'''SymmetricGroup(5)''', '...')")
```

```
sage: c0.evaluate()
sage: W.check_comp()  #random output -- depends on the computer's speed
('d', Cell 0; in=%sage
2+3, out=
5
)
sage: c1.evaluate()
sage: W.check_comp()  #random output -- depends on the computer's speed
('d', Cell 1; in=SymmetricGroup(5), out=
Sym( [ 1 .. 5 ] )
)
```

clear()

clear_queue()

collaborator_names(*max=None*)

Returns a string of the non-owner collaborators on this worksheet.

INPUT:

- `max` - an integer. If this is specified, then only max number of collaborators are shown.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('test1', 'admin')
sage: C = W.collaborators(); C
[]
sage: C.append('sage')
sage: C.append('wstein')
sage: W.collaborator_names()
'sage, wstein'
sage: W.collaborator_names(max=1)
'sage, ...'
```

collaborators()

Return a (reference to the) list of the collaborators who can also view and modify this worksheet.

OUTPUT: list

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('test1', 'admin')
sage: C = W.collaborators(); C
[]
sage: C.append('sage')
sage: W.collaborators()
['sage']
```

completions_html(*id, s, cols=3*)

compute_cell_id_list()

compute_process_has_been_started()

Return True precisely if the compute process has been started, irregardless of whether or not it is currently churning away on a computation.

computing()

Return whether or not a cell is currently being run in the worksheet Sage process.

conf()

Return the configuration object for this worksheet, which is stored in an sobj in the worksheet directory.

OUTPUT: worksheet configuration object.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('test1', 'admin')
sage: W.conf()
Configuration: {}
```

cython_import(*cmd, cell*)

data_directory()

Return path to directory where worksheet data is stored.

OUTPUT: string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.data_directory()
'.../worksheets/admin/0/data/'
```

date_edited()

Returns the date the worksheet was last edited if already recorded otherwise the current local time is recorded and returned.

delete_all_output(*username*)

Delete all the output in all the worksheet cells.

INPUT:

- `username` - name of the user requesting the deletion.

EXAMPLES: We create a new notebook, user, and a worksheet with one cell.

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: W.edit_save('Sage\nsystem:sage\n{{{\n2+3\n///\n5\n}}}')
```

Notice that there is 1 cell with 5 in its output.

```
sage: W.cell_list()
[Cell 0; in=2+3, out=
5]
```

We now delete the output, observe that it is gone.

```
sage: W.delete_all_output('sage')
sage: W.cell_list()
[Cell 0; in=2+3, out=]
```

If an invalid user tries to delete all, a ValueError is raised.

```
sage: W.delete_all_output('hacker')
...
ValueError: user 'hacker' not allowed to edit this worksheet
```

Clean up.

```
sage: nb.delete()
```

delete_cell_input_files()

Delete all the files `code_%s.py` and `code_%s.spyx` that are created when evaluating cells. We do this when we first start the notebook to get rid of clutter.

delete_cell_with_id(*id*)

Remove the cell with given id and return the cell before it.

delete_cells_directory()

Delete the directory in which all the cell computations occur.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: W.edit_save('Sage\n{{{\n3^20\n}}}')
sage: sorted(os.listdir(W.directory()))
['snapshots', 'worksheet.txt']
sage: W.cell_list()[0].evaluate()
sage: sorted(os.listdir(W.directory()))
['cells', 'code', 'data', 'snapshots', 'worksheet.txt']
sage: W.delete_cells_directory()
sage: sorted(os.listdir(W.directory()))
['code', 'data', 'snapshots', 'worksheet.txt']
```

delete_notebook_specific_data()

Delete data from this worksheet this is specific to a certain notebook. This means deleting the attached files, collaborators, and viewers.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('hilbert','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('test1', 'admin')
sage: W.add_viewer('hilbert')
sage: W.delete_notebook_specific_data()
sage: W.viewers()
[]
sage: W.add_collaborator('hilbert')
sage: W.collaborators()
['admin', 'hilbert']
sage: W.delete_notebook_specific_data()
sage: W.collaborators()
['admin']
```

delete_user(*user*)

Delete a user from having any view or ownership of this worksheet.

INPUT:

- `user` - string; the name of a user

EXAMPLES: We create a notebook with 2 users and 1 worksheet that both view.

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('wstein','sage','wstein@sagemath.org',force=True)
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.new_worksheet_with_title_from_text('Sage', owner='sage')
sage: W.add_viewer('wstein')
sage: W.owner()
'sage'
sage: W.viewers()
['wstein']
```

We delete the sage user from the worksheet W. This makes wstein the new owner.

```
sage: W.delete_user('sage')
sage: W.viewers()
['wstein']
sage: W.owner()
'wstein'
```

Then we delete wstein from W, which makes the owner None:

```
sage: W.delete_user('wstein')
sage: W.owner() is None
True
sage: W.viewers()
[]
```

Finally, we clean up.

```
sage: nb.delete()
```

detach (*filename*)

directory()

Return the full path to the directory where this worksheet is stored.

OUTPUT: string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.directory()
'.../worksheets/admin/0'
```

do_sage_extensions_preparsing (*s, files_seen_so_far=, [], this_file=''*)

docbrowser()

Return True if this is a docbrowser worksheet.

OUTPUT: bool

EXAMPLES: We first create a standard worksheet for which docbrowser is of course False:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('test1', 'admin')
sage: W.docbrowser()
False
```

We create a worksheet for which docbrowser is True:

```
sage: W = nb.create_new_worksheet('docs', 'admin', docbrowser=True)
sage: W.docbrowser()
True
```

edit_save (*text, ignore_ids=False*)

Set the contents of this worksheet to the worksheet defined by the plain text string text, which should be a sequence of html and 's code blocks.

INPUT:

- `text` - a string
- `ignore_ids` - bool (default: False); if True ignore all the id's in the code block.

EXAMPLES: We create a new test notebook and a worksheet.

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test Edit Save', 'sage')
```

We set the contents of the worksheet using the edit_save command.

```
sage: W.edit_save('Sage\n{{{\n2+3\n///\n5\n}}}\n{{{\n2+8\n///\n10\n}}}')
sage: W
[Cell 0; in=2+3, out=
5, Cell 1; in=2+8, out=
10]
```

```
sage: W.name()
'Sage'
```

edit_text()
Returns a plain-text version of the worksheet with {{{}}} wiki-formatting, suitable for hand editing.

enqueue(*C, username=None, next=False*)
Queue up the cell C for evaluation in this worksheet.
INPUT:

- C - a Cell
- username - the name of the user that is evaluating this cell (mainly used for loging)

Note: If C.is_asap() is True, then we put C as close to the beginning of the queue as possible, but after all asap cells. Otherwise, C goes at the end of the queue.

eval_asap_no_output(*cmd, username=None*)

everyone_has_deleted_this_worksheet()
Return True if all users have deleted this worksheet, so we know we can safely purge it from disk.
OUTPUT: bool
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: W.everyone_has_deleted_this_worksheet()
False
sage: W.move_to_trash('admin')
sage: W.everyone_has_deleted_this_worksheet()
True
```

filename()
Return the filename (really directory) where the files associated to this worksheet are stored.
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.filename()
'admin/0'
sage: sorted(os.listdir(nb.directory() + '/worksheets/' + W.filename()))
['snapshots', 'worksheet.txt']
```

filename_without_owner()
Return the part of the worksheet filename after the last /, i.e., without any information about the owner of this worksheet.
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.filename_without_owner()
'0'
sage: W.filename()
'admin/0'
```

get_cell_system(*cell*)
Returns the system that will run the input in cell. This defaults to worksheet's system if there is not one specifically given in the cell.
EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: W.edit_save('Sage\nsystem:sage\n{{{\n2+3\n}}}\n\n{{{\n%gap\nSymmetricGroup(5)\n}}}')
sage: c0, c1 = W.cell_list()
sage: W.get_cell_system(c0)
'sage'
sage: W.get_cell_system(c1)
'gap'
sage: W.edit_save('Sage\nsystem:gap\n{{{\n%sage\n2+3\n}}}\n\n{{{\nSymmetricGroup(5)\n}}}')
sage: c0, c1 = W.cell_list()
sage: W.get_cell_system(c0)
'sage'
sage: W.get_cell_system(c1)
'gap'
```

get_cell_with_id(*id*)

get_snapshot_text_filename(*name*)

has_published_version()

Return True if there is a published version of this worksheet.

OUTPUT: bool

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: P = nb.publish_worksheet(W, 'admin')
sage: P.has_published_version()
False
sage: W.has_published_version()
True
```

hide_all()

html(*include_title=True, do_print=False, confirm_before_leave=False, read_only=False*)

html_data_options_list()

html_file_menu()

html_menu()

html_ratings_info()

Return html that renders to give a summary of how this worksheet has been rated.

OUTPUT:

- `string` - a string of HTML as a bunch of table rows.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: W.rate(0, 'this lacks content', 'riemann')
sage: W.rate(3, 'this is great', 'hilbert')
sage: W.html_ratings_info()
'<tr><td>hilbert</td><td align=center>3</td><td>this is great</td></tr>\n<tr><td>riemann</td>
```

html_save_discard_buttons()

html_share_publish_buttons(*select=None, backwards=False*)

html_time_last_edited()

html_time_since_last_edited()

html_title(*username='guest'*)

html_worksheet_body(*do_print, publish=False*)

hunt_file(*filename*)

initialize_sage()

input_text()
Return text version of the input to the worksheet.

interrupt()
Interrupt all currently queued up calculations.

OUTPUT:

- `bool` - return True if no problems interrupting calculation return False if the Sage interpreter had to be restarted.

EXAMPLES: We create a worksheet and start a large factorization going:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: W.edit_save('Sage\n{{{\nfactor(2^997-1)\n}}}')
sage: W.cell_list()[0].evaluate()
```

It's running still

```
sage: W.check_comp()
('w', Cell 0; in=factor(2^997-1), out=...)
```

We interrupt it successfully.

```
sage: W.interrupt()          # random -- could fail on heavily loaded machine
True
```

Now we check and nothing is computing.

```
sage: W.check_comp()         # random -- could fail on heavily loaded machine
('e', None)
```

Clean up.

```
sage: nb.delete()
```

is_active(*user*)
Return True if this worksheet is active for the given user.

INPUT:

- `user` - string

OUTPUT: bool

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Active Test', 'admin')
sage: W.is_active('admin')
True
sage: W.move_to_archive('admin')
sage: W.is_active('admin')
False
```

is_archived(*user*)
Return True if this worksheet is archived for the given user.

INPUT:

- `user` - string

OUTPUT: bool

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Archived Test', 'admin')
sage: W.is_archived('admin')
False
sage: W.move_to_archive('admin')
sage: W.is_archived('admin')
True
```

is_auto_publish()

Returns boolean of "Is this worksheet set to be published automatically when saved?" if private variable "autopublish" is set otherwise False is returned and the variable is set to False.

is_doc_worksheet()

is_last_id_and_previous_is_nonempty(*id*)

is_owner(*username*)

is_published()

Return True if this worksheet is a published worksheet.

OUTPUT:

- `bool` - whether or not owner is 'pub'

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: W.is_published()
False
sage: W.set_owner('pub')
sage: W.is_published()
True
```

is_publisher(*username*)

Return True if username is the username of the publisher of this worksheet, assuming this worksheet was published.

INPUT:

- `username` - string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: P = nb.publish_worksheet(W, 'admin')
sage: P.is_publisher('hearst')
False
sage: P.is_publisher('admin')
True
```

is_rater(*username*)

Return True is the user with given username has rated this worksheet.

INPUT:

- `username` - string

OUTPUT: bool

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: W.rate(0, 'this lacks content', 'riemann')
sage: W.is_rater('admin')
False
sage: W.is_rater('riemann')
True
```

is_trashed(*user*)

Return True if this worksheet is in the trash for the given user.

INPUT:

- `user` - string

OUTPUT: bool

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Trash Test', 'admin')
sage: W.is_trashed('admin')
False
sage: W.move_to_trash('admin')
sage: W.is_trashed('admin')
True
```

javascript_confirm_before_leave()

javascript_for_being_active_worksheet()

javascript_for_jsmath_rendering()

last_compute_walltime()

last_edited()

last_to_edit()

limit_snapshots()

This routine will limit the number of snapshots of a worksheet, as specified by a hard-cioded value below.

Prior behavior was to allow unlimited numbers of snapshots and so this routine will not delete files created prior to this change.

This assumes snapshot names correspond to the `time.time()` method used to create base filenames in seconds in UTC time, and that there are no other extraneous files in the directory.

load_any_changed_attached_files(*s*)

Modify `s` by prepending any necessary load commands corresponding to attached files that have changed.

load_path()

move_out_of_trash(*user*)

Exactly the same as set_active(user).

INPUT:

- `user` - string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Active Test', 'admin')
sage: W.move_to_trash('admin')
sage: W.is_active('admin')
False
sage: W.move_out_of_trash('admin')
sage: W.is_active('admin')
True
```

move_to_archive(*user*)

Move this worksheet to be archived for the given user.

INPUT:

- `user` - string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Archive Test', 'admin')
sage: W.move_to_archive('admin')
sage: W.is_archived('admin')
True
```

move_to_trash(*user*)

Move this worksheet to the trash for the given user.

INPUT:

- `user` - string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Trash Test', 'admin')
sage: W.move_to_trash('admin')
sage: W.is_trashed('admin')
True
```

name()

Return the name of this worksheet.

OUTPUT: string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.name()
'A Test Worksheet'
```

new_cell_after(*id, input=''*)

Insert a new cell into the cell list after the cell with the given integer id.

INPUT:

- `id` - integer
- `input` - string

OUTPUT:

A new cell with the given input text (empty by default).

new_cell_before(*id, input=''*)

Insert a new cell into the cell list before the cell with the given integer id. If the id is not the id of any cell, inserts a new cell at the end of the cell list.

INPUT:

- `id` - integer
- `input` - string

OUTPUT:

A new cell with the given input text (empty by default).

new_text_cell_after(*id, input=''*)

Insert a new cell into the cell list after the cell with the given integer id. If the id is not the id of any cell, inserts a new cell at the end of the cell list.

INPUT:

- `id` - integer
- `input` - string

OUTPUT:

A new cell with the given input text (empty by default).

new_text_cell_before(*id, input=''*)

Insert a new cell into the cell list before the cell with the given integer id. If the id is not the id of any cell, inserts a new cell at the end of the cell list.

INPUT:

- `id` - integer
- `input` - string

OUTPUT:

A new cell with the given input text (empty by default).

next_block_id()

next_hidden_id()

notebook()

Return the notebook that contains this worksheet.

OUTPUT: a Notebook object.

EXAMPLES: This really returns the Notebook object that is set as a global variable of the twist module.

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.notebook()
<class 'sage.server.notebook.notebook.Notebook'>
sage: W.notebook() is sage.server.notebook.twist.notebook
True
```

owner()

ping(*username*)

plain_text(*prompts=False, banner=True*)

Return a plain-text version of the worksheet.

INPUT:

- `prompts` - if True format for inclusion in docstrings.

postprocess_output(*out, C*)

preparse(*s*)

preparse_input(*input, C*)

preparse_introspection_input(*input, C, introspect*)

preparse_nonswitched_input(*input*)

pretty_print()

Return True if output shold be pretty printed by default.

OUTPUT:

- `bool` - True of False

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.pretty_print()
False
sage: W.set_pretty_print('true')
sage: W.pretty_print()
True
```

published_version()
If this worksheet was published, return the published version of this worksheet. Otherwise, raise a ValueError.

OUTPUT: a worksheet (or raise a ValueError)

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: P = nb.publish_worksheet(W, 'admin')
sage: W.published_version() is P
True
```

publisher()
Return username of user that published this worksheet.

OUTPUT: string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: S = nb.publish_worksheet(W, 'admin')
sage: S.publisher()
'admin'
```

queue()

queue_id_list()

quit()

quit_if_idle(*timeout*)
Quit the worksheet process if it has been "idle" for more than `timeout` seconds, where idle is by definition that the worksheet has not reported back that it is actually computing. I.e., an ignored worksheet process (since the user closed their browser) is also considered idle, even if code is running.

rate(*x, comment, username*)
Set the rating on this worksheet by the given user to x and also set the given comment.

INPUT:

- `x` - integer
- `comment` - string
- `usename` - string

EXAMPLES: We create a worksheet and rate it, then look at the ratings.

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: W.rate(3, 'this is great', 'hilbert')
sage: W.ratings()
[('hilbert', 3, 'this is great')]
```

Note that only the last rating by a user counts:

```
sage: W.rate(1, 'this lacks content', 'riemann')
sage: W.rate(0, 'this lacks content', 'riemann')
sage: W.ratings()
[('hilbert', 3, 'this is great'), ('riemann', 0, 'this lacks content')]
```

rating()
Return overall aerage rating of self.

OUTPUT: float or the int -1 to mean "not rated"

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: W.rating()
-1
sage: W.rate(0, 'this lacks content', 'riemann')
sage: W.rate(3, 'this is great', 'hilbert')
sage: W.rating()
1.5
```

ratings()
Return all the ratings of this worksheet.
OUTPUT:
- `list` - a reference to the list of ratings.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: W.ratings()
[]
sage: W.rate(0, 'this lacks content', 'riemann')
sage: W.rate(3, 'this is great', 'hilbert')
sage: W.ratings()
[('riemann', 0, 'this lacks content'), ('hilbert', 3, 'this is great')]
```

record_edit(*user*)

reset_interact_state()
Reset the interact state of this worksheet.

restart_sage()
Restart Sage kernel.

revert_to_last_saved_state()

revert_to_snapshot(*name*)

sage()
Return a started up copy of Sage initialized for computations.
If this is a published worksheet, just return None, since published worksheets must not have any compute functionality.
OUTPUT: a Sage interface

satisfies_search(*search*)
Return True if all words in search are in the saved text of the worksheet.
INPUT: search is a string that describes a search query, i.e., a space-separated collections of words.
OUTPUT: True if the search is satisfied by self, i.e., all the words appear in the text version of self.

save()

save_snapshot(*user, E=None*)

set_active(*user*)
Set his worksheet to be active for the given user.
INPUT:
- `user` - string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Active Test', 'admin')
sage: W.move_to_archive('admin')
```

```
sage: W.is_active('admin')
False
sage: W.set_active('admin')
sage: W.is_active('admin')
True
```

set_auto_publish()
Sets the worksheet to be published automatically when the worksheet is saved if the worksheet isn't already set to this otherwise it is set not to.

set_cell_counter()

set_collaborators(*v*)
Set the list of collaborators to those listed in the list v of strings.

INPUT:

- v - a list of strings

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: nb.add_user('hilbert','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('test1', 'admin')
sage: W.set_collaborators(['sage', 'admin', 'hilbert', 'sage'])
```

Note that repeats are not added multiple times and admin - the owner - isn't added:

```
sage: W.collaborators()
['hilbert', 'sage']
```

set_filename(*filename*)
Set the worksheet filename (actually directory).

INPUT:

- filename - string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.filename()
'admin/0'
sage: W.set_filename('admin/10')
sage: W.filename()
'admin/10'
```

set_filename_without_owner(*nm*)
Set this worksheet filename (actually directory) by getting the owner from the pre-stored owner via `self.owner()`.

INPUT:

- nm - string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.filename()
'admin/0'
sage: W.set_filename_without_owner('5')
sage: W.filename()
'admin/5'
```

set_is_doc_worksheet(*value*)

set_name(*name*)

Set the name of this worksheet.

INPUT:

- `name` - string

EXAMPLES: We create a worksheet and change the name:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.set_name('A renamed worksheet')
sage: W.name()
'A renamed worksheet'
```

set_not_computing()

set_owner(*owner*)

set_pretty_print(*check='false'*)

Set whether or not output should be pretty printed by default.

INPUT:

- `check` - string (default: 'false'); either 'true' or 'false'.

Note: The reason the input is a string and lower case instead of a Python bool is because this gets called indirectory from javascript. (And, Jason Grout wrote this and didn't realize how unpythonic this design is - it should be redone to use True/False.)

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.set_pretty_print('false')
sage: W.pretty_print()
False
sage: W.set_pretty_print('true')
sage: W.pretty_print()
True
```

set_published_version(*filename*)

Set the published version of this worksheet to be the worksheet with given filename.

INPUT:

- `filename` - string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: P = nb.publish_worksheet(W, 'admin')  # indirect test
sage: W._Worksheet__published_version
'pub/0'
sage: W.set_published_version('pub/0')
```

set_system(*system='sage'*)

Set the math software system in which input is evaluated by default.

INPUT:

- `sysem` - string (default: 'sage')

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.set_system('magma')
sage: W.system()
'magma'
```

set_user_view(*user, x*)

Set the view on this worksheet for the given user.

INPUT:

- `user` - a string
- `x` - int, one of the variables ACTIVE, ARCHIVED, TRASH in worksheet.py

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: W.set_user_view('admin', sage.server.notebook.worksheet.ARCHIVED)
sage: W.user_view('admin') == sage.server.notebook.worksheet.ARCHIVED
True
```

set_worksheet_that_was_published(*W*)

Set the worksheet that was published to get self to W.

INPUT:

- `W` - a Worksheet

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: P = nb.publish_worksheet(W, 'admin')
sage: P.worksheet_that_was_published() is W
True
```

We fake things and make it look like P published itself:

```
sage: P.set_worksheet_that_was_published(P)
sage: P.worksheet_that_was_published() is P
True
```

show_all()

snapshot_data()

snapshot_directory()

start_next_comp()

synchro()

synchronize(*s*)

system()

Return the math software system in which by default all input to the notebook is evaluated.

OUTPUT: string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('A Test Worksheet', 'admin')
sage: W.system()
'sage'
sage: W.set_system('mathematica')
sage: W.system()
'mathematica'
```

time_idle()

time_since_last_edited()

truncated_name(*max=30*)

uncache_snapshot_data()

user_autosave_interval(*username*)

user_can_edit(*user*)

Return True if the user with given name is allowed to edit this worksheet.

INPUT:

- `user` - string

OUTPUT: bool

EXAMPLES: We create a notebook with one worksheet and two users.

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: nb.add_user('william', 'william', 'wstein@sagemath.org', force=True)
sage: W = nb.create_new_worksheet('Test', 'sage')
sage: W.user_can_edit('sage')
True
```

At first the user 'william' can't edit this worksheet:

```
sage: W.user_can_edit('william')
False
```

After adding 'william' as a collaborator he can edit the worksheet.

```
sage: W.add_collaborator('william')
sage: W.user_can_edit('william')
True
```

Clean up:

```
sage: nb.delete()
```

user_is_collaborator(*user*)

user_is_only_viewer(*user*)

user_is_viewer(*user*)

user_view(*user*)

Return the view that the given user has of this worksheet. If the user currently doesn't have a view set it to ACTIVE and return ACTIVE.

INPUT:

- `user` - a string

OUTPUT:

- `Python int` - one of ACTIVE, ARCHIVED, TRASH, which are defined in worksheet.py

EXAMPLES: We create a new worksheet and get the view, which is ACTIVE:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: W.user_view('admin')
1
sage: sage.server.notebook.worksheet.ACTIVE
1
```

Now for the admin user we move W to the archive:

```
sage: W.move_to_archive('admin')
```

The view is now archive.

```
sage: W.user_view('admin')
0
sage: sage.server.notebook.worksheet.ARCHIVED
0
```

For any other random viewer the view is set by default to ACTIVE.

```
sage: W.user_view('foo')
1
```

user_view_is (*user, x*)

Return True if the user view of user is x.

INPUT:

- `user` - a string
- `x` - int, one of the variables ACTIVE, ARCHIVED, TRASH in worksheet.py

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: W.user_view_is('admin', sage.server.notebook.worksheet.ARCHIVED)
False
sage: W.user_view_is('admin', sage.server.notebook.worksheet.ACTIVE)
True
```

viewer_names (*max=None*)

Returns a string of the non-owner viewers on this worksheet.

INPUT:

- `max` - an integer. If this is specified, then only max number of viewers are shown.

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('test1', 'admin')
sage: C = W.viewers(); C
[]
sage: C.append('sage')
sage: C.append('wstein')
sage: W.viewer_names()
'sage, wstein'
sage: W.viewer_names(max=1)
'sage, ...'
```

viewers ()

Return list of viewers of this worksheet.

OUTPUT:

- `list` - of string

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: nb.add_user('sage','sage','sage@sagemath.org',force=True)
sage: nb.add_user('hilbert','sage','sage@sagemath.org',force=True)
sage: W = nb.create_new_worksheet('test1', 'admin')
sage: W.add_viewer('hilbert')
sage: W.viewers()
```

```
['hilbert']
sage: W.add_viewer('sage')
sage: W.viewers()
['hilbert', 'sage']
```

warn_about_other_person_editing(*username, threshold*)

Check to see if another user besides username was the last to edit this worksheet during the last `threshold` seconds. If so, return True and that user name. If not, return False.

INPUT:

- `username` - user who would like to edit this file.
- `threshold` - number of seconds, so if there was no activity on this worksheet for this many seconds, then editing is considered safe.

worksheet_command(*cmd*)

worksheet_that_was_published()

Return the worksheet that was published to get this worksheet, if this worksheet was published. Otherwise just return this worksheet.

OUTPUT: Worksheet

EXAMPLES:

```
sage: nb = sage.server.notebook.notebook.Notebook(tmp_dir())
sage: W = nb.create_new_worksheet('Publish Test', 'admin')
sage: W.worksheet_that_was_published() is W
True

sage: S = nb.publish_worksheet(W, 'admin')
sage: S.worksheet_that_was_published() is S
False
sage: S.worksheet_that_was_published() is W
True
```

after_first_word(*s*)

Return everything after the first whitespace in the string s. Returns the empty string if there is nothing after the first whitespace.

INPUT:

- `s` - string

OUTPUT: a string

EXAMPLES:

```
sage: from sage.server.notebook.worksheet import after_first_word
sage: after_first_word("\%gap\n2+2\n")
'2+2\n'
sage: after_first_word("2+2")
''
```

convert_seconds_to_meaningful_time_span(*t*)

convert_time_to_string(*t*)

dictify(*s*)

INPUT:

- `s` - a string like 'in=5, out=7'

OUTPUT:

- `dict` - such as 'in':5, 'out':7

extract_first_compute_cell(*text*)
INPUT: a block of wiki-like marked up text OUTPUT:

- `meta` - meta information about the cell (as a dictionary)
- `input` - string, the input text
- `output` - string, the output text
- `end` - integer, first position after }}} in text.

extract_name(*text*)

extract_system(*text*)

extract_text_before_first_compute_cell(*text*)
OUTPUT: Everything in text up to the first {{{.

first_word(*s*)
Returns everything before the first whitespace in the string s. If there is no whitespace, then the entire string s is returned.

EXAMPLES:

```
sage: from sage.server.notebook.worksheet import first_word
sage: first_word("\%gap\n2+2\n")
'\\%gap'
sage: first_word("2+2")
'2+2'
```

format_completions_as_html(*cell_id, completions*)

ignore_prompts_and_output(*aString*)
Given a string s that defines an input block of code, if the first line begins in `sage:` (or >>>), strip out all lines that don't begin in either `sage:` (or >>>) or `...`, and remove all `sage:` (or >>>) and `...` from the beginning of the remaining lines.

TESTS:

```
sage: test1 = sage.server.notebook.worksheet.__internal_test1
sage: test1 == sage.server.notebook.worksheet.ignore_prompts_and_output(test1)
True


sage: test2 = sage.server.notebook.worksheet.__internal_test2
sage: sage.server.notebook.worksheet.ignore_prompts_and_output(test2)
'2 + 2\n'
```

init_sage_prestart(*server, ulimit*)
Set the module-scope variable _a_sage to an initialized sage server.

INPUT:

- `server, ulimit` - strings that are passed to the Sage pexpect interface constructor

EXAMPLES: The _a_sage variable is initially set to None:

```
sage: sage.server.notebook.worksheet._a_sage
```

We call init_sage_prestart and now _a_sage is a Sage instance:

```
sage: sage.server.notebook.worksheet.init_sage_prestart(None,None)
sage: sage.server.notebook.worksheet._a_sage
Sage
```

initialized_sage(*server, ulimit*)

Return one copy of a Sage compute process that has initialization code run.

INPUT:

- `server` - if sessions will be run via ssh on a remote account then this string specifies that account (passed on to the Sage pexpect interface).
- `ulimit` - string; passed to the ulimit command before running the subprocess

OUTPUT: a pexpect interface to a local or remote copy of Sage

EXAMPLES:

```
sage: S = sage.server.notebook.worksheet.initialized_sage(None,None)
sage: S
Sage
```

next_available_id(*v*)

Return smallest nonnegative integer not in v.

one_prestarted_sage(*server, ulimit*)

Return a Sage interface that has been initialized.

INPUT:

- `server, ulimit` - strings that are passed to the Sage pexpect interface constructor

OUTPUT: an interface to a running copy of Sage

If the global variable multisession is true, each call to one_prestarted_sage returns a new Sage compute instance. Otherwise it always returns the same instance.

EXAMPLES:

```
sage: sage.server.notebook.worksheet.one_prestarted_sage(None,None)
Sage
sage: sage.server.notebook.worksheet.multisession=False
sage: sage.server.notebook.worksheet.one_prestarted_sage(None,None) is sage.server.notebook.work
True
sage: sage.server.notebook.worksheet.multisession=True
```

split_search_string_into_keywords(*s*)

The point of this function is to allow for searches like this:

```
"ws 7" foo bar  Modular  '"the" end'
```

i.e., where search terms can be in quotes and the different quote types can be mixed.

INPUT:

- `s` - a string

OUTPUT:

- `list` - a list of strings

worksheet_filename(*name, owner*)

Return the relative directory name of this worksheet with given name and owner.

INPUT:

- `name` - string, which may have spaces and funny characters, which are replaced by underscores.
- `owner` - string, with no spaces or funny characters

OUTPUT: string

EXAMPLES:

```
sage: sage.server.notebook.worksheet.worksheet_filename('Example worksheet 3', 'sage10')
'sage10/Example_worksheet_3'
sage: sage.server.notebook.worksheet.worksheet_filename('Example#%&! work\sheet 3', 'sage10')
'sage10/Example_____work_sheet_3'
```

2.4 The Sage Notebook Twisted Web Server

TESTS: It is important that this file never be imported by default on startup by Sage, since it is very expensive, since importing Twisted is expensive. This doctests verifies that twist.py isn't imported on startup.

sage: os.system("sage -startuptime | grep twisted.web2 1>/dev/null") != 0 # !=0 means not found True

class AddWorksheet()

render(*ctx*)

class AdminToplevel(*cookie, username*)

userchild_conf()

userchild_home()

class AnonymousToplevel(*cookie, username*)

PasswordChecker()

render(*ctx*)

userchildFactory(*request, name*)

userchild_home()

userchild_pub()

userchild_src()

class CSS()

childFactory(*request, name*)

render(*ctx*)

class CellData(*worksheet, number*)

childFactory(*request, name*)

render(*ctx*)

class Doc (*username*)

childFactory (*request, name*)

render (*ctx*)

class DocLive (*username*)

childFactory (*request, name*)

render (*ctx*)

class DocStatic ()

childFactory (*request, name*)

render (*ctx*)

class DownloadWorksheets (*username*)

render (*ctx*)

class EmptyTrash (*username*)

render (*ctx*)

Rendering this resource (1) empties the trash, and (2) returns a message.

EXAMPLES: We create a notebook with a worksheet, put it in the trash, then empty the trash by creating and rendering this worksheet.

```
sage: n = sage.server.notebook.notebook.Notebook('notebook-test')
sage: n.add_user('sage','sage','sage@sagemath.org',force=True)
sage: W = n.new_worksheet_with_title_from_text('Sage', owner='sage')
sage: W.move_to_trash('sage')
sage: n.worksheet_names()
['sage/0']
sage: sage.server.notebook.twist.notebook = n
sage: E = sage.server.notebook.twist.EmptyTrash('sage'); E
<sage.server.notebook.twist.EmptyTrash object at ...>
sage: E.render(None)
<twisted.web2.http.Response code=200, streamlen=...>
```

Finally we verify that the trashed worksheet is gone:

```
sage: n.worksheet_names()
[]
sage: n.delete()
```

class FailedToplevel (*info, problem, username=None*)

render (*ctx*)

class ForgotPassPage ()

render (*request*)

HTMLResponse (**args, **kwds*)

Returns an HTMLResponse object whose 'Content-Type' header has been set to 'text/html; charset=utf-8

EXAMPLES: sage: from sage.server.notebook.twist import HTMLResponse sage: response = HTMLResponse(stream='<html><head><title>Test</title></head><body>Test</body></html>') sage: re-

sponse.headers <Headers: Raw: {'content-type': ['text/html; charset=utf-8']} Parsed: {'content-type': <RecalcNeeded>}>

class **Help**(*username*)

render(*ctx*)

class **History**(*username*)

render(*ctx*)

class **Images**()

childFactory(*request, name*)

render(*ctx*)

class **InvalidPage**(*msg, username*)

childFactory(*request, name*)

render(*ctx*)

class **Java**()

childFactory(*request, name*)

render(*ctx*)

class **Javascript**()

childFactory(*request, name*)

render(*ctx*)

class **JavascriptLocal**()

childFactory(*request, name*)

render(*ctx*)

class **Keyboard_js**()

childFactory(*request, browser_os*)

class **Keyboard_js_specific**(*browser_os*)

render(*ctx*)

class **ListOfUsers**(*username*)

render(*ctx*)

class **LiveHistory**(*username*)

render(*ctx*)

class **LoginResourceClass**()

childFactory(*request, name*)

render(*ctx*)

class **Logout**()

render(*ctx*)

class **Main_css**()

render(*ctx*)

class **Main_js**()

render(*ctx*)

class **NewWorksheet**(*username*)

render(*ctx*)

class **NotImplementedWorksheetOp**(*op, ws*)

render(*ctx*)

class **NotebookConf**(*username*)

render(*ctx*)

class **NotebookSettings**(*username*)

render(*ctx*)

class **ProcessNotebookSettings**()

render(*ctx*)

class **ProcessUserSettings**()

render(*ctx*)

class **PublicWorksheets**(*username*)

childFactory(*request, name*)

render(*ctx*)

class **PublicWorksheetsHome**(*username*)

childFactor(*request, name*)

class **PublishWorksheetRevision**(*worksheet, rev*)

render(*ctx*)

class **RedirectLogin**()

childFactory(*request, name*)

render(*ctx*)

class **RegConfirmation**()

render(*request*)

class **RegistrationPage**(*userdb*)

render(*request*)

class **Reset_css**()

render(*ctx*)

class **RevertToWorksheetRevision**(*worksheet, rev*)

render(*ctx*)

class **SageTex**(*username*)

childFactory(*request, name*)

render(*ctx*)

class **SendWorksheetToActive**(*username*)

action(*W*)

class **SendWorksheetToArchive**(*username*)

action(*W*)

class **SendWorksheetToFolder**(*username*)

action(*W*)

render(*ctx*)

class **SendWorksheetToStop**(*username*)

Quits each selected worksheet.

action(*W*)

class **SendWorksheetToTrash**(*username*)

action(*W*)

class **SettingsPage**(*username*)

render(*request*)

class **Source**(*path, username*)

childFactory(*request, name*)

render(*ctx*)

class **SourceBrowser**(*username*)

childFactory(*request, name*)

render(*ctx*)

class **Toplevel**(*cookie, username*)

childFactory(*request, name*)
render(*ctx*)
userchildFactory(*request, name*)

class **TrivialResource**()

render(*ctx*)

class **Upload**(*username*)

render(*ctx*)

class **UploadWorksheet**(*username*)

render(*ctx*)

class **UserToplevel**(*cookie, username*)

render(*request*)
userchildFactory(*request, name*)
userchild_doc()
userchild_download_worksheets()
userchild_emptytrash()
userchild_help()
userchild_history()
userchild_home()
userchild_live_history()
userchild_new_worksheet()
userchild_notebook_settings()
userchild_pub()
userchild_sagetex()
userchild_send_to_active()
userchild_send_to_archive()
userchild_send_to_stop()
Quits each selected worksheet.
userchild_send_to_trash()
userchild_settings()
userchild_src()
userchild_upload()
userchild_upload_worksheet()
userchild_users()

class **Worksheet**(*name, username*)

childFactory(*request, op*)

render(*ctx*)

class **WorksheetFile**(*path, username*)

childFactory(*request, name*)

render(*ctx=None*)

class **WorksheetResource**(*name, username*)

id(*ctx*)

class **Worksheet_alive**(*name, username*)

render(*ctx*)

class **Worksheet_cell_update**(*name, username*)

render(*ctx*)

class **Worksheet_cells**(*name, username*)

childFactory(*request, segment*)

render(*ctx*)

class **Worksheet_conf**(*name, username*)

render(*ctx*)

class **Worksheet_copy**(*name, username*)

render(*ctx*)

class **Worksheet_data**(*name, username*)

childFactory(*request, name*)

render(*ctx*)

class **Worksheet_datafile**(*name, username*)

render(*ctx*)

class **Worksheet_delete_all_output**(*name, username*)

render(*ctx*)

class **Worksheet_delete_cell**(*name, username*)
Deletes a notebook cell.

If there is only one cell left in a given worksheet, the request to delete that cell is ignored because there must be a least one cell at all times in a worksheet. (This requirement exists so other functions that evaluate relative to existing cells will still work, and so one can add new cells.)

render(*ctx*)

class **Worksheet_discard_and_quit**(*name, username*)
Save a snapshot of a worksheet and quit.

render(*ctx*)

class **Worksheet_do_upload_data**(*name, username*)

render(*ctx*)

class **Worksheet_download**(*name, username*)

childFactory(*request, name*)

class **Worksheet_edit**(*name, username*)
Return a window that allows the user to edit the text of the worksheet with the given filename.

render(*ctx*)

class **Worksheet_edit_published_page**(*name, username*)

render(*ctx*)

class **Worksheet_eval**(*name, username*)
Evaluate a worksheet cell.

If the request is not authorized (the requester did not enter the correct password for the given worksheet), then the request to evaluate or introspect the cell is ignored.

If the cell contains either 1 or 2 question marks at the end (not on a comment line), then this is interpreted as a request for either introspection to the documentation of the function, or the documentation of the function and the source code of the function respectively.

render(*ctx*)

class **Worksheet_hide_all**(*name, username*)

render(*ctx*)

class **Worksheet_input_settings**(*name, username*)

render(*ctx*)

class **Worksheet_interrupt**(*name, username*)

render(*ctx*)

class **Worksheet_introspect**(*name, username*)
Cell introspection. This is called when the user presses the tab key in the browser in order to introspect.

render(*ctx*)

class **Worksheet_invite_collab**(*name, username*)

render(*ctx*)

class **Worksheet_link_datafile**(*name, username*)

render(*ctx*)

class **Worksheet_new_cell_after**(*name, username*)
Adds a new cell after a given cell.

render(*ctx*)

class **Worksheet_new_cell_before**(*name, username*)
Adds a new cell before a given cell.

render(*ctx*)

class **Worksheet_new_text_cell_after**(*name, username*)
Adds a new text cell after a given cell.

render(*ctx*)

class **Worksheet_new_text_cell_before**(*name, username*)
Adds a new cell before a given cell.

render(*ctx*)

class **Worksheet_plain**(*name, username*)

render(*ctx*)

class **Worksheet_pretty_print**(*name, username*)

childFactory(*request, enable*)

class **Worksheet_print**(*name, username*)

render(*ctx*)

class **Worksheet_publish**(*name, username*)
This is a child resource of the Worksheet resource. It provides a frontend to the mangement of worksheet publication. This mangement functionality includes initializational of publication, re-publication, automated publication when a worksheet saved, and ending of publication.

render(*ctx*)

class **Worksheet_quit_sage**(*name, username*)

render(*ctx*)

class **Worksheet_rate**(*name, username*)

render(*ctx*)

class **Worksheet_rating_info**(*name, username*)

render(*ctx*)

class **Worksheet_rename**(*name, username*)

render(*ctx*)

class **Worksheet_restart_sage**(*name, username*)

render(*ctx*)

class **Worksheet_revert_to_last_saved_state**(*name, username*)

render(*ctx*)

class **Worksheet_revisions**(*name, username*)
Show a list of revisions of this worksheet.

render(*ctx*)

class **Worksheet_save**(*name, username*)
Save the contents of a worksheet after editing it in plain-text edit mode.

render(*ctx*)

class **Worksheet_save_and_close**(*name, username*)
Save a snapshot of a worksheet then quit it.

render(*ctx*)

class **Worksheet_save_and_quit**(*name, username*)
Save a snapshot of a worksheet and quit.

render(*ctx*)

class **Worksheet_save_snapshot**(*name, username*)
Save a snapshot of a worksheet.

render(*ctx*)

class **Worksheet_savedatafile**(*name, username*)

render(*ctx*)

class **Worksheet_set_cell_output_type**(*name, username*)
Set the output type of the cell.

This enables the type of output cell, such as to allowing wrapping for output that is very long.

render(*ctx*)

class **Worksheet_settings**(*name, username*)

render(*ctx*)

class **Worksheet_share**(*name, username*)

render(*ctx*)

class **Worksheet_show_all**(*name, username*)

render(*ctx*)

class **Worksheet_system**(*name, username*)

childFactory(*request, system*)

class **Worksheet_text**(*name, username*)
Return a window that allows the user to edit the text of the worksheet with the given filename.

render(*ctx*)

class **Worksheet_upload_data**(*name, username*)

render(*ctx*)

class **Worksheets**(*username*)

childFactory(*request, name*)

render(*ctx*)

class **WorksheetsAdmin**(*username*)

childFactory(*request, name*)

class **WorksheetsByUser**(*user, username*)

childFactory(*request, name*)

render(*ctx*)

render_list(*ctx*)

class **WorksheetsByUserAdmin**(*user, username*)

render(*ctx*)

do_passwords_match(*pass1, pass2*)
EXAMPLES:

```
sage: from sage.server.notebook.twist import do_passwords_match
sage: do_passwords_match('momcat', 'mothercat')
False
sage: do_passwords_match('mothercat', 'mothercat')
True
```

doc_worksheet()

encode_list(*v*)

extract_title(*html_page*)

gzip_handler(*request*)
Add gzip compression to the request if it makes sense.

init_updates()

is_valid_email(*email*)
from http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/65215

EXAMPLES:

```
sage: from sage.server.notebook.twist import is_valid_email
sage: is_valid_email('joe@washinton.gov')
True
sage: is_valid_email('joe.washington.gov')
False
```

is_valid_password(*password, username*)
Return True if and only if `password` is valid, i.e., is between 6 and 32 characters long, doesn't contain space(s), and doesn't contain `username`.

EXAMPLES:

```
sage: from sage.server.notebook.twist import is_valid_password
sage: is_valid_password('uip@un7!', None)
True
sage: is_valid_password('markusup89', None)
True
sage: is_valid_password('8u7', None)
False
sage: is_valid_password('fUmDagaz8LmtonAowjSe0Pvu9C5Gvr6eKcC6wsAT', None)
False
sage: is_valid_password('rrcF !u78!', None)
False
sage: is_valid_password('markusup89', 'markus')
False
```

is_valid_username(*username*)

Returns True if and only if `username` is valid, i.e., starts with a letter, is between 4 and 32 characters long, and contains only letters, numbers, underscores, and and one dot (.).

EXAMPLES:

```
sage: from sage.server.notebook.twist import is_valid_username
```

`username` must start with a letter

```
sage: is_valid_username('mark10')
True
sage: is_valid_username('10mark')
False
```

`username` must be between 4 and 32 characters long

```
sage: is_valid_username('bob')
False
sage: is_valid_username('I_love_computer_science_and_maths') #33 characters long
False
```

`username` must not have more than one dot (.)

```
sage: is_valid_username('david.andrews')
True
sage: is_valid_username('david.m.andrews')
False
sage: is_valid_username('math125.TA.5')
False
```

`username` must not have any spaces

```
sage: is_valid_username('David Andrews')
False
sage: is_valid_username('David M. Andrews')
False

sage: is_valid_username('sarah_andrews')
True

sage: is_valid_username('TA-1')
False
sage: is_valid_username('math125-TA')
False

sage: is_valid_username('dandrews@sagemath.org')
False
```

message(*msg, cont=None*)

notebook_idle_check()

notebook_save_check()

notebook_updates()

redirect(*url*)

render_worksheet_list(*args, pub, username*)
Returns a rendered worksheet listing.

INPUT:

- `args` - ctx.args where ctx is the dict passed into a resource's render method
- `pub` - boolean, True if this is a listing of public worksheets
- `username` - the user whose worksheets we are listing

OUTPUT: a string

set_cookie(*cookie*)

user_type(*username*)

word_wrap_cols()

worksheet_revision_publish(*worksheet, rev, username*)

worksheet_revision_revert(*worksheet, rev, username*)

2.5 Javascript (AJAX) Component of sage Notebook

Javascript (AJAX) Component of sage Notebook

AUTHORS:

- William Stein
- Tom Boothby
- Alex Clemesha

This file is one big raw triple-quoted string that contains a bunch of javascript. This javascript is inserted into the head of the notebook web page.

class JSKeyCode(*key, alt, ctrl, shift*)

js_test()

class JSKeyHandler()
This class is used to make javascript functions to check for specific keyevents.

add(*name, key='', alt=False, ctrl=False, shift=False*)
Similar to code{set_key(...)}, but this instead checks if there is an existing keycode by the specified name, and associates the specified key combination to that name in addition. This way, if different browsers don't catch one keycode, multiple keycodes can be assigned to the same test.

all_tests()
Builds all tests currently in the handler. Returns a string of javascript code which defines all functions.

set(*name, key='', alt=False, ctrl=False, shift=False*)
Add a named keycode to the handler. When built by code{all_tests()}, it can be called in javascript by code{key_<key_name>(event_object)}. The function returns true if the keycode numbered by the code{key} parameter was pressed with the appropriate modifier keys, false otherwise.

async_lib()

javascript()
Return javascript library for the Sage Notebook.

jmol_lib()

notebook_lib()

2.6 Customization of the Notebook

Customization of the Notebook

2.7 Sage Notebook CSS

Sage Notebook CSS

css(*color='default'*)

Return the CSS header used by the Sage Notebook.

INPUT:

- `color` - string or pair of html colors, e.g., 'gmail' 'grey' `('#ff0000', '#0000ff')`

EXAMPLES:

```
sage: import sage.server.notebook.css as c
sage: type(c.css())
<type 'str'>
```

2.8 Support for the Notebook (introspection and setup)

AUTHORS:

- William Stein (much of this code is from IPython).

completions(*s, globs, format=False, width=90, system='None'*)

Return a list of completions in the context of globs.

cython_import(*filename, verbose=False, compile_message=False, use_cache=False, create_local_c_file=True*)

INPUT:

- `filename` - name of a file that contains cython code

OUTPUT:

- `module` - the module that contains the compiled cython code.

Raises an `ImportError` exception if anything goes wrong.

cython_import_all(*filename, globals, verbose=False, compile_message=False, use_cache=False, create_local_c_file=True*)

INPUT:

- `filename` - name of a file that contains cython code

OUTPUT: changes globals using the attributes of the Cython module that do not begin with an underscore.

Raises an `ImportError` exception if anything goes wrong.

docstring(*obj_name, globs, system='sage'*)
Format `obj_name`'s docstring for printing in Sage notebook.

AUTHORS:

- William Stein: partly taken from IPython for use in Sage
- Nick Alexander: extensions

get_rightmost_identifier(*s*)

help(*obj*)
Display help on s.

Note: This a wrapper around the builtin help. If formats the output as HTML without word wrap, which looks better in the notebook.

INPUT:

- `s` - Python object, module, etc.

OUTPUT: prints out help about s; it's often more more extensive than foo?

TESTS:

```
sage: import numpy.linalg
sage: sage.server.support.help(numpy.linalg.norm)
<html><table notracebacks bgcolor="#386074" cellpadding=10 cellspacing=10><tr><td bgcolor="#f5f5
   <a target='_new' href='cell://docs-....html'>Click to open help window</a>&nbs
<br></font></tr></td></table></html>
```

init(*object_directory=None, globs={}*)
Initialize Sage for use with the web notebook interface.

load_session(*v, filename, state*)

save_session(*filename*)

setup_systems(*globs*)

source_code(*s, globs, system='sage'*)
Format obj's source code for printing in Sage notebook.

AUTHORS:

- William Stein: partly taken from IPython for use in Sage
- Nick Alexander: extensions

syseval(*system, cmd, dir=None*)

INPUT: **system – an object with an eval method that takes as input a cmd (a string), and two dictionaries:** sage_globals and locals.

dir – an otional directory to change to before calling system.eval.

OUTPUT: The output of system.eval is returned.

EXAMPLES: sage: from sage.misc.python import python sage: sage.server.support.syseval(python, '2+4/3') 3 '' sage: sage.server.support.syseval(python, 'import os; os.chdir(".")') '' sage: sage.server.support.syseval(python, 'import os; os.chdir(1,2,3)') Traceback (most recent call last): ... TypeError: chdir() takes exactly 1 argument (3 given) sage: sage.server.support.syseval(gap, "2+3") '5'

tabulate(*v, width=90, ncols=3*)

variables(*with_types=True*)

2.9 Sage Notebook: Introspection

TODO: - add support for grabbing source code from pyrex functions (even if not perfect is better than nothing). - png or mathml output format for docstring

introspect (*S, query, format='html'*)

Return introspection from a given query string.

INPUT:

- `S` - a Sage0 object, i.e., an interface to a running instance of Python with the Sage libraries loaded
- `query` - a string: - if has no '?' then return completion list - if begins or ends in one '?' return docstring - if begins or ends in '??' return source code
- `format` - (string) 'html', 'png', 'none' (only html is implemented right now!)

CHAPTER

THREE

SYMBOLIC CALCULUS

3.1 File: sage/symbolic/ring.pyx (starting at line 1)

class NumpyToSRMorphism()

class SymbolicRing()

Symbolic Ring, parent object for all symbolic expressions.

characteristic()

Return the characteristic of the symbolic ring, which is 0.

OUTPUT:

- a Sage integer

EXAMPLES:

```
sage: c = SR.characteristic(); c
0
sage: type(c)
<type 'sage.rings.integer.Integer'>
```

is_exact()

Return False, because there are approximate elements in the symbolic ring.

EXAMPLES:

```
sage: SR.is_exact()
False
```

Here is an inexact element.

```
sage: SR(1.9393)
1.93930000000000
```

is_field()

Returns True, since the symbolic expression ring is (for the most part) a field.

EXAMPLES:

```
sage: SR.is_field()
True
```

pi()

EXAMPLES:

```
sage: SR.pi() is pi
True
```

symbol()

EXAMPLES:

```
sage: SR.symbol("asdfasdfasdf")
asdfasdfasdf
```

var()

Return the symbolic variable defined by x as an element of the symbolic ring.

EXAMPLES:

```
sage: zz = SR.var('zz'); zz
zz
sage: type(zz)
<type 'sage.symbolic.expression.Expression'>
sage: t = SR.var('theta2'); t
theta2
```

wild()

Return the n-th wild-card for pattern matching and substitution.

INPUT:

- n - a nonnegative integer

OUTPUT:

- i^{th} wildcard expression

EXAMPLES:

```
sage: x,y = var('x,y')
sage: w0 = SR.wild(0); w1 = SR.wild(1)
sage: pattern = sin(x)*w0*w1^2; pattern
$0*$1^2*sin(x)
sage: f = atan(sin(x)*3*x^2); f
arctan(3*x^2*sin(x))
sage: f.has(pattern)
True
sage: f.subs(pattern == x^2)
arctan(x^2)
```

class **UnderscoreSageMorphism**()

is_SymbolicExpressionRing()

Returns True if *R* is the symbolic expression ring.

EXAMPLES:

```
sage: from sage.symbolic.ring import is_SymbolicExpressionRing
sage: is_SymbolicExpressionRing(ZZ)
False
sage: is_SymbolicExpressionRing(SR)
True
```

is_SymbolicVariable()

Returns True if x is a variable.

EXAMPLES:

```
sage: from sage.symbolic.ring import is_SymbolicVariable
sage: is_SymbolicVariable(x)
True
sage: is_SymbolicVariable(x+2)
False
```

the_SymbolicRing()
Return the unique symbolic ring object.

(This is mainly used for unpickling.)

EXAMPES:

```
sage: sage.symbolic.ring.the_SymbolicRing()
Symbolic Ring
sage: sage.symbolic.ring.the_SymbolicRing() is sage.symbolic.ring.the_SymbolicRing()
True
sage: sage.symbolic.ring.the_SymbolicRing() is SR
True
```

var()
EXAMPLES:

```
sage: from sage.symbolic.ring import var
sage: var("x y z")
(x, y, z)
sage: var("x,y,z")
(x, y, z)
sage: var("x , y , z")
(x, y, z)
sage: var("z")
z
```

3.2 Symbolic Expressions

RELATIONAL EXPRESSIONS:

We create a relational expression:

```
sage: x = var('x')
sage: eqn = (x-1)^2 <= x^2 - 2*x + 3
sage: eqn.subs(x == 5)
16 <= 18
```

Notice that squaring the relation squares both sides.

```
sage: eqn^2
(x - 1)^4 <= (x^2 - 2*x + 3)^2
sage: eqn.expand()
x^2 - 2*x + 1 <= x^2 - 2*x + 3
```

The can transform a true relational into a false one:

```
sage: eqn = SR(-5) < SR(-3); eqn
-5 < -3
sage: bool(eqn)
True
sage: eqn^2
25 < 9
sage: bool(eqn^2)
False
```

We can do arithmetic with relationals:

```
sage: e = x+1 <= x-2
sage: e + 2
x + 3 <= x
sage: e - 1
x <= x - 3
sage: e*(-1)
-x - 1 <= -x + 2
sage: (-2)*e
-2*x - 2 <= -2*x + 4
sage: e*5
5*x + 5 <= 5*x - 10
sage: e/5
1/5*x + 1/5 <= 1/5*x - 2/5
sage: 5/e
5/(x + 1) <= 5/(x - 2)
sage: e/(-2)
-1/2*x - 1/2 <= -1/2*x + 1
sage: -2/e
-2/(x + 1) <= -2/(x - 2)
```

We can even add together two relations, so long as the operators are the same:

```
sage: (x^3 + x <= x - 17)  + (-x <= x - 10)
x^3 <= 2*x - 27
```

Here they aren't:

```
sage: (x^3 + x <= x - 17)  + (-x >= x - 10)
...
TypeError: incompatible relations
```

ARBITRARY SAGE ELEMENTS:

You can work symbolically with any Sage datatype. This can lead to nonsense if the data type is strange, e.g., an element of a finite field (at present).

We mix Singular variables with symbolic variables:

```
sage: R.<u,v> = QQ[]
sage: var('a,b,c')
(a, b, c)
sage: expand((u + v + a + b + c)^2)
a^2 + 2*a*b + 2*a*c + 2*a*u + 2*a*v + b^2 + 2*b*c + 2*b*u + 2*b*v + c^2 + 2*c*u + 2*c*v + u^2 + 2*u*v
```

TESTS:

Test jacobian on pynac expressions. #5546

```
sage: var('x,y')
(x, y)
sage: f = x + y
sage: jacobian(f, [x,y])
[1 1]
```

Test if matrices work #5546

```
sage: var('x,y,z')
(x, y, z)
sage: M = matrix(2,2,[x,y,z,x])
sage: v = vector([x,y])
sage: M * v
(x^2 + y^2, x*y + x*z)
sage: v*M
(x^2 + y*z, 2*x*y)
```

class Expression()

Order()

Order, as in big oh notation.

OUTPUT: symbolic expression

EXAMPLES: sage: n = var('n') sage: (17*n^3).Order() Order(n^3)

add_to_both_sides()

Returns a relation obtained by adding x to both sides of this relation.

EXAMPLES:

```
sage: var('x y z')
(x, y, z)
sage: eqn = x^2 + y^2 + z^2 <= 1
sage: eqn.add_to_both_sides(-z^2)
x^2 + y^2 <= -z^2 + 1
sage: eqn.add_to_both_sides(I)
x^2 + y^2 + z^2 + I <= (I + 1)
```

arccos()

Return the arc cosine of self.

EXAMPLES:

```
sage: x.arccos()
arccos(x)
sage: SR(1).arccos()
0
sage: SR(1/2).arccos()
1/3*pi
sage: SR(0.4).arccos()
arccos(0.400000000000000)
```

Use .n() to get a numerical approximation:

```
sage: SR(0.4).arccos().n()
1.15927948072741
sage: plot(lambda x: SR(x).arccos(), -1,1)
```

arccosh()

Return the inverse hyperbolic cosine of self.

EXAMPLES:

```
sage: x.arccosh()
arccosh(x)
sage: SR(0).arccosh()
1/2*I*pi
sage: SR(1/2).arccosh()
arccosh(1/2)
```

```
sage: SR(CDF(1/2)).arccosh()
arccosh(0.5)
```

Use .n() to get a numerical approximation:

```
sage: SR(CDF(1/2)).arccosh().n()
1.0471975512*I
sage: maxima('acosh(0.5)')
1.047197551196598*%i
```

arcsin()

Return the arcsin of x, i.e., the number y between -pi and pi such that sin(y) == x.

EXAMPLES:

```
sage: x.arcsin()
arcsin(x)
sage: SR(0.5).arcsin()
arcsin(0.500000000000000)
```

Use .n() to get a numerical approximation:

```
sage: SR(0.5).arcsin().n()
0.523598775598299

sage: SR(0.999).arcsin()
arcsin(0.999000000000000)
sage: SR(-0.999).arcsin()
-arcsin(0.999000000000000)
sage: SR(0.999).arcsin().n()
1.52607123962616
```

arcsinh()

Return the inverse hyperbolic sine of self.

EXAMPLES:

```
sage: x.arcsinh()
arcsinh(x)
sage: SR(0).arcsinh()
0
sage: SR(1).arcsinh()
arcsinh(1)
sage: SR(1.0).arcsinh()
arcsinh(1.00000000000000)
```

Use .n() to get a numerical approximation:

```
sage: SR(1.0).arcsinh().n()
0.881373587019543
sage: maxima('asinh(1.0)')
0.881373587019543
```

Sage automatically applies certain identies:: sage: SR(3/2).arcsinh().cosh() 1/2*sqrt(13)

arctan()

Return the arc tangent of self.

EXAMPLES:

```
sage: x = var('x')
sage: x.arctan()
arctan(x)
sage: SR(1).arctan()
```

```
1/4*pi
sage: SR(1/2).arctan()
arctan(1/2)
sage: SR(0.5).arctan()
arctan(0.500000000000000)
```

Use .n() to get a numerical approximation:

```
sage: SR(0.5).arctan().n()
0.463647609000806
sage: plot(lambda x: SR(x).arctan(), -20,20)
```

arctan2()

Return the inverse of the 2-variable tan function on self and x.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: x.arctan2(y)
arctan2(x, y)
sage: SR(1/2).arctan2(1/2)
1/4*pi
sage: maxima.eval('atan2(1/2,1/2)')
'%pi/4'

sage: SR(-0.7).arctan2(SR(-0.6))
-pi + arctan(1.16666666666667)
```

Use .n() to get a numerical approximation:

```
sage: SR(-0.7).arctan2(SR(-0.6)).n()
-2.27942259892257
```

TESTS:

We compare a bunch of different evaluation points between Sage and Maxima:

```
sage: float(SR(0.7).arctan2(0.6))
0.8621700546672264
sage: maxima('atan2(0.7,0.6)')
.862170054667226...
sage: float(SR(0.7).arctan2(-0.6))
2.2794225989225669
sage: maxima('atan2(0.7,-0.6)')
2.279422598922567
sage: float(SR(-0.7).arctan2(0.6))
-0.8621700546672264
sage: maxima('atan2(-0.7,0.6)')
-.862170054667226...
sage: float(SR(-0.7).arctan2(-0.6))
-2.2794225989225669
sage: maxima('atan2(-0.7,-0.6)')
-2.279422598922567
sage: float(SR(0).arctan2(-0.6))
3.1415926535897931
sage: maxima('atan2(0,-0.6)')
3.141592653589793
sage: float(SR(0).arctan2(0.6))
0.0
sage: maxima('atan2(0,0.6)')
0.0
```

```
sage: SR(0).arctan2(0)
0

sage: SR(I).arctan2(1)
arctan2(I, 1)
sage: SR(CDF(0,1)).arctan2(1)
arctan2(1.0*I, 1)
sage: SR(1).arctan2(CDF(0,1))
arctan2(1, 1.0*I)
```

arctanh()

Return the inverse hyperbolic tangent of self.

EXAMPLES:

```
sage: x.arctanh()
arctanh(x)
sage: SR(0).arctanh()
0
sage: SR(1/2).arctanh()
arctanh(1/2)
sage: SR(0.5).arctanh()
arctanh(0.500000000000000)
```

Use .n() to get a numerical approximation:

```
sage: SR(0.5).arctanh().n()
0.549306144334055
sage: SR(0.5).arctanh().tanh()
0.500000000000000
sage: maxima('atanh(0.5)')
.5493061443340...
```

args()

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = x + y
sage: f.arguments()
(x, y)

sage: g = f.function(x)
sage: g.arguments()
(x,)
```

arguments()

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = x + y
sage: f.arguments()
(x, y)

sage: g = f.function(x)
sage: g.arguments()
(x,)
```

assume()

Assume that this equation holds. This is relevant for symbolic integration, among other things.

EXAMPLES: We call the assume method to assume that $x > 2$:

```
sage: (x > 2).assume()
```

Bool returns True below if the inequality is *definitely* known to be True.

```
sage: bool(x > 0)
True
sage: bool(x < 0)
False
```

This may or may not be True, so bool returns False:

```
sage: bool(x > 3)
False
```

TESTS:

```
sage: v,c = var('v,c')
sage: assume(c != 0)
sage: integral((1+v^2/c^2)^3/(1-v^2/c^2)^(3/2),v)
-75/8*sqrt(c^2)*arcsin(sqrt(c^2)*v/c^2) - 17/8*v^3/(sqrt(-v^2/c^2 + 1)*c^2) - 1/4*v^5/(sqrt(
```

binomial()

Return binomial coefficient "self choose k".

OUTPUT: symbolic expression

EXAMPLES: sage: var('x, y') (x, y) sage: SR(5).binomial(SR(3)) 10 sage: x.binomial(SR(3)) 1/6*x^3 - 1/2*x^2 + 1/3*x sage: x.binomial(y) binomial(x,y)

coeff()

Returns the coefficient of s^n in this symoblic expression.

INPUT:

- s - expression
- n - integer, default 1

OUTPUT:

- coefficient of s^n

Sometimes it may be necessary to expand or factor first, since this is not done automatically.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
sage: f.collect(x)
x^3*sin(x*y) + (a + y + 1/y)*x + 2*sin(x*y)/x + 100
sage: f.coefficient(x,0)
100
sage: f.coefficient(x,-1)
2*sin(x*y)
sage: f.coefficient(x,1)
a + y + 1/y
sage: f.coefficient(x,2)
0
sage: f.coefficient(x,3)
sin(x*y)
sage: f.coefficient(x^3)
sin(x*y)
sage: f.coefficient(sin(x*y))
x^3 + 2/x
sage: f.collect(sin(x*y))
```

```
(x^3 + 2/x)*sin(x*y) + a*x + x*y + x/y + 100

sage: var('a, x, y, z')
(a, x, y, z)
sage: f = (a*sqrt(2))*x^2 + sin(y)*x^(1/2) + z^z
sage: f.coefficient(sin(y))
sqrt(x)
sage: f.coefficient(x^2)
sqrt(2)*a
sage: f.coefficient(x^(1/2))
sin(y)
sage: f.coefficient(1)
0
sage: f.coefficient(x, 0)
sqrt(x)*sin(y) + z^z
```

coefficient()

Returns the coefficient of s^n in this symoblic expression.

INPUT:

- s - expression
- n - integer, default 1

OUTPUT:

- coefficient of s^n

Sometimes it may be necessary to expand or factor first, since this is not done automatically.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
sage: f.collect(x)
x^3*sin(x*y) + (a + y + 1/y)*x + 2*sin(x*y)/x + 100
sage: f.coefficient(x,0)
100
sage: f.coefficient(x,-1)
2*sin(x*y)
sage: f.coefficient(x,1)
a + y + 1/y
sage: f.coefficient(x,2)
0
sage: f.coefficient(x,3)
sin(x*y)
sage: f.coefficient(x^3)
sin(x*y)
sage: f.coefficient(sin(x*y))
x^3 + 2/x
sage: f.collect(sin(x*y))
(x^3 + 2/x)*sin(x*y) + a*x + x*y + x/y + 100

sage: var('a, x, y, z')
(a, x, y, z)
sage: f = (a*sqrt(2))*x^2 + sin(y)*x^(1/2) + z^z
sage: f.coefficient(sin(y))
sqrt(x)
sage: f.coefficient(x^2)
sqrt(2)*a
```

```
sage: f.coefficient(x^(1/2))
sin(y)
sage: f.coefficient(1)
0
sage: f.coefficient(x, 0)
sqrt(x)*sin(y) + z^z
```

coefficients()

Coefficients of this symbolic expression as a polynomial in x.

INPUT:

- x - optional variable

OUTPUT:

- A list of pairs (expr, n), where expr is a symbolic expression and n is a power.

EXAMPLES:

```
sage: var('x, y, a')
(x, y, a)
sage: p = x^3 - (x-3)*(x^2+x) + 1
sage: p.coefficients()
[[1, 0], [3, 1], [2, 2]]
sage: p = expand((x-a*sqrt(2))^2 + x + 1); p
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
sage: p.coefficients(a)
[[x^2 + x + 1, 0], [-2*sqrt(2)*x, 1], [2, 2]]
sage: p.coefficients(x)
[[2*a^2 + 1, 0], [-2*sqrt(2)*a + 1, 1], [1, 2]]
```

A polynomial with wacky exponents:

```
sage: p = (17/3*a)*x^(3/2) + x*y + 1/x + x^x
sage: p.coefficients(x)
[[1, -1], [x^x, 0], [y, 1], [17/3*a, 3/2]]
```

coeffs()

Coefficients of this symbolic expression as a polynomial in x.

INPUT:

- x - optional variable

OUTPUT:

- A list of pairs (expr, n), where expr is a symbolic expression and n is a power.

EXAMPLES:

```
sage: var('x, y, a')
(x, y, a)
sage: p = x^3 - (x-3)*(x^2+x) + 1
sage: p.coefficients()
[[1, 0], [3, 1], [2, 2]]
sage: p = expand((x-a*sqrt(2))^2 + x + 1); p
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
sage: p.coefficients(a)
[[x^2 + x + 1, 0], [-2*sqrt(2)*x, 1], [2, 2]]
sage: p.coefficients(x)
[[2*a^2 + 1, 0], [-2*sqrt(2)*a + 1, 1], [1, 2]]
```

A polynomial with wacky exponents:

```
sage: p = (17/3*a)*x^(3/2) + x*y + 1/x + x^x
sage: p.coefficients(x)
[[1, -1], [x^x, 0], [y, 1], [17/3*a, 3/2]]
```

collect()

INPUT:

- s - a symbol

OUTPUT:

- expression

EXAMPLES:

```
sage: var('x,y,z')
(x, y, z)
sage: f = 4*x*y + x*z + 20*y^2 + 21*y*z + 4*z^2 + x^2*y^2*z^2
sage: f.collect(x)
x^2*y^2*z^2 + (4*y + z)*x + 20*y^2 + 21*y*z + 4*z^2
sage: f.collect(y)
(x^2*z^2 + 20)*y^2 + (4*x + 21*z)*y + x*z + 4*z^2
sage: f.collect(z)
(x^2*y^2 + 4)*z^2 + (x + 21*y)*z + 4*x*y + 20*y^2
```

collect_common_factors()

EXAMPLES:

```
sage: var('x')
x
sage: (x/(x^2 + x)).collect_common_factors()
1/(x + 1)
```

combine()

Returns a simplified version of this symbolic expression by combining all terms with the same denominator into a single term.

EXAMPLES:

```
sage: var('x, y, a, b, c')
(x, y, a, b, c)
sage: f = x*(x-1)/(x^2 - 7) + y^2/(x^2-7) + 1/(x+1) + b/a + c/a; f
(x - 1)*x/(x^2 - 7) + y^2/(x^2 - 7) + b/a + c/a + 1/(x + 1)
sage: f.combine()
((x - 1)*x + y^2)/(x^2 - 7) + (b + c)/a + 1/(x + 1)
```

conjugate()

Return the complex conjugate of this symbolic expression.

EXAMPLES:

```
sage: a = 1 + 2*I
sage: a.conjugate()
-2*I + 1
sage: a = sqrt(2) + 3^(1/3)*I; a
sqrt(2) + I*3^(1/3)
sage: a.conjugate()
sqrt(2) - I*3^(1/3)

sage: SR(CDF.0).conjugate()
-1.0*I
sage: x.conjugate()
conjugate(x)
```

```
sage: SR(RDF(1.5)).conjugate()
1.5
sage: SR(float(1.5)).conjugate()
1.50000000000000
sage: SR(I).conjugate()
-I
sage: ( 1+I  + (2-3*I)*x).conjugate()
(3*I + 2)*conjugate(x) - I + 1
```

cos()

Return the cosine of self.

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: cos(x^2 + y^2)
cos(x^2 + y^2)
sage: cos(sage.symbolic.constants.pi)
-1
sage: cos(SR(1))
cos(1)
sage: cos(SR(RealField(150)(1)))
0.540302305868139717400936607442976603732310420
```

In order to get a numeric approximation use .n():

```
sage: SR(RR(1)).cos().n()
0.540302305868140
sage: SR(float(1)).cos().n()
0.540302305868140
```

cosh()

Return cosh of self.

We have $sinh(x) = (e^{x} + e^{-x})/2$.

EXAMPLES:

```
sage: x.cosh()
cosh(x)
sage: SR(1).cosh()
cosh(1)
sage: SR(0).cosh()
1
sage: SR(1.0).cosh()
cosh(1.00000000000000)
```

Use .n() to get a numerical approximation:

```
sage: SR(1.0).cosh().n()
1.54308063481524
sage: maxima('cosh(1.0)')
1.543080634815244
sage: SR(1.0000000000000000000000000).cosh()
cosh(1.0000000000000000000000000)
sage: SR(1).cosh().n(90)
1.54308063481524377847790560
sage: SR(RIF(1)).cosh()
cosh(1)
sage: SR(RIF(1)).cosh().n()
1.543080634815244?
```

csgn()

Return the sign of self, which is -1 if self < 0, 0 if self == 0, and 1 if self > 0, or unevaluated when self is a nonconstant symbolic expression.

It can be somewhat arbitrary when self is not real.

EXAMPLES: sage: x = var('x') sage: SR(-2).csgn() -1 sage: SR(0.0).csgn() 0 sage: SR(10).csgn() 1 sage: x.csgn() csgn(x) sage: SR(CDF.0).csgn() 1 sage: SR(I).csgn() 1

default_variable()

Return the default variable, which is by definition the first variable in self, or x is there are no variables in self. The result is cached.

EXAMPLES:

```
sage: sqrt(2).default_variable()
x
sage: x, theta, a = var('x, theta, a')
sage: f = x^2 + theta^3 - a^x
sage: f.default_variable()
a
```

Note that this is the first *variable*, not the first *argument*:

```
sage: f(theta, a, x) = a + theta^3
sage: f.default_variable()
a
sage: f.variables()
(a, theta)
sage: f.arguments()
(theta, a, x)
```

degree()

Return the exponent of the highest nonnegative power of s in self.

OUTPUT:

- an integer >= 0.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y^10 + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + 2*sin(x*y)/x + x/y^10 + 100
sage: f.degree(x)
3
sage: f.degree(y)
1
sage: f.degree(sin(x*y))
1
sage: (x^-3+y).degree(x)
0
```

denominator()

Returns the denominator of this symbolic expression. If the expression is not a quotient, then this will just return 1.

EXAMPLES:

```
sage: x, y, z, theta = var('x, y, z, theta')
sage: f = (sqrt(x) + sqrt(y) + sqrt(z))/(x^10 - y^10 - sqrt(theta))
sage: f.denominator()
sqrt(theta) - x^10 + y^10
```

```
sage: y = var('y')
sage: g = x + y/(x + 2); g
x + y/(x + 2)
sage: g.numerator()
x + y/(x + 2)
sage: g.denominator()
1
```

derivative()

Returns the derivative of this expressions with respect to the variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See Also:

_derivative()

EXAMPLES:

```
sage: var("x y")
(x, y)
sage: t = (x^2+y)^2
sage: t.derivative(x)
4*(x^2 + y)*x
sage: t.derivative(x, 2)
12*x^2 + 4*y
sage: t.derivative(x, 2, y)
4
sage: t.derivative(y)
2*x^2 + 2*y

sage: t = sin(x+y^2)*tan(x*y)
sage: t.derivative(x)
(tan(x*y)^2 + 1)*y*sin(y^2 + x) + cos(y^2 + x)*tan(x*y)
sage: t.derivative(y)
(tan(x*y)^2 + 1)*x*sin(y^2 + x) + 2*y*cos(y^2 + x)*tan(x*y)

sage: h = sin(x)/cos(x)
sage: derivative(h,x,x,x)
6*sin(x)^4/cos(x)^4 + 8*sin(x)^2/cos(x)^2 + 2
sage: derivative(h,x,3)
6*sin(x)^4/cos(x)^4 + 8*sin(x)^2/cos(x)^2 + 2

sage: var('x, y')
(x, y)
sage: u = (sin(x) + cos(y))*(cos(x) - sin(y))
sage: derivative(u,x,y)
sin(x)*sin(y) - cos(x)*cos(y)
sage: f = ((x^2+1)/(x^2-1))^(1/4)
sage: g = derivative(f, x); g # this is a complex expression
1/2*(x/(x^2 - 1) - (x^2 + 1)*x/(x^2 - 1)^2)/((x^2 + 1)/(x^2 - 1))^(3/4)
sage: g.factor()
-x/((x^2 - 1)^(5/4)*(x^2 + 1)^(3/4))

sage: y = var('y')
sage: f = y^(sin(x))
sage: derivative(f, x)
y^sin(x)*log(y)*cos(x)
```

```
sage: g(x) = sqrt(5-2*x)
sage: g_3 = derivative(g, x, 3); g_3(2)
-3

sage: f = x*e^(-x)
sage: derivative(f, 100)
x*e^(-x) - 100*e^(-x)

sage: g = 1/(sqrt((x^2-1)*(x+5)^6))
sage: derivative(g, x)
-((x + 5)^6*x + 3*(x + 5)^5*(x^2 - 1))/((x + 5)^6*(x^2 - 1))^(3/2)
```

TESTS:

```
sage: t.derivative()
...
ValueError: No differentiation variable specified.
```

diff()

Returns the derivative of this expressions with respect to the variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See Also:

_derivative()

EXAMPLES:

```
sage: var("x y")
(x, y)
sage: t = (x^2+y)^2
sage: t.derivative(x)
4*(x^2 + y)*x
sage: t.derivative(x, 2)
12*x^2 + 4*y
sage: t.derivative(x, 2, y)
4
sage: t.derivative(y)
2*x^2 + 2*y

sage: t = sin(x+y^2)*tan(x*y)
sage: t.derivative(x)
(tan(x*y)^2 + 1)*y*sin(y^2 + x) + cos(y^2 + x)*tan(x*y)
sage: t.derivative(y)
(tan(x*y)^2 + 1)*x*sin(y^2 + x) + 2*y*cos(y^2 + x)*tan(x*y)

sage: h = sin(x)/cos(x)
sage: derivative(h,x,x,x)
6*sin(x)^4/cos(x)^4 + 8*sin(x)^2/cos(x)^2 + 2
sage: derivative(h,x,3)
6*sin(x)^4/cos(x)^4 + 8*sin(x)^2/cos(x)^2 + 2

sage: var('x, y')
(x, y)
sage: u = (sin(x) + cos(y))*(cos(x) - sin(y))
sage: derivative(u,x,y)
sin(x)*sin(y) - cos(x)*cos(y)
sage: f = ((x^2+1)/(x^2-1))^(1/4)
sage: g = derivative(f, x); g # this is a complex expression
1/2*(x/(x^2 - 1) - (x^2 + 1)*x/(x^2 - 1)^2)/((x^2 + 1)/(x^2 - 1))^(3/4)
```

```
sage: g.factor()
-x/((x^2 - 1)^(5/4)*(x^2 + 1)^(3/4))

sage: y = var('y')
sage: f = y^(sin(x))
sage: derivative(f, x)
y^sin(x)*log(y)*cos(x)

sage: g(x) = sqrt(5-2*x)
sage: g_3 = derivative(g, x, 3); g_3(2)
-3

sage: f = x*e^(-x)
sage: derivative(f, 100)
x*e^(-x) - 100*e^(-x)

sage: g = 1/(sqrt((x^2-1)*(x+5)^6))
sage: derivative(g, x)
-((x + 5)^6*x + 3*(x + 5)^5*(x^2 - 1))/((x + 5)^6*(x^2 - 1))^(3/2)
```

TESTS:

```
sage: t.derivative()
...
ValueError: No differentiation variable specified.
```

differentiate()

Returns the derivative of this expressions with respect to the variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See Also:

_derivative()

EXAMPLES:

```
sage: var("x y")
(x, y)
sage: t = (x^2+y)^2
sage: t.derivative(x)
4*(x^2 + y)*x
sage: t.derivative(x, 2)
12*x^2 + 4*y
sage: t.derivative(x, 2, y)
4
sage: t.derivative(y)
2*x^2 + 2*y

sage: t = sin(x+y^2)*tan(x*y)
sage: t.derivative(x)
(tan(x*y)^2 + 1)*y*sin(y^2 + x) + cos(y^2 + x)*tan(x*y)
sage: t.derivative(y)
(tan(x*y)^2 + 1)*x*sin(y^2 + x) + 2*y*cos(y^2 + x)*tan(x*y)

sage: h = sin(x)/cos(x)
sage: derivative(h,x,x,x)
6*sin(x)^4/cos(x)^4 + 8*sin(x)^2/cos(x)^2 + 2
sage: derivative(h,x,3)
6*sin(x)^4/cos(x)^4 + 8*sin(x)^2/cos(x)^2 + 2
```

```
sage: var('x, y')
(x, y)
sage: u = (sin(x) + cos(y))*(cos(x) - sin(y))
sage: derivative(u,x,y)
sin(x)*sin(y) - cos(x)*cos(y)
sage: f = ((x^2+1)/(x^2-1))^(1/4)
sage: g = derivative(f, x); g # this is a complex expression
1/2*(x/(x^2 - 1) - (x^2 + 1)*x/(x^2 - 1)^2)/((x^2 + 1)/(x^2 - 1))^(3/4)
sage: g.factor()
-x/((x^2 - 1)^(5/4)*(x^2 + 1)^(3/4))

sage: y = var('y')
sage: f = y^(sin(x))
sage: derivative(f, x)
y^sin(x)*log(y)*cos(x)

sage: g(x) = sqrt(5-2*x)
sage: g_3 = derivative(g, x, 3); g_3(2)
-3

sage: f = x*e^(-x)
sage: derivative(f, 100)
x*e^(-x) - 100*e^(-x)

sage: g = 1/(sqrt((x^2-1)*(x+5)^6))
sage: derivative(g, x)
-((x + 5)^6*x + 3*(x + 5)^5*(x^2 - 1))/((x + 5)^6*(x^2 - 1))^(3/2)
```

TESTS:

```
sage: t.derivative()
...
ValueError: No differentiation variable specified.
```

divide_both_sides()

Returns a relation obtained by dividing both sides of this relation by *x*.

Note: The *checksign* keyword argument is currently ignored and is included for backward compatibility reasons only.

EXAMPLES:

```
sage: theta = var('theta')
sage: eqn =   (x^3 + theta < sin(x*theta))
sage: eqn.divide_both_sides(theta, checksign=False)
(x^3 + theta)/theta < sin(theta*x)/theta
sage: eqn.divide_both_sides(theta)
(x^3 + theta)/theta < sin(theta*x)/theta
sage: eqn/theta
(x^3 + theta)/theta < sin(theta*x)/theta
```

exp()

Return exponential function of self, i.e., e to the power of self.

EXAMPLES:

```
sage: x.exp()
e^x
sage: SR(0).exp()
1
sage: SR(1/2).exp()
e^(1/2)
```

```
sage: SR(0.5).exp()
e^0.500000000000000
sage: (pi*I).exp()
-1
```

Use .n() to get a numerical approximation:

```
sage: SR(0.5).exp().n()
1.64872127070013
sage: math.exp(0.5)
1.6487212707001282

sage: SR(0.5).exp().log()
0.500000000000000
```

exp_simplify()

Simplifies this symbolic expression, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables

DETAILS: This uses the Maxima radcan() command. From the Maxima documentation: "All functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, produces a regular form. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference can be simplified by radcan to zero. For some expressions radcan is quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial fraction expansions of exponents."

ALIAS: radical_simplify, simplify_radical, simplify_log, log_simplify, exp_simplify, simplify_exp are all the same

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)

sage: f = log(x*y)
sage: f.simplify_radical()
log(x) + log(y)

sage: f = (log(x+x^2)-log(x))^a/log(1+x)^(a/2)
sage: f.simplify_radical()
log(x + 1)^(1/2*a)

sage: f = (e^x-1)/(1+e^(x/2))
sage: f.simplify_exp()
e^(1/2*x) - 1
```

expand()

Expand this symbolic expression. Products of sums and exponentiated sums are multiplied out, numerators of rational expressions which are sums are split into their respective terms, and multiplications are distributed over addition at all levels.

EXAMPLES:

We expand the expression $(x-y)^5$ using both method and functional notation.

```
sage: x,y = var('x,y')
sage: a = (x-y)^5
sage: a.expand()
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
sage: expand(a)
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
```

We expand some other expressions:

```
sage: expand((x-1)^3/(y-1))
x^3/(y - 1) - 3*x^2/(y - 1) + 3*x/(y - 1) - 1/(y - 1)
sage: expand((x+sin((x+y)^2))^2)
x^2 + 2*x*sin((x + y)^2) + sin((x + y)^2)^2
```

We can expand individual sides of a relation:

```
sage: a = (16*x-13)^2 == (3*x+5)^2/2
sage: a.expand()
256*x^2 - 416*x + 169 == 9/2*x^2 + 15*x + 25/2
sage: a.expand('left')
256*x^2 - 416*x + 169 == 1/2*(3*x + 5)^2
sage: a.expand('right')
(16*x - 13)^2 == 9/2*x^2 + 15*x + 25/2
```

TESTS:

sage: var('x,y') (x, y) sage: ((x + (2/3)*y)^3).expand() x^3 + 2*x^2*y + 4/3*x*y^2 + 8/27*y^3 sage: expand((x*sin(x) - cos(y)/x)^2) x^2*sin(x)^2 - 2*sin(x)*cos(y) + cos(y)^2/x^2 sage: f = (x-y)*(x+y); f (x - y)*(x + y) sage: f.expand() x^2 - y^2

expand_rational()

Expand this symbolic expression. Products of sums and exponentiated sums are multiplied out, numerators of rational expressions which are sums are split into their respective terms, and multiplications are distributed over addition at all levels.

EXAMPLES:

We expand the expression $(x-y)^5$ using both method and functional notation.

```
sage: x,y = var('x,y')
sage: a = (x-y)^5
sage: a.expand()
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
sage: expand(a)
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
```

We expand some other expressions:

```
sage: expand((x-1)^3/(y-1))
x^3/(y - 1) - 3*x^2/(y - 1) + 3*x/(y - 1) - 1/(y - 1)
sage: expand((x+sin((x+y)^2))^2)
x^2 + 2*x*sin((x + y)^2) + sin((x + y)^2)^2
```

We can expand individual sides of a relation:

```
sage: a = (16*x-13)^2 == (3*x+5)^2/2
sage: a.expand()
256*x^2 - 416*x + 169 == 9/2*x^2 + 15*x + 25/2
sage: a.expand('left')
256*x^2 - 416*x + 169 == 1/2*(3*x + 5)^2
sage: a.expand('right')
(16*x - 13)^2 == 9/2*x^2 + 15*x + 25/2
```

TESTS:

sage: var('x,y') (x, y) sage: ((x + (2/3)*y)^3).expand() x^3 + 2*x^2*y + 4/3*x*y^2 + 8/27*y^3 sage: expand((x*sin(x) - cos(y)/x)^2) x^2*sin(x)^2 - 2*sin(x)*cos(y) + cos(y)^2/x^2 sage: f = (x-y)*(x+y); f (x - y)*(x + y) sage: f.expand() x^2 - y^2

expand_trig()

Expands trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in self. For best results, self should already be expanded.

INPUT:

- `full` - (default: False) To enhance user control of simplification, this function expands only one level at a time by default, expanding sums of angles or multiple angles. To obtain full expansion into sines and cosines immediately, set the optional parameter full to True.
- `half_angles` - (default: False) If True, causes half-angles to be simplified away.
- `plus` - (default: True) Controls the sum rule; expansion of sums (e.g. 'sin(x + y)') will take place only if plus is True.
- `times` - (default: True) Controls the product rule, expansion of products (e.g. sin(2*x)) will take place only if times is True.

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: sin(5*x).expand_trig()
sin(x)^5 - 10*sin(x)^3*cos(x)^2 + 5*sin(x)*cos(x)^4
sage: cos(2*x + var('y')).expand_trig()
-sin(2*x)*sin(y) + cos(2*x)*cos(y)
```

We illustrate various options to this function:

```
sage: f = sin(sin(3*cos(2*x))*x)
sage: f.expand_trig()
sin(-(sin(cos(2*x))^3 - 3*sin(cos(2*x))*cos(cos(2*x))^2)*x)
sage: f.expand_trig(full=True)
sin(((sin(sin(x)^2)*cos(cos(x)^2) - sin(cos(x)^2)*cos(sin(x)^2))^3 - 3*(sin(sin(x)^2)*cos(co
sage: sin(2*x).expand_trig(times=False)
sin(2*x)
sage: sin(2*x).expand_trig(times=True)
2*sin(x)*cos(x)
sage: sin(2 + x).expand_trig(plus=False)
sin(x + 2)
sage: sin(2 + x).expand_trig(plus=True)
sin(2)*cos(x) + sin(x)*cos(2)
sage: sin(x/2).expand_trig(half_angles=False)
sin(1/2*x)
sage: sin(x/2).expand_trig(half_angles=True)
1/2*sqrt(-cos(x) + 1)*sqrt(2)
```

ALIASES:

`trig_expand()` and `expand_trig()` are the same

`factor`()

Factors self, containing any number of variables or functions, into factors irreducible over the integers.

INPUT:

- `self` - a symbolic expression
- `dontfactor` - list (default: []), a list of variables with respect to which factoring is not to occur. Factoring also will not take place with respect to any variables which are less important (using the variable ordering assumed for CRE form) than those on the 'dontfactor' list.

EXAMPLES:

```
sage: x,y,z = var('x, y, z')
sage: (x^3-y^3).factor()
(x - y)*(x^2 + x*y + y^2)
sage: factor(-8*y - 4*x + z^2*(2*y + x))
(z - 2)*(z + 2)*(x + 2*y)
sage: f = -1 - 2*x - x^2 + y^2 + 2*x*y^2 + x^2*y^2
sage: F = factor(f/(36*(1 + 2*y + y^2)), dontfactor=[x]); F
1/36*(y - 1)*(x^2 + 2*x + 1)/(y + 1)
```

If you are factoring a polynomial with rational coefficients (and dontfactor is empty) the factorization is done using Singular instead of Maxima, so the following is very fast instead of dreadfully slow:

```
sage: var('x,y')
(x, y)
sage: (x^99 + y^99).factor()
(x + y)*(x^2 - x*y + y^2)*(x^6 - x^3*y^3 + y^6)*...
```

factor_list()

Returns a list of the factors of self, as computed by the factor command.

INPUT:

- self - a symbolic expression
- dontfactor - see docs for factor()

Note: If you already have a factored expression and just want to get at the individual factors, use _factor_list() instead.

EXAMPLES:

```
sage: var('x, y, z')
(x, y, z)
sage: f = x^3-y^3
sage: f.factor()
(x - y)*(x^2 + x*y + y^2)
```

Notice that the -1 factor is separated out:

```
sage: f.factor_list()
[(x - y, 1), (x^2 + x*y + y^2, 1)]
```

We factor a fairly straightforward expression:

```
sage: factor(-8*y - 4*x + z^2*(2*y + x)).factor_list()
[(z - 2, 1), (z + 2, 1), (x + 2*y, 1)]
```

A more complicated example:

```
sage: var('x, u, v')
(x, u, v)
sage: f = expand((2*u*v^2-v^2-4*u^3)^2 * (-u)^3 * (x-sin(x))^3)
sage: f.factor()
-(x - sin(x))^3*(4*u^3 - 2*u*v^2 + v^2)^2*u^3
sage: g = f.factor_list(); g
[(x - sin(x), 3), (4*u^3 - 2*u*v^2 + v^2, 2), (u, 3), (-1, 1)]
```

This function also works for quotients:

```
sage: f = -1 - 2*x - x^2 + y^2 + 2*x*y^2 + x^2*y^2
sage: g = f/(36*(1 + 2*y + y^2)); g
1/36*(x^2*y^2 + 2*x*y^2 - x^2 + y^2 - 2*x - 1)/(y^2 + 2*y + 1)
sage: g.factor(dontfactor=[x])
1/36*(y - 1)*(x^2 + 2*x + 1)/(y + 1)
sage: g.factor_list(dontfactor=[x])
[(y - 1, 1), (y + 1, -1), (x^2 + 2*x + 1, 1), (1/36, 1)]
```

This example also illustrates that the exponents do not have to be integers:

```
sage: f = x^(2*sin(x)) * (x-1)^(sqrt(2)*x); f
(x - 1)^(sqrt(2)*x)*x^(2*sin(x))
sage: f.factor_list()
[(x - 1, sqrt(2)*x), (x, 2*sin(x))]
```

factorial()

Return the factorial of self.

OUTPUT: symbolic expression

EXAMPLES: sage: var('x, y') (x, y) sage: SR(5).factorial() 120 sage: x.factorial() factorial(x) sage: (x^2+y^3).factorial() factorial(x^2 + y^3)

find()

Find all occurences of the given pattern in this expression.

Note that once a subexpression matches the pattern, the search doesn't extend to subexpressions of it.

EXAMPLES:

```
sage: var('x,y,z,a,b')
(x, y, z, a, b)
sage: w0 = SR.wild(0); w1 = SR.wild(1)

sage: (sin(x)*sin(y)).find(sin(w0))
[sin(x), sin(y)]

sage: ((sin(x)+sin(y))*(a+b)).expand().find(sin(w0))
[sin(x), sin(y)]

sage: (1+x+x^2+x^3).find(x)
[x]
sage: (1+x+x^2+x^3).find(x^w0)
[x^3, x^2]

sage: (1+x+x^2+x^3).find(y)
[]

# subexpressions of a match are not listed
sage: ((x^y)^z).find(w0^w1)
[(x^y)^z]
```

find_maximum_on_interval()

Numerically find the maximum of the expression `self` on the interval [a,b] (or [b,a]) along with the point at which the maximum is attained.

See the documentation for `self.find_minimum_on_interval` for more details.

EXAMPLES:

```
sage: f = x*cos(x)
sage: f.find_maximum_on_interval(0,5)
(0.5610963381910451, 0.8603335890...)
sage: f.find_maximum_on_interval(0,5, tol=0.1, maxfun=10)
(0.561090323458081..., 0.857926501456...)
```

find_minimum_on_interval()

Numerically find the minimum of the expression `self` on the interval [a,b] (or [b,a]) and the point at which it attains that minimum. Note that `self` must be a function of (at most) one variable.

INPUT:

- `var` - variable (default: first variable in self)
- `a,b` - endpoints of interval on which to minimize self.
- `tol` - the convergence tolerance
- `maxfun` - maximum function evaluations

OUTPUT:

- `minval` - (float) the minimum value that self takes on in the interval [a,b]
- `x` - (float) the point at which self takes on the minimum value

EXAMPLES:

```
sage: f = x*cos(x)
sage: f.find_minimum_on_interval(1, 5)
(-3.288371395590..., 3.4256184695...)
sage: f.find_minimum_on_interval(1, 5, tol=1e-3)
(-3.288371361890..., 3.4257507903...)
sage: f.find_minimum_on_interval(1, 5, tol=1e-2, maxfun=10)
(-3.288370845983..., 3.4250840220...)
sage: show(f.plot(0, 20))
sage: f.find_minimum_on_interval(1, 15)
(-9.477294259479..., 9.5293344109...)
```

ALGORITHM:

Uses `scipy.optimize.fminbound` which uses Brent's method.

AUTHORS:

- William Stein (2007-12-07)

find_root()

Numerically find a root of self on the closed interval [a,b] (or [b,a]) if possible, where self is a function in the one variable.

INPUT:

- `a, b` - endpoints of the interval
- `var` - optional variable
- `xtol, rtol` - the routine converges when a root is known to lie within xtol of the value return. Should be = 0. The routine modifies this to take into account the relative precision of doubles.
- `maxiter` - integer; if convergence is not achieved in maxiter iterations, an error is raised. Must be = 0.
- `full_output` - bool (default: False), if True, also return object that contains information about convergence.

EXAMPLES:

Note that in this example both f(-2) and f(3) are positive, yet we still find a root in that interval:

```
sage: f = x^2 - 1
sage: f.find_root(-2, 3)
1.0
sage: f.find_root(-2, 3, x)
1.0
sage: z, result = f.find_root(-2, 3, full_output=True)
sage: result.converged
True
sage: result.flag
'converged'
sage: result.function_calls
11
sage: result.iterations
10
sage: result.root
1.0
```

More examples:

```
sage: (sin(x) + exp(x)).find_root(-10, 10)
-0.588532743981862...
sage: sin(x).find_root(-1,1)
0.0
sage: (1/tan(x)).find_root(3,3.5)
3.1415926535...
```

An example with a square root:

```
sage: f = 1 + x + sqrt(x+2); f.find_root(-2,10)
-1.6180339887498949
```

Some examples that Ted Kosan came up with:

```
sage: t = var('t')
sage: v = 0.004*(9600*e^(-(1200*t)) - 2400*e^(-(300*t)))
sage: v.find_root(0, 0.002)
0.001540327067911417...

sage: a = .004*(8*e^(-(300*t)) - 8*e^(-(1200*t)))*(720000*e^(-(300*t)) - 11520000*e^(-(1200*
```

There is a 0 very close to the origin:

```
sage: show(plot(a, 0, .002), xmin=0, xmax=.002)
```

Using solve does not work to find it:

```
sage: a.solve(t)
[]
```

However `find_root` works beautifully:

```
sage: a.find_root(0,0.002)
0.0004110514049349...
```

We illustrate that root finding is only implemented in one dimension:

```
sage: x, y = var('x,y')
sage: (x-y).find_root(-2,2)
...
NotImplementedError: root finding currently only implemented in 1 dimension.
```

TESTS:

Test the special case that failed for the first attempt to fix #3980:

```
sage: t = var('t')
sage: find_root(1/t - x,0,2)
...
NotImplementedError: root finding currently only implemented in 1 dimension.
```

forget()

Forget the given constraint.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: forget()
sage: assume(x>0, y < 2)
sage: assumptions()
[x > 0, y < 2]
sage: forget(y < 2)
sage: assumptions()
[x > 0]
```

full_simplify()

Applies simplify_trig, simplify_rational, and simplify_radical to self (in that order).

ALIAS: simplfy_full and full_simplify are the same.

EXAMPLES:

```
sage: a = log(8)/log(2)
sage: a.simplify_full()
3

sage: f = sin(x)^2 + cos(x)^2
sage: f.simplify_full()
1

sage: f = sin(x/(x^2 + x))
sage: f.simplify_full()
sin(1/(x + 1))
```

function()

Return a callable symbolic expression with the given variables.

EXAMPLES:

We will use several symbolic variables in the examples below:

```
sage: var('x, y, z, t, a, w, n')
(x, y, z, t, a, w, n)

sage: u = sin(x) + x*cos(y)
sage: g = u.function(x,y)
sage: g(x,y)
x*cos(y) + sin(x)
sage: g(t,z)
t*cos(z) + sin(t)
sage: g(x^2, x^y)
x^2*cos(x^y) + sin(x^2)

sage: f = (x^2 + sin(a*w)).function(a,x,w); f
(a, x, w) |--> x^2 + sin(a*w)
sage: f(1,2,3)
sin(3) + 4
```

Using the `function()` method we can obtain the above function f, but viewed as a function of different variables:

```
sage: h = f.function(w,a); h
(w, a) |--> x^2 + sin(a*w)
```

This notation also works:

```
sage: h(w,a) = f
sage: h
(w, a) |--> x^2 + sin(a*w)
```

You can even make a symbolic expression f into a function by writing `f(x,y) = f`:

```
sage: f = x^n + y^n; f
x^n + y^n
sage: f(x,y) = f
sage: f
(x, y) |--> x^n + y^n
sage: f(2,3)
2^n + 3^n
```

gamma()

Return the Gamma function evaluated at self.

EXAMPLES: sage: x = var('x') sage: x.gamma() gamma(x) sage: SR(2).gamma() 1 sage: SR(10).gamma() 362880 sage: SR(10.0r).gamma() gamma(10.0000000000000)

Use .n() to get a numerical approximation:

```
sage: SR(10.0r).gamma().n()
362880.000000000
sage: SR(CDF(1,1)).gamma()
gamma(1.0 + 1.0*I)
sage: SR(CDF(1,1)).gamma().n()
0.498015668118 - 0.154949828302*I

sage: gp('gamma(1+I)') # 32-bit
0.49801566811835604271369111746219809195 - 0.15494982830181068512495513048388660520*I

sage: gp('gamma(1+I)') # 64-bit
0.49801566811835604271369111746219809195 - 0.15494982830181068512495513048388660520*I

sage: set_verbose(-1); plot(lambda x: SR(x).gamma(), -6,4).show(ymin=-3,ymax=3)
```

gcd()

Return the gcd of self and b, which must be integers or polynomials over the rational numbers.

TODO: I tried the massive gcd from http://trac.sagemath.org/sage_trac/ticket/694 on Ginac dies after about 10 seconds. Singular easily does that GCD now. Since Ginac only handles poly gcd over QQ, we should change ginac itself to use Singular.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: SR(10).gcd(SR(15))
5
sage: (x^3 - 1).gcd(x-1)
x - 1
sage: (x^3 - 1).gcd(x^2+x+1)
x^2 + x + 1
sage: (x^3 - sage.symbolic.constants.pi).gcd(x-sage.symbolic.constants.pi)
...
RuntimeError: gcd: arguments must be polynomials over the rationals
sage: gcd(x^3 - y^3, x-y)
-x + y
sage: gcd(x^100-y^100, x^10-y^10)
-x^10 + y^10
sage: gcd(expand( (x^2+17*x+3/7*y)*(x^5 - 17*y + 2/3) ), expand((x^13+17*x+3/7*y)*(x^5 - 17*
1/7*x^5 - 17/7*y + 2/21
```

gradient()

Compute the gradient of a symbolic function.

This function returns a vector whose components are the derivatives of the original function with respect to the arguments of the original function. Alternatively, you can specify the variables as a list.

EXAMPLES:

```
sage: x,y = var('x y')
sage: f = x^2+y^2
sage: f.gradient()
(2*x, 2*y)
sage: g(x,y) = x^2+y^2
sage: g.gradient()
((x, y) |--> 2*x, (x, y) |--> 2*y)
sage: n = var('n')
sage: f(x,y) = x^n+y^n
sage: f.gradient()
((x, y) |--> n*x^(n - 1), (x, y) |--> n*y^(n - 1))
```

```
sage: f.gradient([y,x])
((x, y) |--> n*y^(n - 1), (x, y) |--> n*x^(n - 1))
```

has()

EXAMPLES:

```
sage: var('x,y,a'); w0 = SR.wild(); w1 = SR.wild()
(x, y, a)
sage: (x*sin(x + y + 2*a)).has(y)
True
```

Here "x+y" is not a subexpression of "x+y+2*a" (which has the subexpressions "x", "y" and "2*a"):

```
sage: (x*sin(x + y + 2*a)).has(x+y)
False
sage: (x*sin(x + y + 2*a)).has(x + y + w0)
True
```

The following fails because "2*(x+y)" automatically gets converted to "2*x+2*y" of which "x+y" is not a subexpression:

```
sage: (x*sin(2*(x+y) + 2*a)).has(x+y)
False
```

Although x^1==x and x^0==1, neither "x" nor "1" are actually of the form "x^something":

```
sage: (x+1).has(x^w0)
False
```

Here is another possible pitfall, where the first expression matches because the term "-x" has the form "(-1)*x" in GiNaC. To check whether a polynomial contains a linear term you should use the coeff() function instead.

```
sage: (4*x^2 - x + 3).has(w0*x)
True
sage: (4*x^2 + x + 3).has(w0*x)
False
sage: (4*x^2 + x + 3).has(x)
True
sage: (4*x^2 - x + 3).coeff(x,1)
-1
sage: (4*x^2 + x + 3).coeff(x,1)
1
```

hessian()

Compute the hessian of a function. This returns a matrix components are the 2nd partial derivatives of the original function.

EXAMPLES:

```
sage: x,y = var('x y')
sage: f = x^2+y^2
sage: f.hessian()
[2 0]
[0 2]
sage: g(x,y) = x^2+y^2
sage: g.hessian()
[(x, y) |--> 2 (x, y) |--> 0]
[(x, y) |--> 0 (x, y) |--> 2]
```

imag()

Return the imaginary part of this symbolic expression.

EXAMPLES:

```
sage: sqrt(-2).imag_part()
sqrt(2)
```

We simplify $\ln(\exp(z))$ to z for $-\pi < \mathrm{Im}(z) <= \pi$:

```
sage: z = var('z')
sage: f = log(exp(z))
sage: assume(-pi < imag(z))
sage: assume(imag(z) <= pi)
sage: f
log(e^z)
sage: f.simplify()
z
sage: forget()
```

A more symbolic example:

```
sage: var('a, b')
(a, b)
sage: f = log(a + b*I)
sage: f.imag_part()
arctan2(real_part(b) + imag_part(a), real_part(a) - imag_part(b))
```

TESTS:

```
sage: x = var('x')
sage: x.imag_part()
imag_part(x)
sage: SR(2+3*I).imag_part()
3
sage: SR(CC(2,3)).imag_part()
3.00000000000000
sage: SR(CDF(2,3)).imag_part()
3.0
```

imag_part()

Return the imaginary part of this symbolic expression.

EXAMPLES:

```
sage: sqrt(-2).imag_part()
sqrt(2)
```

We simplify $\ln(\exp(z))$ to z for $-\pi < \mathrm{Im}(z) <= \pi$:

```
sage: z = var('z')
sage: f = log(exp(z))
sage: assume(-pi < imag(z))
sage: assume(imag(z) <= pi)
sage: f
log(e^z)
sage: f.simplify()
z
sage: forget()
```

A more symbolic example:

```
sage: var('a, b')
(a, b)
sage: f = log(a + b*I)
sage: f.imag_part()
arctan2(real_part(b) + imag_part(a), real_part(a) - imag_part(b))
```

TESTS:

```
sage: x = var('x')
sage: x.imag_part()
imag_part(x)
sage: SR(2+3*I).imag_part()
3
sage: SR(CC(2,3)).imag_part()
3.00000000000000
sage: SR(CDF(2,3)).imag_part()
3.0
```

integral()

Compute the integral of self. Please see sage.calculus.calculus.integral for more details.

EXAMPLES:

```
sage: sin(x).integral(x,0,3)
-cos(3) + 1
sage: sin(x).integral(x)
-cos(x)
```

integrate()

Compute the integral of self. Please see sage.calculus.calculus.integral for more details.

EXAMPLES:

```
sage: sin(x).integral(x,0,3)
-cos(3) + 1
sage: sin(x).integral(x)
-cos(x)
```

inverse_laplace()

Return inverse Laplace transform of self. See sage.calculus.calculus.inverse_laplace

EXAMPLES:

```
sage: var('w, m')
(w, m)
sage: f = (1/(w^2+10)).inverse_laplace(w, m); f
1/10*sqrt(10)*sin(sqrt(10)*m)
```

is_polynomial()

Return True if self is a polynomial in the given variable.

EXAMPLES:

```
sage: var('x,y,z')
(x, y, z)
sage: t = x^2 + y; t
x^2 + y
sage: t.is_polynomial(x)
True
sage: t.is_polynomial(y)
True
sage: t.is_polynomial(z)
True

sage: t = sin(x) + y; t
y + sin(x)
sage: t.is_polynomial(x)
False
sage: t.is_polynomial(y)
True
```

```
sage: t.is_polynomial(sin(x))
True
```

is_relational()
Return True if self is a relational expression.
EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 == x^2 - 2*x + 3
sage: eqn.is_relational()
True
sage: sin(x).is_relational()
False
```

is_unit()
Return True if this expression is a unit of the symbolic ring.
EXAMPLES:

```
sage: SR(1).is_unit()
True
sage: SR(-1).is_unit()
True
sage: SR(0).is_unit()
False
```

iterator()
Return an iterator over the arguments of this expression.
EXAMPLES:

```
sage: x,y,z = var('x,y,z')
sage: list((x+y+z).iterator())
[x, y, z]
sage: list((x*y*z).iterator())
[x, y, z]
sage: list((x^y*z*(x+y)).iterator())
[x + y, x^y, z]
```

laplace()
Return Lapalace transform of self. See sage.calculus.calculus.laplace
EXAMPLES:

```
sage: var('x,s,z')
(x, s, z)
sage: (z + exp(x)).laplace(x, s)
z/s + 1/(s - 1)
```

leading_coeff()
Return the leading coefficient of s in self.
EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
sage: f.leading_coefficient(x)
sin(x*y)
sage: f.leading_coefficient(y)
x
```

```
sage: f.leading_coefficient(sin(x*y))
x^3 + 2/x
```

leading_coefficient()

Return the leading coefficient of s in self.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
sage: f.leading_coefficient(x)
sin(x*y)
sage: f.leading_coefficient(y)
x
sage: f.leading_coefficient(sin(x*y))
x^3 + 2/x
```

left()

If self is a relational expression, return the left hand side of the relation. Otherwise, raise a ValueError.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 == x^2 - 2*x + 3
sage: eqn.left_hand_side()
(x - 1)^2
sage: eqn.lhs()
(x - 1)^2
sage: eqn.left()
(x - 1)^2
```

left_hand_side()

If self is a relational expression, return the left hand side of the relation. Otherwise, raise a ValueError.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 == x^2 - 2*x + 3
sage: eqn.left_hand_side()
(x - 1)^2
sage: eqn.lhs()
(x - 1)^2
sage: eqn.left()
(x - 1)^2
```

lgamma()

Return the log-gamma function evaluated at self. This is the logarithm of gamma of self, where gamma is a complex function such that gamma(n) equals factorial(n-1).

EXAMPLES: sage: x = var('x') sage: x.lgamma() lgamma(x) sage: SR(2).lgamma() 0 sage: SR(5).lgamma() log(24) sage: SR(5-1).factorial().log() log(24) sage: set_verbose(-1); plot(lambda x: SR(x).lgamma(), -7,8, plot_points=1000).show() sage: math.exp(0.5) 1.6487212707001282 sage: plot(lambda x: (SR(x).exp() - SR(-x).exp())/2 - SR(x).sinh(), -1, 1)

lhs()

If self is a relational expression, return the left hand side of the relation. Otherwise, raise a ValueError.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 == x^2 - 2*x + 3
```

```
sage: eqn.left_hand_side()
(x - 1)^2
sage: eqn.lhs()
(x - 1)^2
sage: eqn.left()
(x - 1)^2
```

limit()

Return a symbolic limit. See sage.calculus.calculus.limit

EXAMPLES:

```
sage: (sin(x)/x).limit(x=0)
1
```

log()

Return the logarithm of self.

EXAMPLES:

```
sage: x, y = var('x, y')
sage: x.log()
log(x)
sage: (x^y + y^x).log()
log(x^y + y^x)
sage: SR(0).log()
...
RuntimeError: log_eval(): log(0)
sage: SR(1).log()
0
sage: SR(1/2).log()
log(1/2)
sage: SR(0.5).log()
log(0.500000000000000)
```

Use .n() to get a numerical approximation:

```
sage: SR(0.5).log().n()
-0.693147180559945
sage: SR(0.5).log().exp()
0.500000000000000
sage: math.log(0.5)
-0.69314718055994529
sage: plot(lambda x: SR(x).log(), 0.1,10)
```

log_simplify()

Simplifies this symbolic expression, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables

DETAILS: This uses the Maxima radcan() command. From the Maxima documentation: "All functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, produces a regular form. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference can be simplified by radcan to zero. For some expressions radcan is quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial fraction expansions of exponents."

ALIAS: radical_simplify, simplify_radical, simplify_log, log_simplify, exp_simplify, simplify_exp are all the same

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
```

```
sage: f = log(x*y)
sage: f.simplify_radical()
log(x) + log(y)

sage: f = (log(x+x^2)-log(x))^a/log(1+x)^(a/2)
sage: f.simplify_radical()
log(x + 1)^(1/2*a)

sage: f = (e^x-1)/(1+e^(x/2))
sage: f.simplify_exp()
e^(1/2*x) - 1
```

low_degree()

Return the exponent of the lowest nonpositive power of s in self.

OUTPUT:

- an integer <= 0.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y^10 + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + 2*sin(x*y)/x + x/y^10 + 100
sage: f.low_degree(x)
-1
sage: f.low_degree(y)
-10
sage: f.low_degree(sin(x*y))
0
sage: (x^3+y).low_degree(x)
0
```

match()

See http://www.ginac.de/tutorial/Pattern-matching-and-advanced-substitutions.html

EXAMPLES:

```
sage: var('x,y,z,a,b,c,d,e,f')
(x, y, z, a, b, c, d, e, f)
sage: w0 = SR.wild(0); w1 = SR.wild(1); w2 = SR.wild(2)
sage: ((x+y)^a).match((x+y)^a)
True
sage: ((x+y)^a).match((x+y)^b)
False
sage: ((x+y)^a).match(w0^w1)
True
sage: ((x+y)^a).match(w0^w0)
False
sage: ((x+y)^(x+y)).match(w0^w0)
True
sage: ((a+b)*(a+c)).match((a+w0)*(a+w1))
True
sage: ((a+b)*(a+c)).match((w0+b)*(w0+c))
True
sage: ((a+b)*(a+c)).match((w0+w1)*(w0+w2))     # surprising?
False
sage: (a*(x+y)+a*z+b).match(a*w0+w1)
True
sage: (a+b+c+d+e+f).match(c)
False
```

```
sage: (a+b+c+d+e+f).has(c)
True
sage: (a+b+c+d+e+f).match(c+w0)
True
sage: (a+b+c+d+e+f).match(c+e+w0)
True
sage: (a+b).match(a+b+w0)
True
sage: (a*b^2).match(a^w0*b^w1)
False
sage: (a*b^2).match(a*b^w1)
True
sage: (x*x.arctan2(x^2)).match(w0*w0.arctan2(w0^2))
True
```

minpoly()

Return the minimal polynomial of this symbolic expression.

EXAMPLES:

```
sage: golden_ratio.minpoly()
x^2 - x - 1
```

multiply_both_sides()

Returns a relation obtained by multiplying both sides of this relation by x.

Note: The *checksign* keyword argument is currently ignored and is included for backward compatibility reasons only.

EXAMPLES:

```
sage: var('x,y'); f = x + 3 < y - 2
(x, y)
sage: f.multiply_both_sides(7)
7*x + 21 < 7*y - 14
sage: f.multiply_both_sides(-1/2)
-1/2*x - 3/2 < -1/2*y + 1
sage: f*(-2/3)
-2/3*x - 2 < -2/3*y + 4/3
sage: f*(-pi)
-(x + 3)*pi < -(y - 2)*pi
```

Since the direction of the inequality never changes when doing arithmetic with equations, you can multiply or divide the equation by a quantity with unknown sign:

```
sage: f*(1+I)
(I + 1)*x + 3*I + 3 < (I + 1)*y - 2*I - 2
sage: f = sqrt(2) + x == y^3
sage: f.multiply_both_sides(I)
I*x + I*sqrt(2) == I*y^3
sage: f.multiply_both_sides(-1)
-x - sqrt(2) == -y^3
```

Note that the direction of the following inequalities is not reversed:

```
sage: (x^3 + 1 > 2*sqrt(3)) * (-1)
-x^3 - 1 > -2*sqrt(3)
sage: (x^3 + 1 >= 2*sqrt(3)) * (-1)
-x^3 - 1 >= -2*sqrt(3)
sage: (x^3 + 1 <= 2*sqrt(3)) * (-1)
-x^3 - 1 <= -2*sqrt(3)
```

n()

Return a numerical approximation this symbolic expression as either a real or complex number with at least the requested number of bits or digits of precision.

EXAMPLES:

```
sage: sin(x).subs(x=5).n()
-0.958924274663138
sage: sin(x).subs(x=5).n(100)
-0.95892427466313846889315440616
sage: sin(x).subs(x=5).n(digits=50)
-0.95892427466313846889315440615599397335246154396460
sage: zeta(x).subs(x=2).numerical_approx(digits=50)
1.6449340668482264364724151666460251892189499012068

sage: cos(3).numerical_approx(200)
-0.98999249660044545727157279473126130239367909661558832881409
sage: numerical_approx(cos(3), digits=10)
-0.9899924966
sage: (i + 1).numerical_approx(32)
1.00000000 + 1.00000000*I
sage: (pi + e + sqrt(2)).numerical_approx(100)
7.2740880444219335226246195788
```

TESTS:

We test the evaluation of different infinities available in pynac:

```
sage: t = x - oo; t
-Infinity
sage: t.n()
-infinity
sage: t = x + oo; t
+Infinity
sage: t.n()
+infinity
sage: t = x - unsigned_infinity; t
Infinity
sage: t.n()
+infinity
```

nintegral()

Compute the numerical integral of self. Please see sage.calculus.calculus.nintegral for more details.

EXAMPLES:

```
sage: sin(x).nintegral(x,0,3)
(1.989992496600..., 2.209335488557...e-14, 21, 0)
```

nintegrate()

Compute the numerical integral of self. Please see sage.calculus.calculus.nintegral for more details.

EXAMPLES:

```
sage: sin(x).nintegral(x,0,3)
(1.989992496600..., 2.209335488557...e-14, 21, 0)
```

nops()

Returns the number of arguments of this expression.

EXAMPLES:

```
sage: var('a,b,c,x,y')
(a, b, c, x, y)
sage: a.number_of_operands()
0
sage: (a^2 + b^2 + (x+y)^2).number_of_operands()
3
sage: (a^2).number_of_operands()
2
sage: (a*b^2*c).number_of_operands()
3
```

norm()

The complex norm of this symbolic expression, i.e., the expression times its complex conjugate.

EXAMPLES:

```
sage: a = 1 + 2*I
sage: a.norm()
5
sage: a = sqrt(2) + 3^(1/3)*I; a
sqrt(2) + I*3^(1/3)
sage: a.norm()
3^(2/3) + 2
sage: CDF(a).norm()
4.08008382305
sage: CDF(a.norm())
4.08008382305
```

number_of_arguments()

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = x + y
sage: f.number_of_arguments()
2

sage: g = f.function(x)
sage: g.number_of_arguments()
1

sage: x,y,z = var('x,y,z')
sage: (x+y).number_of_arguments()
2
sage: (x+1).number_of_arguments()
1
sage: (sin(x)+1).number_of_arguments()
1
sage: (sin(z)+x+y).number_of_arguments()
3
sage: (sin(x+y)).number_of_arguments()
2

sage: ( 2^(8/9) - 2^(1/9) )(x-1)
...
ValueError: the number of arguments must be less than or equal to 0
```

number_of_operands()

Returns the number of arguments of this expression.

EXAMPLES:

```
sage: var('a,b,c,x,y')
(a, b, c, x, y)
sage: a.number_of_operands()
0
sage: (a^2 + b^2 + (x+y)^2).number_of_operands()
3
sage: (a^2).number_of_operands()
2
sage: (a*b^2*c).number_of_operands()
3
```

numerator()

Returns the numerator of this symbolic expression. If the expression is not a quotient, then this will return the expression itself.

EXAMPLES:

```
sage: a, x, y = var('a,x,y')
sage: f = x*(x-a)/((x^2 - y)*(x-a)); f
x/(x^2 - y)
sage: f.numerator()
x
sage: f.denominator()
x^2 - y

sage: y = var('y')
sage: g = x + y/(x + 2); g
x + y/(x + 2)
sage: g.numerator()
x + y/(x + 2)
sage: g.denominator()
1
```

numerical_approx()

Return a numerical approximation this symbolic expression as either a real or complex number with at least the requested number of bits or digits of precision.

EXAMPLES:

```
sage: sin(x).subs(x=5).n()
-0.958924274663138
sage: sin(x).subs(x=5).n(100)
-0.95892427466313846889315440616
sage: sin(x).subs(x=5).n(digits=50)
-0.95892427466313846889315440615599397335246154396460
sage: zeta(x).subs(x=2).numerical_approx(digits=50)
1.6449340668482264364724151666460251892189499012068

sage: cos(3).numerical_approx(200)
-0.98999249660044545727157279473126130239367909661558832881409
sage: numerical_approx(cos(3), digits=10)
-0.9899924966
sage: (i + 1).numerical_approx(32)
1.00000000 + 1.00000000*I
sage: (pi + e + sqrt(2)).numerical_approx(100)
7.2740880444219335226246195788
```

TESTS:

We test the evaluation of different infinities available in pynac:

```
sage: t = x - oo; t
-Infinity
sage: t.n()
-infinity
sage: t = x + oo; t
+Infinity
sage: t.n()
+infinity
sage: t = x - unsigned_infinity; t
Infinity
sage: t.n()
+infinity
```

operands()

Returns a list containing the operands of this expression.

EXAMPLES:

```
sage: var('a,b,c,x,y')
(a, b, c, x, y)
sage: (a^2 + b^2 + (x+y)^2).operands()
[(x + y)^2, a^2, b^2]
sage: (a^2).operands()
[a, 2]
sage: (a*b^2*c).operands()
[a, b^2, c]
```

operator()

Returns the topmost operator in this expression.

EXAMPLES:

```
sage: x,y,z = var('x,y,z')
sage: (x+y).operator()
<built-in function add>
sage: (x^y).operator()
<built-in function pow>
sage: (x^y * z).operator()
<built-in function mul>
sage: (x < y).operator()
<built-in function lt>

sage: abs(x).operator()
abs
sage: r = gamma(x).operator(); type(r)
<class 'sage.functions.other.Function_gamma'>

sage: from sage.symbolic.function import function
sage: psi = function('psi', 1)
sage: psi(x).operator()
psi

sage: r = psi(x).operator()
sage: r == psi
True

sage: f = function('f', 1, conjugate_func=lambda x: 2*x)
sage: nf = f(x).operator()
sage: nf(x).conjugate()
2*x
```

```
sage: f = function('f')
sage: a = f(x).diff(x); a
D[0](f)(x)
sage: a.operator()
D[0](f)
```

TESTS: sage: (x <= y).operator() <built-in function le> sage: (x == y).operator() <built-in function eq> sage: (x != y).operator() <built-in function ne> sage: (x > y).operator() <built-in function gt> sage: (x >= y).operator() <built-in function ge>

partial_fraction()

Return the partial fraction expansion of `self` with respect to the given variable.

INPUT:

- `var` - variable name or string (default: first variable)

OUTPUT: Symbolic expression

EXAMPLES:

```
sage: f = x^2/(x+1)^3
sage: f.partial_fraction()
1/(x + 1) - 2/(x + 1)^2 + 1/(x + 1)^3
sage: f.partial_fraction()
1/(x + 1) - 2/(x + 1)^2 + 1/(x + 1)^3
```

Notice that the first variable in the expression is used by default:

```
sage: y = var('y')
sage: f = y^2/(y+1)^3
sage: f.partial_fraction()
1/(y + 1) - 2/(y + 1)^2 + 1/(y + 1)^3

sage: f = y^2/(y+1)^3 + x/(x-1)^3
sage: f.partial_fraction()
y^2/(y^3 + 3*y^2 + 3*y + 1) + 1/(x - 1)^2 + 1/(x - 1)^3
```

You can explicitly specify which variable is used:

```
sage: f.partial_fraction(y)
x/(x^3 - 3*x^2 + 3*x - 1) + 1/(y + 1) - 2/(y + 1)^2 + 1/(y + 1)^3
```

plot()

Plot a symbolic expression. All arguments are passed onto the standard plot command.

EXAMPLES:

This displays a straight line:

```
sage: sin(2).plot((x,0,3))
```

This draws a red oscillatory curve:

```
sage: sin(x^2).plot((x,0,2*pi), rgbcolor=(1,0,0))
```

Another plot using the variable theta:

```
sage: var('theta')
theta
sage: (cos(theta) - erf(theta)).plot((theta,-2*pi,2*pi))
```

A very thick green plot with a frame:

```
sage: sin(x).plot((x,-4*pi, 4*pi), thickness=20, rgbcolor=(0,0.7,0)).show(frame=True)
```

You can embed 2d plots in 3d space as follows:

```
sage: plot(sin(x^2), (x,-pi, pi), thickness=2).plot3d(z = 1)
```

A more complicated family:

```
sage: G = sum([plot(sin(n*x), (x,-2*pi, 2*pi)).plot3d(z=n) for n in [0,0.1,..1]])
sage: G.show(frame_aspect_ratio=[1,1,1/2])
```

A plot involving the floor function:

```
sage: plot(1.0 - x * floor(1/x), (x,0.00001,1.0))
```

Sage used to allow symbolic functions with "no arguments"; this no longer works:

```
sage: plot(2*sin, -4, 4)
...
TypeError: unsupported operand parent(s) for '*': 'Integer Ring' and '<class 'sage.functions
```

You should evaluate the function first:

```
sage: plot(2*sin(x), -4, 4)
```

TESTS:

```
sage: f(x) = x*(1 - x)
sage: plot(f,0,1)
```

poly()

Express this symbolic expression as a polynomial in x. If this is not a polynomial in x, then some coefficients may be functions of x.

Warning: This is different from `polynomial()` which returns a Sage polynomial over a given base ring.

EXAMPLES:

```
sage: var('a, x')
(a, x)
sage: p = expand((x-a*sqrt(2))^2 + x + 1); p
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
sage: p.poly(a)
-2*sqrt(2)*a*x + 2*a^2 + x^2 + x + 1
sage: bool(expand(p.poly(a)) == p)
True
sage: p.poly(x)
-(2*sqrt(2)*a - 1)*x + 2*a^2 + x^2 + 1
```

polynomial()

Return this symbolic expression as an algebraic polynomial over the given base ring, if possible.

The point of this function is that it converts purely symbolic polynomials into optimised algebraic polynomials over a given base ring.

Warning: This is different from meth:*poly* which is used to rewrite self as a polynomial in terms of one of the variables.

INPUT:

- `base_ring` - a ring

EXAMPLES:

```
sage: f = x^2 -2/3*x + 1
sage: f.polynomial(QQ)
x^2 - 2/3*x + 1
```

```
sage: f.polynomial(GF(19))
x^2 + 12*x + 1
```

Polynomials can be useful for getting the coefficients of an expression:

```
sage: g = 6*x^2 - 5
sage: g.coefficients()
[[-5, 0], [6, 2]]
sage: g.polynomial(QQ).list()
[-5, 0, 6]
sage: g.polynomial(QQ).dict()
{0: -5, 2: 6}

sage: f = x^2*e + x + pi/e
sage: f.polynomial(RDF)
2.71828182846*x^2 + 1.0*x + 1.15572734979
sage: g = f.polynomial(RR); g
2.71828182845905*x^2 + 1.00000000000000*x + 1.15572734979092
sage: g.parent()
Univariate Polynomial Ring in x over Real Field with 53 bits of precision
sage: f.polynomial(RealField(100))
2.7182818284590452353602874714*x^2 + 1.0000000000000000000000000000*x + 1.1557273497909217
sage: f.polynomial(CDF)
2.71828182846*x^2 + 1.0*x + 1.15572734979
sage: f.polynomial(CC)
2.71828182845905*x^2 + 1.00000000000000*x + 1.15572734979092
```

We coerce a multivariate polynomial with complex symbolic coefficients:

```
sage: x, y, n = var('x, y, n')
sage: f = pi^3*x - y^2*e - I; f
pi^3*x - y^2*e - I
sage: f.polynomial(CDF)
(-2.71828182846)*y^2 + 31.0062766803*x - 1.0*I
sage: f.polynomial(CC)
(-2.71828182845905)*y^2 + 31.0062766802998*x - 1.00000000000000*I
sage: f.polynomial(ComplexField(70))
(-2.7182818284590452354)*y^2 + 31.006276680299820175*x - 1.0000000000000000000*I
```

Another polynomial:

```
sage: f = sum((e*I)^n*x^n for n in range(5)); f
x^4*e^4 - I*x^3*e^3 - x^2*e^2 + I*x*e + 1
sage: f.polynomial(CDF)
54.5981500331*x^4 - 20.0855369232*I*x^3 - 7.38905609893*x^2 + 2.71828182846*I*x + 1.0
sage: f.polynomial(CC)
54.5981500331442*x^4 - 20.0855369231877*I*x^3 - 7.38905609893065*x^2 + 2.71828182845905*I*x
```

A multivariate polynomial over a finite field:

```
sage: f = (3*x^5 - 5*y^5)^7; f
(3*x^5 - 5*y^5)^7
sage: g = f.polynomial(GF(7)); g
3*x^35 + 2*y^35
sage: parent(g)
Multivariate Polynomial Ring in x, y over Finite Field of size 7
```

power_series()

Return algebraic power series associated to this symbolic expression, which must be a polynomial in one variable, with coefficients coercible to the base ring.

The power series is truncated one more than the degree.

EXAMPLES:

```
sage: theta = var('theta')
sage: f = theta^3 + (1/3)*theta - 17/3
sage: g = f.power_series(QQ); g
-17/3 + 1/3*theta + theta^3 + O(theta^4)
sage: g^3
-4913/27 + 289/9*theta - 17/9*theta^2 + 2602/27*theta^3 + O(theta^4)
sage: g.parent()
Power Series Ring in theta over Rational Field
```

pyobject()

Get the underlying Python object corresponding to this expression, assuming this expression is a single numerical value. Otherwise, a TypeError is raised.

EXAMPLES:

```
sage: var('x')
x
sage: b = -17/3
sage: a = SR(b)
sage: a.pyobject()
-17/3
sage: a.pyobject() is b
True
```

radical_simplify()

Simplifies this symbolic expression, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables

DETAILS: This uses the Maxima radcan() command. From the Maxima documentation: "All functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, produces a regular form. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference can be simplified by radcan to zero. For some expressions radcan is quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial fraction expansions of exponents."

ALIAS: radical_simplify, simplify_radical, simplify_log, log_simplify, exp_simplify, simplify_exp are all the same

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)

sage: f = log(x*y)
sage: f.simplify_radical()
log(x) + log(y)

sage: f = (log(x+x^2)-log(x))^a/log(1+x)^(a/2)
sage: f.simplify_radical()
log(x + 1)^(1/2*a)

sage: f = (e^x-1)/(1+e^(x/2))
sage: f.simplify_exp()
e^(1/2*x) - 1
```

rational_expand()

Expand this symbolic expression. Products of sums and exponentiated sums are multiplied out, numerators of rational expressions which are sums are split into their respective terms, and multiplications are distributed over addition at all levels.

EXAMPLES:

We expand the expression $(x-y)^5$ using both method and functional notation.

```
sage: x,y = var('x,y')
sage: a = (x-y)^5
sage: a.expand()
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
sage: expand(a)
x^5 - 5*x^4*y + 10*x^3*y^2 - 10*x^2*y^3 + 5*x*y^4 - y^5
```

We expand some other expressions:

```
sage: expand((x-1)^3/(y-1))
x^3/(y - 1) - 3*x^2/(y - 1) + 3*x/(y - 1) - 1/(y - 1)
sage: expand((x+sin((x+y)^2))^2)
x^2 + 2*x*sin((x + y)^2) + sin((x + y)^2)^2
```

We can expand individual sides of a relation:

```
sage: a = (16*x-13)^2 == (3*x+5)^2/2
sage: a.expand()
256*x^2 - 416*x + 169 == 9/2*x^2 + 15*x + 25/2
sage: a.expand('left')
256*x^2 - 416*x + 169 == 1/2*(3*x + 5)^2
sage: a.expand('right')
(16*x - 13)^2 == 9/2*x^2 + 15*x + 25/2
```

TESTS:

sage: var('x,y') (x, y) sage: ((x + (2/3)*y)^3).expand() x^3 + 2*x^2*y + 4/3*x*y^2 + 8/27*y^3 sage: expand((x*sin(x) - cos(y)/x)^2) x^2*sin(x)^2 - 2*sin(x)*cos(y) + cos(y)^2/x^2 sage: f = (x-y)*(x+y); f (x - y)*(x + y) sage: f.expand() x^2 - y^2

rational_simplify()

Simplify by expanding repeatedly rational expressions.

ALIAS: rational_simplify and simplify_rational are the same

EXAMPLES:

```
sage: f = sin(x/(x^2 + x))
sage: f
sin(x/(x^2 + x))
sage: f.simplify_rational()
sin(1/(x + 1))

sage: f = ((x - 1)^(3/2) - (x + 1)*sqrt(x - 1))/sqrt((x - 1)*(x + 1)); f
((x - 1)^(3/2) - sqrt(x - 1)*(x + 1))/sqrt((x - 1)*(x + 1))
sage: f.simplify_rational()
-2*sqrt(x - 1)/sqrt(x^2 - 1)
```

real()

Return the real part of this symbolic expression.

EXAMPLES:

```
sage: x = var('x')
sage: x.real_part()
real_part(x)
sage: SR(2+3*I).real_part()
2
sage: SR(CDF(2,3)).real_part()
2.0
sage: SR(CC(2,3)).real_part()
2.00000000000000
```

```
sage: f = log(x)
sage: f.real_part()
log(abs(x))
```

real_part()

Return the real part of this symbolic expression.

EXAMPLES:

```
sage: x = var('x')
sage: x.real_part()
real_part(x)
sage: SR(2+3*I).real_part()
2
sage: SR(CDF(2,3)).real_part()
2.0
sage: SR(CC(2,3)).real_part()
2.00000000000000

sage: f = log(x)
sage: f.real_part()
log(abs(x))
```

rhs()

If self is a relational expression, return the right hand side of the relation. Otherwise, raise a ValueError.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 <= x^2 - 2*x + 3
sage: eqn.right_hand_side()
x^2 - 2*x + 3
sage: eqn.rhs()
x^2 - 2*x + 3
sage: eqn.right()
x^2 - 2*x + 3
```

right()

If self is a relational expression, return the right hand side of the relation. Otherwise, raise a ValueError.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 <= x^2 - 2*x + 3
sage: eqn.right_hand_side()
x^2 - 2*x + 3
sage: eqn.rhs()
x^2 - 2*x + 3
sage: eqn.right()
x^2 - 2*x + 3
```

right_hand_side()

If self is a relational expression, return the right hand side of the relation. Otherwise, raise a ValueError.

EXAMPLES:

```
sage: x = var('x')
sage: eqn = (x-1)^2 <= x^2 - 2*x + 3
sage: eqn.right_hand_side()
x^2 - 2*x + 3
sage: eqn.rhs()
x^2 - 2*x + 3
```

```
sage: eqn.right()
x^2 - 2*x + 3
```

roots()

Returns roots of `self` that can be found exactly, possibly with multiplicities. Not all roots are guaranteed to be found.

Warning: This is *not* a numerical solver - use `find_root` to solve for self == 0 numerically on an interval.

INPUT:

- `x` - variable to view the function in terms of (use default variable if not given)
- `explicit_solutions` - bool (default True); require that roots be explicit rather than implicit
- `multiplicities` - bool (default True); when True, return multiplicities
- `ring` - a ring (default None): if not None, convert self to a polynomial over ring and find roots over ring

OUTPUT:

list of pairs (root, multiplicity) or list of roots

If there are infinitely many roots, e.g., a function like $\sin(x)$, only one is returned.

EXAMPLES:

```
sage: var('x, a')
(x, a)
```

A simple example:

```
sage: ((x^2-1)^2).roots()
[(-1, 2), (1, 2)]
sage: ((x^2-1)^2).roots(multiplicities=False)
[-1, 1]
```

A complicated example.

```
sage: f = expand((x^2 - 1)^3*(x^2 + 1)*(x-a)); f
-a*x^8 + x^9 + 2*a*x^6 - 2*x^7 - 2*a*x^2 + 2*x^3 + a - x
```

The default variable is a, since it is the first in alphabetical order:

```
sage: f.roots()
[(x, 1)]
```

As a polynomial in a, x is indeed a root:

```
sage: f.poly(a)
x^9 - 2*x^7 + 2*x^3 - (x^8 - 2*x^6 + 2*x^2 - 1)*a - x
sage: f(a=x)
0
```

The roots in terms of x are what we expect:

```
sage: f.roots(x)
[(a, 1), (-I, 1), (I, 1), (1, 3), (-1, 3)]
```

Only one root of $\sin(x) = 0$ is given:

```
sage: f = sin(x)
sage: f.roots(x)
[(0, 1)]
```

We derive the roots of a general quadratic polynomial:

```
sage: var('a,b,c,x')
(a, b, c, x)
sage: (a*x^2 + b*x + c).roots(x)
[(-1/2*(b + sqrt(-4*a*c + b^2))/a, 1), (-1/2*(b - sqrt(-4*a*c + b^2))/a, 1)]
```

By default, all the roots are required to be explicit rather than implicit. To get implicit roots, pass `explicit_solutions=False` to `.roots()`

```
sage: var('x')
x
sage: f = x^(1/9) + (2^(8/9) - 2^(1/9))*(x - 1) - x^(8/9)
sage: f.roots()
...
RuntimeError: no explicit roots found
sage: f.roots(explicit_solutions=False)
[((2^(8/9) - 2^(1/9) + x^(8/9) - x^(1/9))/(2^(8/9) - 2^(1/9)), 1)]
```

Another example, but involving a degree 5 poly whose roots don't get computed explicitly:

```
sage: f = x^5 + x^3 + 17*x + 1
sage: f.roots()
...
RuntimeError: no explicit roots found
sage: f.roots(explicit_solutions=False)
[(x^5 + x^3 + 17*x + 1, 1)]
sage: f.roots(explicit_solutions=False, multiplicities=False)
[x^5 + x^3 + 17*x + 1]
```

Now let's find some roots over different rings:

```
sage: f.roots(ring=CC)
[(-0.0588115223184495, 1), (-1.331099917875... - 1.52241655183732*I, 1), (-1.331099917875...
sage: (2.5*f).roots(ring=RR)
[(-0.058811522318449..., 1)]
sage: f.roots(ring=CC, multiplicities=False)
[-0.0588115223184495, -1.331099917875... - 1.52241655183732*I, -1.331099917875... + 1.522416
sage: f.roots(ring=QQ)
[]
sage: f.roots(ring=QQbar, multiplicities=False)
[-0.05881152231844944?, -1.331099917875796? - 1.522416551837318?*I, -1.331099917875796? + 1.
```

Root finding over finite fields:

```
sage: f.roots(ring=GF(7^2, 'a'))
[(3, 1), (4*a + 6, 2), (3*a + 3, 2)]
```

TESTS:

```
sage: (sqrt(3) * f).roots(ring=QQ)
...
TypeError: unable to convert sqrt(3) to a rational
```

series()

Return the power series expansion of self in terms of the variable symbol to the given order.

INPUT:

- `symbol` - a variable
- `order` - an integer

OUTPUT:

- a power series

To truncate the power series and obtain a normal expression, use the truncate command.

EXAMPLES:

We expand a polynomial in x about 0, about 1, and also truncate it back to a polynomial:

```
sage: var('x,y')
(x, y)
sage: f = (x^3 - sin(y)*x^2 - 5*x + 3); f
x^3 - x^2*sin(y) - 5*x + 3
sage: g = f.series(x, 4); g
3 + (-5)*x + (-sin(y))*x^2 + 1*x^3
sage: g.truncate()
x^3 - x^2*sin(y) - 5*x + 3
sage: g = f.series(x==1, 4); g
(-sin(y) - 1) + (-2*sin(y) - 2)*(x - 1) + (-sin(y) + 3)*(x - 1)^2 + 1*(x - 1)^3
sage: h = g.truncate(); h
-(sin(y) - 3)*(x - 1)^2 + (x - 1)^3 - 2*(sin(y) + 1)*(x - 1) - sin(y) - 1
sage: h.expand()
x^3 - x^2*sin(y) - 5*x + 3
```

We computer another series expansion of an analytic function:

```
sage: f = sin(x)/x^2
sage: f.series(x,7)
1*x^(-1) + (-1/6)*x + 1/120*x^3 + (-1/5040)*x^5 + Order(x^7)
sage: f.series(x==1,3)
(sin(1)) + (-2*sin(1) + cos(1))*(x - 1) + (5/2*sin(1) - 2*cos(1))*(x - 1)^2 + Order((x - 1)^
sage: f.series(x==1,3).truncate().expand()
5/2*x^2*sin(1) - 2*x^2*cos(1) - 7*x*sin(1) + 5*x*cos(1) + 11/2*sin(1) - 3*cos(1)
```

Following the GiNaC tutorial, we use John Machin's amazing formula $\pi = 16\tan^{-1}(1/5) - 4\tan^{-1}(1/239)$ to compute digits of π. We expand the arc tangent around 0 and insert the fractions 1/5 and 1/239.

```
sage: x = var('x')
sage: f = atan(x).series(x, 10); f
1*x + (-1/3)*x^3 + 1/5*x^5 + (-1/7)*x^7 + 1/9*x^9 + Order(x^10)
sage: float(16*f.subs(x==1/5) - 4*f.subs(x==1/239))
3.1415926824043994
```

show()

Show this symbolic expression, i.e., typeset it nicely.

EXAMPLES:

```
sage: (x^2 + 1).show()
x^{2}  + 1
```

simplify()

Returns a simplified version of this symbolic expression.

Note: Currently, this does just sends the expression to Maxima and converts it back to Sage.

See Also:

simplify_full(), simplify_trig(), simplify_rational(), simplify_radical()

EXAMPLES:

```
sage: a = var('a'); f = x*sin(2)/(x^a); f
x*sin(2)/x^a
sage: f.simplify()
x^(-a + 1)*sin(2)
```

simplify_exp()

Simplifies this symbolic expression, which can contain logs, exponentials, and radicals, by converting it

into a form which is canonical over a large class of expressions and a given ordering of variables

DETAILS: This uses the Maxima radcan() command. From the Maxima documentation: "All functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, produces a regular form. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference can be simplified by radcan to zero. For some expressions radcan is quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial fraction expansions of exponents."

ALIAS: radical_simplify, simplify_radical, simplify_log, log_simplify, exp_simplify, simplify_exp are all the same

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)

sage: f = log(x*y)
sage: f.simplify_radical()
log(x) + log(y)

sage: f = (log(x+x^2)-log(x))^a/log(1+x)^(a/2)
sage: f.simplify_radical()
log(x + 1)^(1/2*a)

sage: f = (e^x-1)/(1+e^(x/2))
sage: f.simplify_exp()
e^(1/2*x) - 1
```

simplify_full()

Applies simplify_trig, simplify_rational, and simplify_radical to self (in that order).

ALIAS: simplfy_full and full_simplify are the same.

EXAMPLES:

```
sage: a = log(8)/log(2)
sage: a.simplify_full()
3

sage: f = sin(x)^2 + cos(x)^2
sage: f.simplify_full()
1

sage: f = sin(x/(x^2 + x))
sage: f.simplify_full()
sin(1/(x + 1))
```

simplify_log()

Simplifies this symbolic expression, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables

DETAILS: This uses the Maxima radcan() command. From the Maxima documentation: "All functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, produces a regular form. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference can be simplified by radcan to zero. For some expressions radcan is quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial fraction expansions of exponents."

ALIAS: radical_simplify, simplify_radical, simplify_log, log_simplify, exp_simplify, simplify_exp are all the same

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)

sage: f = log(x*y)
sage: f.simplify_radical()
log(x) + log(y)

sage: f = (log(x+x^2)-log(x))^a/log(1+x)^(a/2)
sage: f.simplify_radical()
log(x + 1)^(1/2*a)

sage: f = (e^x-1)/(1+e^(x/2))
sage: f.simplify_exp()
e^(1/2*x) - 1
```

simplify_radical()

Simplifies this symbolic expression, which can contain logs, exponentials, and radicals, by converting it into a form which is canonical over a large class of expressions and a given ordering of variables

DETAILS: This uses the Maxima radcan() command. From the Maxima documentation: "All functionally equivalent forms are mapped into a unique form. For a somewhat larger class of expressions, produces a regular form. Two equivalent expressions in this class do not necessarily have the same appearance, but their difference can be simplified by radcan to zero. For some expressions radcan is quite time consuming. This is the cost of exploring certain relationships among the components of the expression for simplifications based on factoring and partial fraction expansions of exponents."

ALIAS: radical_simplify, simplify_radical, simplify_log, log_simplify, exp_simplify, simplify_exp are all the same

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)

sage: f = log(x*y)
sage: f.simplify_radical()
log(x) + log(y)

sage: f = (log(x+x^2)-log(x))^a/log(1+x)^(a/2)
sage: f.simplify_radical()
log(x + 1)^(1/2*a)

sage: f = (e^x-1)/(1+e^(x/2))
sage: f.simplify_exp()
e^(1/2*x) - 1
```

simplify_rational()

Simplify by expanding repeatedly rational expressions.

ALIAS: rational_simplify and simplify_rational are the same

EXAMPLES:

```
sage: f = sin(x/(x^2 + x))
sage: f
sin(x/(x^2 + x))
sage: f.simplify_rational()
sin(1/(x + 1))

sage: f = ((x - 1)^(3/2) - (x + 1)*sqrt(x - 1))/sqrt((x - 1)*(x + 1)); f
((x - 1)^(3/2) - sqrt(x - 1)*(x + 1))/sqrt((x - 1)*(x + 1))
```

```
sage: f.simplify_rational()
-2*sqrt(x - 1)/sqrt(x^2 - 1)
```

simplify_trig()

First expands using trig_expand, then employs the identities $\sin(x)^2 + \cos(x)^2 = 1$ and $\cosh(x)^2 - \sin(x)^2 = 1$ to simplify expressions containing tan, sec, etc., to sin, cos, sinh, cosh.

ALIAS: trig_simplify and simplify_trig are the same

EXAMPLES:

```
sage: f = sin(x)^2 + cos(x)^2; f
sin(x)^2 + cos(x)^2
sage: f.simplify()
sin(x)^2 + cos(x)^2
sage: f.simplify_trig()
1
```

sin()

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: sin(x^2 + y^2)
sin(x^2 + y^2)
sage: sin(sage.symbolic.constants.pi)
0
sage: sin(SR(1))
sin(1)
sage: sin(SR(RealField(150)(1)))
0.84147098480789650665250232163029899962256306
```

sinh()

Return sinh of self.

We have $sinh(x) = (e^{x} - e^{-x})/2$.

EXAMPLES:

```
sage: x.sinh()
sinh(x)
sage: SR(1).sinh()
sinh(1)
sage: SR(0).sinh()
0
sage: SR(1.0).sinh()
sinh(1.00000000000000)
```

Use .n() to get a numerical approximation:

```
sage: SR(1.0).sinh().n()
1.17520119364380
sage: maxima('sinh(1.0)')
1.175201193643801

sinh(1.0000000000000000000000000)
sage: SR(1).sinh().n(90)
1.17520119364380145688238193
sage: SR(RIF(1)).sinh()
sinh(1)
sage: SR(RIF(1)).sinh().n()
1.175201193643802?
```

solve()

Analytically solve the equation `self == 0` for the variable x.

> **Warning:** This is not a numerical solver - use `find_root` to solve for self == 0 numerically on an interval.

INPUT:

- `x` - variable to solve for
- `multiplicities` - bool (default: False); if True, return corresponding multiplicities.
- **`explicit_solutions` - bool; if True, require that all** solutions returned be explicit (rather than implicit)

EXAMPLES:

```
sage: z = var('z')
sage: (z^5 - 1).solve(z)
[z == e^(2/5*I*pi), z == e^(4/5*I*pi), z == e^(-4/5*I*pi), z == e^(-2/5*I*pi), z == 1]

sage: var('Q')
Q
sage: solve(Q*sqrt(Q^2 + 2) - 1,Q)
[Q == 1/sqrt(-sqrt(2) + 1), Q == 1/sqrt(sqrt(2) + 1)]
```

sqrt()

EXAMPLES: sage: var('x, y') (x, y) sage: SR(2).sqrt() sqrt(2) sage: (x^2+y^2).sqrt() sqrt(x^2 + y^2) sage: (x^2).sqrt() sqrt(x^2)

step()

Return the value of the Heaviside step function, which is 0 for negative x, 1/2 for 0, and 1 for positive x.

EXAMPLES:

```
sage: x = var('x')
sage: SR(1.5).step()
1
sage: SR(0).step()
1/2
sage: SR(-1/2).step()
0
sage: SR(float(-1)).step()
0
```

subs()

EXAMPLES:

```
sage: var('x,y,z,a,b,c,d,e,f')
(x, y, z, a, b, c, d, e, f)
sage: w0 = SR.wild(0); w1 = SR.wild(1)
sage: t = a^2 + b^2 + (x+y)^3

# substitute with keyword arguments (works only with symbols)
sage: t.subs(a=c)
(x + y)^3 + b^2 + c^2

# substitute with a dictionary argument
sage: t.subs({a^2: c})
(x + y)^3 + b^2 + c

sage: t.subs({w0^2: w0^3})
(x + y)^3 + a^3 + b^3
```

```
# substitute with a relational expression
sage: t.subs(w0^2 == w0^3)
(x + y)^3 + a^3 + b^3

sage: t.subs(w0==w0^2)
(x^2 + y^2)^18 + a^16 + b^16

# more than one keyword argument is accepted
sage: t.subs(a=b, b=c)
(x + y)^3 + b^2 + c^2

# using keyword arguments with a dictionary is allowed
sage: t.subs({a:b}, b=c)
(x + y)^3 + b^2 + c^2

# in this case keyword arguments override the dictionary
sage: t.subs({a:b}, a=c)
(x + y)^3 + b^2 + c^2

sage: t.subs({a:b, b:c})
(x + y)^3 + b^2 + c^2
```

TESTS: # no arguments return the same expression sage: t.subs() (x + y)^3 + a^2 + b^2
similarly for an empty dictionary argument sage: t.subs({}) (x + y)^3 + a^2 + b^2
non keyword or dictionary argument returns error sage: t.subs(5) Traceback (most recent call last): ... TypeError: subs takes either a set of keyword arguments, a dictionary, or a symbolic relational expression
substitutions with infinity sage: (x/y).subs(y=oo) 0 sage: (x/y).subs(x=oo) +Infinity sage: (x*y).subs(x=oo) +Infinity sage: (x^y).subs(x=oo) Traceback (most recent call last): ... RuntimeError: power::eval(): pow(Infinity, x) for non numeric x is not defined. sage: (x^y).subs(y=oo) Traceback (most recent call last): ... RuntimeError: power::eval(): pow(x, Infinity) for non numeric x is not defined. sage: (x+y).subs(x=oo) +Infinity sage: (x-y).subs(y=oo) -Infinity sage: gamma(x).subs(x=-1) Infinity sage: 1/gamma(x).subs(x=-1) 0

subs_expr()

Given a dictionary of key:value pairs, substitute all occurrences of key for value in self. The substitutions can also be given as a number of symbolic equalities key == value; see the examples.

> **Warning:** This is a formal pattern substitution, which may or may not have any mathematical meaning. The exact rules used at present in Sage are determined by Maxima's subst command. Sometimes patterns are not replaced even though one would think they should be - see examples below.

EXAMPLES:

```
sage: f = x^2 + 1
sage: f.subs_expr(x^2 == x)
x + 1

sage: var('x,y,z'); f = x^3 + y^2 + z
(x, y, z)
sage: f.subs_expr(x^3 == y^2, z == 1)
2*y^2 + 1
```

Or the same thing giving the substitutions as a dictionary:

```
sage: f.subs_expr({x^3:y^2, z:1})
2*y^2 + 1
```

```
sage: f = x^2 + x^4
sage: f.subs_expr(x^2 == x)
x^4 + x
sage: f = cos(x^2) + sin(x^2)
sage: f.subs_expr(x^2 == x)
sin(x) + cos(x)

sage: f(x,y,t) = cos(x) + sin(y) + x^2 + y^2 + t
sage: f.subs_expr(y^2 == t)
(x, y, t) |--> x^2 + 2*t + sin(y) + cos(x)
```

The following seems really weird, but it *is* what Maple does:

```
sage: f.subs_expr(x^2 + y^2 == t)
(x, y, t) |--> x^2 + y^2 + t + sin(y) + cos(x)
sage: maple.eval('subs(x^2 + y^2 = t, cos(x) + sin(y) + x^2 + y^2 + t)')          # optional
'cos(x)+sin(y)+x^2+y^2+t'
sage: maxima.quit()
sage: maxima.eval('cos(x) + sin(y) + x^2 + y^2 + t, x^2 + y^2 = t')
'sin(y)+y^2+cos(x)+x^2+t'
```

Actually Mathematica does something that makes more sense:

```
sage: mathematica.eval('Cos[x] + Sin[y] + x^2 + y^2 + t /. x^2 + y^2 -> t')       # optional
2 t + Cos[x] + Sin[y]
```

substitute()

EXAMPLES:

```
sage: var('x,y,z,a,b,c,d,e,f')
(x, y, z, a, b, c, d, e, f)
sage: w0 = SR.wild(0); w1 = SR.wild(1)
sage: t = a^2 + b^2 + (x+y)^3

# substitute with keyword arguments (works only with symbols)
sage: t.subs(a=c)
(x + y)^3 + b^2 + c^2

# substitute with a dictionary argument
sage: t.subs({a^2: c})
(x + y)^3 + b^2 + c

sage: t.subs({w0^2: w0^3})
(x + y)^3 + a^3 + b^3

# substitute with a relational expression
sage: t.subs(w0^2 == w0^3)
(x + y)^3 + a^3 + b^3

sage: t.subs(w0==w0^2)
(x^2 + y^2)^18 + a^16 + b^16

# more than one keyword argument is accepted
sage: t.subs(a=b, b=c)
(x + y)^3 + b^2 + c^2

# using keyword arguments with a dictionary is allowed
sage: t.subs({a:b}, b=c)
(x + y)^3 + b^2 + c^2
```

```
# in this case keyword arguments override the dictionary
sage: t.subs({a:b}, a=c)
(x + y)^3 + b^2 + c^2

sage: t.subs({a:b, b:c})
(x + y)^3 + b^2 + c^2
```

TESTS: # no arguments return the same expression sage: t.subs() (x + y)^3 + a^2 + b^2
similarly for an empty dictionary argument sage: t.subs({}) (x + y)^3 + a^2 + b^2
non keyword or dictionary argument returns error sage: t.subs(5) Traceback (most recent call last): ... TypeError: subs takes either a set of keyword arguments, a dictionary, or a symbolic relational expression
substitutions with infinity sage: (x/y).subs(y=oo) 0 sage: (x/y).subs(x=oo) +Infinity sage: (x*y).subs(x=oo) +Infinity sage: (x^y).subs(x=oo) Traceback (most recent call last): ... RuntimeError: power::eval(): pow(Infinity, x) for non numeric x is not defined. sage: (x^y).subs(y=oo) Traceback (most recent call last): ... RuntimeError: power::eval(): pow(x, Infinity) for non numeric x is not defined. sage: (x+y).subs(x=oo) +Infinity sage: (x-y).subs(y=oo) -Infinity sage: gamma(x).subs(x=-1) Infinity sage: 1/gamma(x).subs(x=-1) 0

substitute_expression()

Given a dictionary of key:value pairs, substitute all occurrences of key for value in self. The substitutions can also be given as a number of symbolic equalities key == value; see the examples.

> **Warning:** This is a formal pattern substitution, which may or may not have any mathematical meaning. The exact rules used at present in Sage are determined by Maxima's subst command. Sometimes patterns are not replaced even though one would think they should be - see examples below.

EXAMPLES:

```
sage: f = x^2 + 1
sage: f.subs_expr(x^2 == x)
x + 1

sage: var('x,y,z'); f = x^3 + y^2 + z
(x, y, z)
sage: f.subs_expr(x^3 == y^2, z == 1)
2*y^2 + 1
```

Or the same thing giving the substitutions as a dictionary:

```
sage: f.subs_expr({x^3:y^2, z:1})
2*y^2 + 1

sage: f = x^2 + x^4
sage: f.subs_expr(x^2 == x)
x^4 + x
sage: f = cos(x^2) + sin(x^2)
sage: f.subs_expr(x^2 == x)
sin(x) + cos(x)

sage: f(x,y,t) = cos(x) + sin(y) + x^2 + y^2 + t
sage: f.subs_expr(y^2 == t)
(x, y, t) |--> x^2 + 2*t + sin(y) + cos(x)
```

The following seems really weird, but it *is* what Maple does:

```
sage: f.subs_expr(x^2 + y^2 == t)
(x, y, t) |--> x^2 + y^2 + t + sin(y) + cos(x)
sage: maple.eval('subs(x^2 + y^2 = t, cos(x) + sin(y) + x^2 + y^2 + t)')          # optional
```

```
'cos(x)+sin(y)+x^2+y^2+t'
sage: maxima.quit()
sage: maxima.eval('cos(x) + sin(y) + x^2 + y^2 + t, x^2 + y^2 = t')
'sin(y)+y^2+cos(x)+x^2+t'
```

Actually Mathematica does something that makes more sense:

```
sage: mathematica.eval('Cos[x] + Sin[y] + x^2 + y^2 + t /. x^2 + y^2 -> t')       # optional
2 t + Cos[x] + Sin[y]
```

substitute_function()

Returns this symbolic expressions all occurrences of the function *original* replaced with the function *new*.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: clear_functions()
sage: foo = function('foo'); bar = function('bar')
sage: f = foo(x) + 1/foo(pi*y)
sage: f.substitute_function(foo, bar)
1/bar(pi*y) + bar(x)
```

subtract_from_both_sides()

Returns a relation obtained by subtracting *x* from both sides of this relation.

EXAMPLES:

```
sage: eqn = x*sin(x)*sqrt(3) + sqrt(2) > cos(sin(x))
sage: eqn.subtract_from_both_sides(sqrt(2))
sqrt(3)*x*sin(x) > -sqrt(2) + cos(sin(x))
sage: eqn.subtract_from_both_sides(cos(sin(x)))
sqrt(3)*x*sin(x) + sqrt(2) - cos(sin(x)) > 0
```

tan()

EXAMPLES:

```
sage: var('x, y')
(x, y)
sage: tan(x^2 + y^2)
tan(x^2 + y^2)
sage: tan(sage.symbolic.constants.pi/2)
...
ValueError: simple pole at 1/2*pi
sage: tan(SR(1))
tan(1)
sage: tan(SR(RealField(150)(1)))
1.5574077246549022305069748074583601730872508
```

tanh()

Return tanh of self.

We have $ anh(x) = sinh(x) / cosh(x)$.

EXAMPLES:

```
sage: x.tanh()
tanh(x)
sage: SR(1).tanh()
tanh(1)
sage: SR(0).tanh()
0
sage: SR(1.0).tanh()
tanh(1.00000000000000)
```

Use .n() to get a numerical approximation:

```
sage: SR(1.0).tanh().n()
0.761594155955765
sage: maxima('tanh(1.0)')
.7615941559557649
sage: plot(lambda x: SR(x).tanh(), -1, 1)
```

taylor()

Expands this symbolic expression in a truncated Taylor or Laurent series in the variable v around the point a, containing terms through $(x-a)^n$.

INPUT:

- v - variable
- a - number
- n - integer

EXAMPLES:

```
sage: var('a, x, z')
(a, x, z)
sage: taylor(a*log(z), z, 2, 3)
1/24*(z - 2)^3*a - 1/8*(z - 2)^2*a + 1/2*(z - 2)*a + a*log(2)
sage: taylor(sqrt (sin(x) + a*x + 1), x, 0, 3)
1/48*(3*a^3 + 9*a^2 + 9*a - 1)*x^3 - 1/8*(a^2 + 2*a + 1)*x^2 + 1/2*(a + 1)*x + 1
sage: taylor (sqrt (x + 1), x, 0, 5)
7/256*x^5 - 5/128*x^4 + 1/16*x^3 - 1/8*x^2 + 1/2*x + 1
sage: taylor (1/log (x + 1), x, 0, 3)
-19/720*x^3 + 1/24*x^2 - 1/12*x + 1/x + 1/2
sage: taylor (cos(x) - sec(x), x, 0, 5)
-1/6*x^4 - x^2
sage: taylor ((cos(x) - sec(x))^3, x, 0, 9)
-1/2*x^8 - x^6
sage: taylor (1/(cos(x) - sec(x))^3, x, 0, 5)
-15377/7983360*x^4 - 6767/604800*x^2 + 11/120/x^2 + 1/2/x^4 - 1/x^6 - 347/15120
```

test_relation()

Test this relation at several random values, attempting to find a contradiction. If this relation has no variables, it will also test this relation after casting into the domain.

Because the interval fields never return false positives, we can be assured that if True or False is returned (and proof is False) then the answer is correct.

INPUT:

```
ntests -- (default 20) the number of iterations to run
domain -- (optional) the domain from which to draw the random values
          defaults to CIF for equality testing and RIF for
          order testing
proof --  (default True) if False and the domain is an interval field,
          regard overlapping (potentially equal) intervals as equal,
          and return True if all tests succeeded.
```

OUTPUT:

```
True - this relation holds in the domain and has no variables
False - a contradiction was found
NotImplemented - no contradiction found
```

EXAMPLES:

```
sage: (3 < pi).test_relation()
True
```

```
sage: (0 >= pi).test_relation()
False
sage: (exp(pi) - pi).n()
19.9990999791895
sage: (exp(pi) - pi == 20).test_relation()
False
sage: (sin(x)^2 + cos(x)^2 == 1).test_relation()
NotImplemented
sage: (sin(x)^2 + cos(x)^2 == 1).test_relation(proof=False)
True
sage: (x == 1).test_relation()
False
sage: var('x,y')
(x, y)
sage: (x < y).test_relation()
False
```

TESTS:

```
sage: all_relations = [op for name, op in sorted(operator.__dict__.items()) if len(name) ==
sage: all_relations
[<built-in function eq>, <built-in function ge>, <built-in function gt>, <built-in function
sage: [op(3, pi).test_relation() for op in all_relations]
[False, False, False, True, True, True]
sage: [op(pi, pi).test_relation() for op in all_relations]
[True, True, False, True, False, False]

sage: s = 'some_very_long_variable_name_which_will_definitely_collide_if_we_use_a_reasonable
sage: t1, t2 = var(','.join([s+'1',s+'2']))
sage: (t1 == t2).test_relation()
False
```

trailing_coeff()

Return the trailing coefficient of s in self, i.e., the coefficient of the smallest power of s in self.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
sage: f.trailing_coefficient(x)
2*sin(x*y)
sage: f.trailing_coefficient(y)
x
sage: f.trailing_coefficient(sin(x*y))
a*x + x*y + x/y + 100
```

trailing_coefficient()

Return the trailing coefficient of s in self, i.e., the coefficient of the smallest power of s in self.

EXAMPLES:

```
sage: var('x,y,a')
(x, y, a)
sage: f = 100 + a*x + x^3*sin(x*y) + x*y + x/y + 2*sin(x*y)/x; f
x^3*sin(x*y) + a*x + x*y + x/y + 2*sin(x*y)/x + 100
sage: f.trailing_coefficient(x)
2*sin(x*y)
sage: f.trailing_coefficient(y)
x
```

```
sage: f.trailing_coefficient(sin(x*y))
a*x + x*y + x/y + 100
```

trig_expand()

Expands trigonometric and hyperbolic functions of sums of angles and of multiple angles occurring in self. For best results, self should already be expanded.

INPUT:

- full - (default: False) To enhance user control of simplification, this function expands only one level at a time by default, expanding sums of angles or multiple angles. To obtain full expansion into sines and cosines immediately, set the optional parameter full to True.
- half_angles - (default: False) If True, causes half-angles to be simplified away.
- plus - (default: True) Controls the sum rule; expansion of sums (e.g. 'sin(x + y)') will take place only if plus is True.
- times - (default: True) Controls the product rule, expansion of products (e.g. sin(2*x)) will take place only if times is True.

OUTPUT: a symbolic expression

EXAMPLES:

```
sage: sin(5*x).expand_trig()
sin(x)^5 - 10*sin(x)^3*cos(x)^2 + 5*sin(x)*cos(x)^4
sage: cos(2*x + var('y')).expand_trig()
-sin(2*x)*sin(y) + cos(2*x)*cos(y)
```

We illustrate various options to this function:

```
sage: f = sin(sin(3*cos(2*x))*x)
sage: f.expand_trig()
sin(-(sin(cos(2*x))^3 - 3*sin(cos(2*x))*cos(cos(2*x))^2)*x)
sage: f.expand_trig(full=True)
sin(((sin(sin(x)^2)*cos(cos(x)^2) - sin(cos(x)^2)*cos(sin(x)^2))^3 - 3*(sin(sin(x)^2)*cos(co
sage: sin(2*x).expand_trig(times=False)
sin(2*x)
sage: sin(2*x).expand_trig(times=True)
2*sin(x)*cos(x)
sage: sin(2 + x).expand_trig(plus=False)
sin(x + 2)
sage: sin(2 + x).expand_trig(plus=True)
sin(2)*cos(x) + sin(x)*cos(2)
sage: sin(x/2).expand_trig(half_angles=False)
sin(1/2*x)
sage: sin(x/2).expand_trig(half_angles=True)
1/2*sqrt(-cos(x) + 1)*sqrt(2)
```

ALIASES:

trig_expand() and expand_trig() are the same

trig_simplify()

First expands using trig_expand, then employs the identities $\sin(x)^2 + \cos(x)^2 = 1$ and $\cosh(x)^2 - \sin(x)^2 = 1$ to simplify expressions containing tan, sec, etc., to sin, cos, sinh, cosh.

ALIAS: trig_simplify and simplify_trig are the same

EXAMPLES:

```
sage: f = sin(x)^2 + cos(x)^2; f
sin(x)^2 + cos(x)^2
sage: f.simplify()
sin(x)^2 + cos(x)^2
```

```
sage: f.simplify_trig()
1
```

truncate()

Given a power series or expression, return the corresponding expression without the big oh.

INPUT:

- a series as output by the series command

OUTPUT:

- expression

EXAMPLES:

```
sage: f = sin(x)/x^2
sage: f.truncate()
sin(x)/x^2
sage: f.series(x,7)
1*x^(-1) + (-1/6)*x + 1/120*x^3 + (-1/5040)*x^5 + Order(x^7)
sage: f.series(x,7).truncate()
-1/5040*x^5 + 1/120*x^3 - 1/6*x + 1/x
sage: f.series(x==1,3).truncate().expand()
5/2*x^2*sin(1) - 2*x^2*cos(1) - 7*x*sin(1) + 5*x*cos(1) + 11/2*sin(1) - 3*cos(1)
```

variables()

Return sorted list of variables that occur in this expression.

EXAMPLES:

```
sage: (x,y,z) = var('x,y,z')
sage: (x+y).variables()
(x, y)
sage: (2*x).variables()
(x,)
sage: (x^y).variables()
(x, y)
sage: sin(x+y^z).variables()
(x, y, z)
```

zeta()

EXAMPLES:

```
sage: x, y = var('x, y')
sage: (x/y).zeta()
zeta(x/y)
sage: SR(2).zeta()
1/6*pi^2
sage: SR(3).zeta()
zeta(3)
sage: SR(CDF(0,1)).zeta()
zeta(1.0*I)
sage: SR(CDF(0,1)).zeta().n()
0.00330022368532 - 0.418155449141*I
sage: CDF(0,1).zeta()
0.00330022368532 - 0.418155449141*I
sage: plot(lambda x: SR(x).zeta(), -10,10).show(ymin=-3,ymax=3)
```

TESTS:

```
sage: t = SR(1).zeta(); t
zeta(1)
```

```
sage: t.n()
+infinity
```

class **ExpressionIterator**()

next()
x.next() -> the next value, or raise StopIteration

is_Expression()
Retruns True if *x* is a symbolic Expression.

EXAMPLES:

```
sage: from sage.symbolic.expression import is_Expression
sage: is_Expression(x)
True
sage: is_Expression(2)
False
sage: is_Expression(SR(2))
True
```

is_SymbolicEquation()
Returns True if *x* is a symbolic equation.

EXAMPLES:

The following two examples are symbolic equations:

```
sage: from sage.symbolic.expression import is_SymbolicEquation
sage: is_SymbolicEquation(sin(x) == x)
True
sage: is_SymbolicEquation(sin(x) < x)
True
sage: is_SymbolicEquation(x)
False
```

This is not, since `2==3` evaluates to the boolean `False`:

```
sage: is_SymbolicEquation(2 == 3)
False
```

However here since both 2 and 3 are coerced to be symbolic, we obtain a symbolic equation:

```
sage: is_SymbolicEquation(SR(2) == SR(3))
True
```

3.3 Symbolic Equations and Inequalities.

Sage can solve symbolic equations and express inequalities. For example, we derive the quadratic formula as follows:

```
sage: a,b,c = var('a,b,c')
sage: qe = (a*x^2 + b*x + c == 0)
sage: qe
a*x^2 + b*x + c == 0
sage: print solve(qe, x)
[
x == -1/2*(b + sqrt(-4*a*c + b^2))/a,
```

```
x == -1/2*(b - sqrt(-4*a*c + b^2))/a
]
```

3.3.1 The operator, left hand side, and right hand side

Operators:

```
sage: eqn = x^3 + 2/3 >= x - pi
sage: eqn.operator()
<built-in function ge>
sage: (x^3 + 2/3 < x - pi).operator()
<built-in function lt>
sage: (x^3 + 2/3 == x - pi).operator()
<built-in function eq>
```

Left hand side:

```
sage: eqn = x^3 + 2/3 >= x - pi
sage: eqn.lhs()
x^3 + 2/3
sage: eqn.left()
x^3 + 2/3
sage: eqn.left_hand_side()
x^3 + 2/3
```

Right hand side:

```
sage: (x + sqrt(2) >= sqrt(3) + 5/2).right()
sqrt(3) + 5/2
sage: (x + sqrt(2) >= sqrt(3) + 5/2).rhs()
sqrt(3) + 5/2
sage: (x + sqrt(2) >= sqrt(3) + 5/2).right_hand_side()
sqrt(3) + 5/2
```

3.3.2 Arithmetic

Add two symbolic equations:

```
sage: var('a,b')
(a, b)
sage: m = 144 == -10 * a + b
sage: n = 136 == 10 * a + b
sage: m + n
280 == 2*b
sage: int(-144) + m
0 == -10*a + b - 144
```

Subtract two symbolic equations:

```
sage: var('a,b')
(a, b)
sage: m = 144 == 20 * a + b
sage: n = 136 == 10 * a + b
```

```
sage: m - n
8 == 10*a
sage: int(144) - m
0 == -20*a - b + 144
```

Multiply two symbolic equations:

```
sage: x = var('x')
sage: m = x == 5*x + 1
sage: n = sin(x) == sin(x+2*pi)
sage: m * n
x*sin(x) == (5*x + 1)*sin(2*pi + x)
sage: m = 2*x == 3*x^2 - 5
sage: int(-1) * m
-2*x == -3*x^2 + 5
```

Divide two symbolic equations:

```
sage: x = var('x')
sage: m = x == 5*x + 1
sage: n = sin(x) == sin(x+2*pi)
sage: m/n
x/sin(x) == (5*x + 1)/sin(2*pi + x)
sage: m = x != 5*x + 1
sage: n = sin(x) != sin(x+2*pi)
sage: m/n
x/sin(x) != (5*x + 1)/sin(2*pi + x)
```

3.3.3 Substitution

Substitution into relations:

```
sage: x, a = var('x, a')
sage: eq = (x^3 + a == sin(x/a)); eq
x^3 + a == sin(x/a)
sage: eq.substitute(x=5*x)
125*x^3 + a == sin(5*x/a)
sage: eq.substitute(a=1)
x^3 + 1 == sin(x)
sage: eq.substitute(a=x)
x^3 + x == sin(1)
sage: eq.substitute(a=x, x=1)
x + 1 == sin(1/x)
sage: eq.substitute({a:x, x:1})
x + 1 == sin(1/x)
```

3.3.4 Solving

We can solve equations:

```
sage: x = var('x')
sage: S = solve(x^3 - 1 == 0, x)
sage: S
```

```
[x == 1/2*I*sqrt(3) - 1/2, x == -1/2*I*sqrt(3) - 1/2, x == 1]
sage: S[0]
x == 1/2*I*sqrt(3) - 1/2
sage: S[0].right()
1/2*I*sqrt(3) - 1/2
sage: S = solve(x^3 - 1 == 0, x, solution_dict=True)
sage: S
[{x: 1/2*I*sqrt(3) - 1/2}, {x: -1/2*I*sqrt(3) - 1/2}, {x: 1}]
sage: z = 5
sage: solve(z^2 == sqrt(3),z)
...
TypeError: 5 is not a valid variable.
```

We illustrate finding multiplicities of solutions:

```
sage: f = (x-1)^5*(x^2+1)
sage: solve(f == 0, x)
[x == -I, x == I, x == 1]
sage: solve(f == 0, x, multiplicities=True)
([x == -I, x == I, x == 1], [1, 1, 5])
```

We can numerically find roots of equations:

```
sage: (x == sin(x)).find_root(-2,2)
0.0
sage: (x^5 + 3*x + 2 == 0).find_root(-2,2,x)
-0.63283452024215225
sage: (cos(x) == sin(x)).find_root(10,20)
19.634954084936208
```

We illustrate some valid error conditions:

```
sage: (cos(x) != sin(x)).find_root(10,20)
...
ValueError: Symbolic equation must be an equality.
sage: (SR(3)==SR(2)).find_root(-1,1)
...
RuntimeError: no zero in the interval, since constant expression is not 0.
```

There must be at most one variable:

```
sage: x, y = var('x,y')
sage: (x == y).find_root(-2,2)
...
NotImplementedError: root finding currently only implemented in 1 dimension.
```

3.3.5 Assumptions

Forgetting assumptions:

```
sage: var('x,y')
(x, y)
sage: forget() #Clear assumptions
sage: assume(x>0, y < 2)
```

```
sage: assumptions()
[x > 0, y < 2]
sage: (y < 2).forget()
sage: assumptions()
[x > 0]
```

3.3.6 Miscellaneous

Conversion to Maxima:

```
sage: x = var('x')
sage: eq = (x^(3/5) >= pi^2 + e^i)
sage: eq._maxima_init_()
'(x)^(3/5) >= ((%pi)^(2))+(exp(0+%i*1))'
sage: e1 = x^3 + x == sin(2*x)
sage: z = e1._maxima_()
sage: z.parent() is sage.calculus.calculus.maxima
True
sage: z = e1._maxima_(maxima)
sage: z.parent() is maxima
True
sage: z = maxima(e1)
sage: z.parent() is maxima
True
```

Conversion to Maple:

```
sage: x = var('x')
sage: eq = (x == 2)
sage: eq._maple_init_()
'x = 2'
```

Comparison:

```
sage: x = var('x')
sage: (x>0) == (x>0)
True
sage: (x>0) == (x>1)
False
sage: (x>0) != (x>1)
True
```

Variables appearing in the relation:

```
sage: var('x,y,z,w')
(x, y, z, w)
sage: f =  (x+y+w) == (x^2 - y^2 - z^3);   f
w + x + y == x^2 - y^2 - z^3
sage: f.variables()
(w, x, y, z)
```

LaTeX output:

```
sage: latex(x^(3/5) >= pi)
x^{\frac{3}{5}}  \geq  \pi
```

3.3.7 More Examples

```
sage: x,y,a = var('x,y,a')
sage: f = x^2 + y^2 == 1
sage: f.solve(x)
[x == -sqrt(-y^2 + 1), x == sqrt(-y^2 + 1)]
```

```
sage: f = x^5 + a
sage: solve(f==0,x)
[x == (-a)^(1/5)*e^(2/5*I*pi), x == (-a)^(1/5)*e^(4/5*I*pi), x == (-a)^(1/5)*e^(-4/5*I*pi), x == (-a
```

You can also do arithmetic with inequalities, as illustrated below:

```
sage: var('x y')
(x, y)
sage: f = x + 3 == y - 2
sage: f
x + 3 == y - 2
sage: g = f - 3; g
x == y - 5
sage: h =  x^3 + sqrt(2) == x*y*sin(x)
sage: h
sqrt(2) + x^3 == x*y*sin(x)
sage: h - sqrt(2)
x^3 == x*y*sin(x) - sqrt(2)
sage: h + f
x + sqrt(2) + x^3 + 3 == x*y*sin(x) + y - 2
sage: f = x + 3 < y - 2
sage: g = 2 < x+10
sage: f - g
x + 1 < -x + y - 12
sage: f + g
x + 5 < x + y + 8
sage: f*(-1)
-x - 3 < -y + 2
```

TESTS:

We test serializing symbolic equations:

```
sage: eqn = x^3 + 2/3 >= x
sage: loads(dumps(eqn))
x^3 + 2/3 >= x
sage: loads(dumps(eqn)) == eqn
True
```

AUTHORS:

- Bobby Moretti: initial version (based on a trick that Robert Bradshaw suggested).
- William Stein: second version
- William Stein (2007-07-16): added arithmetic with symbolic equations

solve (*f, *args, **kwds*)
Algebraically solve an equation of system of equations for given variables.

INPUT:

- `f` - equation or system of equations (given by a list or tuple)
- `*args` - variables to solve for.
- `solution_dict = True` - return a list of dictionaries containing the solutions.

EXAMPLES:

```
sage: x, y = var('x, y')
sage: solve([x+y==6, x-y==4], x, y)
[[x == 5, y == 1]]
sage: solve([x^2+y^2 == 1, y^2 == x^3 + x + 1], x, y)
[[x == -1/2*I*sqrt(3) - 1/2, y == -1/2*sqrt(-I*sqrt(3) + 3)*sqrt(2)],
 [x == -1/2*I*sqrt(3) - 1/2, y == 1/2*sqrt(-I*sqrt(3) + 3)*sqrt(2)],
 [x == 1/2*I*sqrt(3) - 1/2, y == -1/2*sqrt(I*sqrt(3) + 3)*sqrt(2)],
 [x == 1/2*I*sqrt(3) - 1/2, y == 1/2*sqrt(I*sqrt(3) + 3)*sqrt(2)],
 [x == 0, y == -1],
 [x == 0, y == 1]]
sage: solutions=solve([x^2+y^2 == 1, y^2 == x^3 + x + 1], x, y, solution_dict=True)
sage: for solution in solutions: print solution[x].n(digits=3), ",", solution[y].n(digits=3)
-0.500 - 0.866*I , -1.27 + 0.341*I
-0.500 - 0.866*I , 1.27 - 0.341*I
-0.500 + 0.866*I , -1.27 - 0.341*I
-0.500 + 0.866*I , 1.27 + 0.341*I
0.000 , -1.00
0.000 , 1.00
sage: z = 5
sage: solve([8*z + y == 3, -z +7*y == 0],y,z)
...
TypeError: 5 is not a valid variable.
```

If `True` appears in the list of equations it is ignored, and if `False` appears in the list then no solutions are returned. E.g., note that the first `3==3` evaluates to `True`, not to a symbolic equation.

```
sage: solve([3==3, 1.00000000000000*x^3 == 0], x)
[x == 0]
sage: solve([1.00000000000000*x^3 == 0], x)
[x == 0]
```

Here, the first equation evaluates to `False`, so there are no solutions:

```
sage: solve([1==3, 1.00000000000000*x^3 == 0], x)
[]


sage: var('s,j,b,m,g')
(s, j, b, m, g)
sage: sys = [ m*(1-s) - b*s*j, b*s*j-g*j ];
sage: solve(sys,s,j)
[[s == 1, j == 0], [s == g/b, j == (b - g)*m/(b*g)]]
sage: solve(sys,(s,j))
[[s == 1, j == 0], [s == g/b, j == (b - g)*m/(b*g)]]
sage: solve(sys,[s,j])
[[s == 1, j == 0], [s == g/b, j == (b - g)*m/(b*g)]]
```

solve_mod(*eqns, modulus, solution_dict=False*)

Return all solutions to an equation or list of equations modulo the given integer modulus. Each equation must involve only polynomials in 1 or many variables.

By default the solutions are returned as n-tuples, where n is the number of variables appearing anywhere in the given equations. The variables are in alphabetical order.

INPUT:

- `eqns` - equation or list of equations
- `modulus` - an integer
- `solution_dict` - (default: False) if True, return a list of dictionaries containing the solutions.

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: solve_mod([x^2 + 2 == x, x^2 + y == y^2], 14)
[(4, 2), (4, 6), (4, 9), (4, 13)]
sage: solve_mod([x^2 == 1, 4*x  == 11], 15)
[(14,)]
```

Fermat's equation modulo 3 with exponent 5:

```
sage: var('x,y,z')
(x, y, z)
sage: solve_mod([x^5 + y^5 == z^5], 3)
[(0, 0, 0), (0, 1, 1), (0, 2, 2), (1, 0, 1), (1, 1, 2), (1, 2, 0), (2, 0, 2), (2, 1, 0), (2, 2,
```

We can solve with respect to a bigger modulus if it consists only of small prime factors:

```
sage: [d] = solve_mod([5*x + y == 3, 2*x - 3*y == 9], 3*5*7*11*19*23*29, solution_dict = True)
sage: d[x]
12915279
sage: d[y]
8610183
```

We solve an simple equation modulo 2:

```
sage: x,y = var('x,y')
sage: solve_mod([x == y], 2)
[(0, 0), (1, 1)]
```

Warning: The current implementation splits the modulus into prime powers, then naively enumerates all possible solutions and finally combines the solution using the Chinese Remainder Theorem. The interface is good, but the algorithm is horrible if the modulus has some larger prime factors! Sage {does} have the ability to do something much faster in certain cases at least by using Groebner basis, linear algebra techniques, etc. But for a lot of toy problems this function as is might be useful. At least it establishes an interface.

solve_mod_enumerate(*eqns, modulus*)

Return all solutions to an equation or list of equations modulo the given integer modulus. Each equation must involve only polynomials in 1 or many variables.

The solutions are returned as n-tuples, where n is the number of variables appearing anywhere in the given equations. The variables are in alphabetical order.

INPUT:

- `eqns` - equation or list of equations
- `modulus` - an integer

EXAMPLES:

```
sage: var('x,y')
(x, y)
sage: solve_mod([x^2 + 2 == x, x^2 + y == y^2], 14)
[(4, 2), (4, 6), (4, 9), (4, 13)]
sage: solve_mod([x^2 == 1, 4*x  == 11], 15)
[(14,)]
```

Fermat's equation modulo 3 with exponent 5:

```
sage: var('x,y,z')
(x, y, z)
sage: solve_mod([x^5 + y^5 == z^5], 3)
[(0, 0, 0), (0, 1, 1), (0, 2, 2), (1, 0, 1), (1, 1, 2), (1, 2, 0), (2, 0, 2), (2, 1, 0), (2, 2,
```

We solve an simple equation modulo 2:

```
sage: x,y = var('x,y')
sage: solve_mod([x == y], 2)
[(0, 0), (1, 1)]
```

Warning: Currently this naively enumerates all possible solutions. The interface is good, but the algorithm is horrible if the modulus is at all large! Sage *does* have the ability to do something much faster in certain cases at least by using the Chinese Remainder Theorem, Groebner basis, linear algebra techniques, etc. But for a lot of toy problems this function as is might be useful. At the very least, it establishes an interface.

string_to_list_of_solutions(*s*)

Used internally by the symbolic solve command to convert the output of Maxima's solve command to a list of solutions in Sage's symbolic package.

EXAMPLES:

We derive the (monic) quadratic formula:

```
sage: var('x,a,b')
(x, a, b)
sage: solve(x^2 + a*x + b == 0, x)
[x == -1/2*a - 1/2*sqrt(a^2 - 4*b), x == -1/2*a + 1/2*sqrt(a^2 - 4*b)]
```

Behind the scenes when the above is evaluated the function `string_to_list_of_solutions()` is called with input the string *s* below:

```
sage: s = '[x=-(sqrt(a^2-4*b)+a)/2,x=(sqrt(a^2-4*b)-a)/2]'
sage: sage.symbolic.relation.string_to_list_of_solutions(s)
 [x == -1/2*a - 1/2*sqrt(a^2 - 4*b), x == -1/2*a + 1/2*sqrt(a^2 - 4*b)]
```

test_relation_maxima(*relation*)

Return True if this (in)equality is definitely true. Return False if it is false or the algorithm for testing (in)equality is inconclusive.

EXAMPLES:

```
sage: from sage.symbolic.relation import test_relation_maxima
sage: k = var('k')
sage: pol = 1/(k-1) - 1/k -1/k/(k-1);
sage: test_relation_maxima(pol == 0)
True
sage: f = sin(x)^2 + cos(x)^2 - 1
sage: test_relation_maxima(f == 0)
```

```
True
sage: test_relation_maxima( x == x )
True
sage: test_relation_maxima( x != x )
False
sage: test_relation_maxima( x > x )
False
sage: test_relation_maxima( x^2 > x )
False
sage: test_relation_maxima( x + 2 > x )
True
sage: test_relation_maxima( x - 2 > x )
False
```

3.4 Symbolic Computation.

AUTHORS:

- Bobby Moretti and William Stein (2006-2007)

The Sage calculus module is loosely based on the Sage Enhancement Proposal found at: http://www.sagemath.org:9001/CalculusSEP.

EXAMPLES:

The basic units of the calculus package are symbolic expressions which are elements of the symbolic expression ring (SR). To create a symbolic variable object in Sage, use the `var()` function, whose argument is the text of that variable. Note that Sage is intelligent about LaTeXing variable names.

```
sage: x1 = var('x1'); x1
x1
sage: latex(x1)
x_{1}
sage: theta = var('theta'); theta
theta
sage: latex(theta)
\theta
```

Sage predefines `x` to be a global indeterminate. Thus the following works:

```
sage: x^2
x^2
sage: type(x)
<type 'sage.symbolic.expression.Expression'>
```

More complicated expressions in Sage can be built up using ordinary arithmetic. The following are valid, and follow the rules of Python arithmetic: (The '=' operator represents assignment, and not equality)

```
sage: var('x,y,z')
(x, y, z)
sage: f = x + y + z/(2*sin(y*z/55))
sage: g = f^f; g
(x + y + 1/2*z/sin(1/55*y*z))^(x + y + 1/2*z/sin(1/55*y*z))
```

Differentiation and integration are available, but behind the scenes through maxima:

```
sage: f = sin(x)/cos(2*y)
sage: f.derivative(y)
2*sin(x)*sin(2*y)/cos(2*y)^2
sage: g = f.integral(x); g
-cos(x)/cos(2*y)
```

Note that these methods require an explicit variable name. If none is given, Sage will try to find one for you.

```
sage: f = sin(x); f.derivative()
cos(x)
```

However when this is ambiguous, Sage will raise an exception:

```
sage: f = sin(x+y); f.derivative()
...
ValueError: No differentiation variable specified.
```

Substitution works similarly. We can substitute with a python dict:

```
sage: f = sin(x*y - z)
sage: f({x: var('t'), y: z})
sin(t*z - z)
```

Also we can substitute with keywords:

```
sage: f = sin(x*y - z)
sage: f(x = t, y = z)
sin(t*z - z)
```

It was formerly the case that if there was no ambiguity of variable names, we didn't have to specify them; that still works for the moment, but the behavior is deprecated:

```
sage: f = sin(x)
sage: f(y)
doctest:...: DeprecationWarning: Substitution using function-call syntax and unnamed arguments is dep
sin(y)
sage: f(pi)
0
```

However if there is ambiguity, we should explicitly state what variables we're substituting for:

```
sage: f = sin(2*pi*x/y)
sage: f(x=4)
sin(8*pi/y)
```

We can also make a `CallableSymbolicExpression`, which is a `SymbolicExpression` that is a function of specified variables in a fixed order. Each `SymbolicExpression` has a `function(...)` method that is used to create a `CallableSymbolicExpression`, as illustrated below:

```
sage: u = log((2-x)/(y+5))
sage: f = u.function(x, y); f
(x, y) |--> log(-(x - 2)/(y + 5))
```

There is an easier way of creating a `CallableSymbolicExpression`, which relies on the Sage preparser.

```
sage: f(x,y) = log(x)*cos(y); f
(x, y) |--> log(x)*cos(y)
```

Then we have fixed an order of variables and there is no ambiguity substituting or evaluating:

```
sage: f(x,y) = log((2-x)/(y+5))
sage: f(7,t)
log(-5/(t + 5))
```

Some further examples:

```
sage: f = 5*sin(x)
sage: f
5*sin(x)
sage: f(x=2)
5*sin(2)
sage: f(x=pi)
0
sage: float(f(x=pi))
0.0
```

Another example:

```
sage: f = integrate(1/sqrt(9+x^2), x); f
arcsinh(1/3*x)
sage: f(x=3)
arcsinh(1)
sage: f.derivative(x)
1/3/sqrt(1/9*x^2 + 1)
```

We compute the length of the parabola from 0 to 2:

```
sage: x = var('x')
sage: y = x^2
sage: dy = derivative(y,x)
sage: z = integral(sqrt(1 + dy^2), x, 0, 2)
sage: z
sqrt(17) + 1/4*arcsinh(4)
sage: n(z,200)
4.6467837624329358733826155674904591885104869874232887508703
sage: float(z)
4.6467837624329356
```

We test pickling:

```
sage: x, y = var('x,y')
sage: f = -sqrt(pi)*(x^3 + sin(x/cos(y)))
sage: bool(loads(dumps(f)) == f)
True
```

Coercion examples:

We coerce various symbolic expressions into the complex numbers:

```
sage: CC(I)
1.00000000000000*I
sage: CC(2*I)
2.00000000000000*I
sage: ComplexField(200)(2*I)
2.0000000000000000000000000000000000000000000000000000000000*I
sage: ComplexField(200)(sin(I))
1.1752011936438014568823818505956008151557179813340958702296*I
sage: f = sin(I) + cos(I/2); f
sin(I) + cos(1/2*I)
sage: CC(f)
1.12762596520638 + 1.17520119364380*I
sage: ComplexField(200)(f)
1.1276259652063807852262251614026720125478471180986674836290 + 1.1752011936438014568823818505956008151
sage: ComplexField(100)(f)
1.1276259652063807852262251614 + 1.1752011936438014568823818506*I
```

We illustrate construction of an inverse sum where each denominator has a new variable name:

```
sage: f = sum(1/var('n%s'%i)^i for i in range(10))
sage: f
1/n1 + 1/n2^2 + 1/n3^3 + 1/n4^4 + 1/n5^5 + 1/n6^6 + 1/n7^7 + 1/n8^8 + 1/n9^9 + 1
```

Note that after calling var, the variables are immediately available for use:

```
sage: (n1 + n2)^5
(n1 + n2)^5
```

We can, of course, substitute:

```
sage: f(n9=9,n7=n6)
1/n1 + 1/n2^2 + 1/n3^3 + 1/n4^4 + 1/n5^5 + 1/n6^6 + 1/n6^7 + 1/n8^8 + 387420490/387420489
```

TESTS:

Substitution:

```
sage: f = x
sage: f(x=5)
5
```

Simplifying expressions involving scientific notation:

```
sage: k = var('k')
sage: a0 = 2e-06; a1 = 12
sage: c = a1 + a0*k; c
(2.00000000000000e-6)*k + 12
sage: sqrt(c)
sqrt((2.00000000000000e-6)*k + 12)
sage: sqrt(c^3)
sqrt(((2.00000000000000e-6)*k + 12)^3)
```

The symbolic Calculus package uses its own copy of maxima for simplification, etc., which is separate from the default system-wide version:

```
sage: maxima.eval('[x,y]: [1,2]')
'[1,2]'
sage: maxima.eval('expand((x+y)^3)')
'27'
```

If the copy of maxima used by the symbolic calculus package were the same as the default one, then the following would return 27, which would be very confusing indeed!

```
sage: x, y = var('x,y')
sage: expand((x+y)^3)
x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

Set x to be 5 in maxima:

```
sage: maxima('x: 5')
5
sage: maxima('x + x + %pi')
%pi+10
```

This simplification is done using maxima (behind the scenes):

```
sage: x + x + pi
 pi + 2*x
```

Note that x is still x, since the maxima used by the calculus package is different than the one in the interactive interpreter.

Check to see that the problem with the variables method mentioned in Trac ticket #3779 is actually fixed:

```
sage: f = function('F',x)
sage: diff(f*SR(1),x)
D[0](F)(x)
```

clear_functions()

Clear all user-defined functions from the symbol tables.

EXAMPLES:

```
sage: f = function('foo')
sage: id_f1 = id(f)
sage: id_f1 == id(function('foo'))
True
sage: clear_functions()
sage: id_f1 == id(function('foo'))
False
```

dummy_diff(**args*)

This function is called when 'diff' appears in a Maxima string.

EXAMPLES:

```
sage: from sage.calculus.calculus import dummy_diff
sage: x,y = var('x,y')
sage: dummy_diff(sin(x*y), x, SR(2), y, SR(1))
-x*y^2*cos(x*y) - 2*y*sin(x*y)
```

Here the function is used implicitly:

```
sage: a = var('a')
sage: f = function('cr', a)
sage: g = f.diff(a); g
D[0](cr)(a)
```

dummy_limit(**args*)

This function is called to create formal wrappers of limits that Maxima can't compute:

EXAMPLES:

```
sage: a = lim(exp(x^2)*(1-erf(x)), x=infinity); a
limit(-e^(x^2)*erf(x) + e^(x^2), x, +Infinity)
sage: a = sage.calculus.calculus.dummy_limit(sin(x)/x, x, 0);a
limit(sin(x)/x, x, 0)
```

function(*s, *args*)

Create a formal symbolic function with the name *s*.

EXAMPLES:

```
sage: var('a, b')
(a, b)
sage: f = function('cr', a)
sage: g = f.diff(a).integral(b)
sage: g
b*D[0](cr)(a)
```

In Sage 4.0, you need to use `substitute_function()` to replace all occurrences of a function with another:

```
sage: g.substitute_function(cr, cos)
-b*sin(a)

sage: g.substitute_function(cr, (sin(x) + cos(x)).function(x))
-(sin(a) - cos(a))*b
```

In Sage 4.0, basic arithmetic with unevaluated functions is no longer supported:

```
sage: x = var('x')
sage: f = function('f')
sage: 2*f
...
TypeError: unsupported operand parent(s) for '*': 'Integer Ring' and '<type 'sage.symbolic.funct
```

You now need to evaluate the function in order to do the arithmetic:

```
sage: 2*f(x)
2*f(x)
```

integral(*expression, v=None, a=None, b=None, algorithm='maxima'*)

Returns the indefinite integral with respect to the variable v, ignoring the constant of integration. Or, if endpoints a and b are specified, returns the definite integral over the interval $[a, b]$.

If `self` has only one variable, then it returns the integral with respect to that variable.

INPUT:

- `v` - (optional) a variable or variable name
- `a` - (optional) lower endpoint of definite integral

- b - (optional) upper endpoint of definite integral
- algorithm - (default: 'maxima') one of
  - 'maxima' - use maxima (the default)
  - 'sympy' - use sympy (also in Sage)
  - 'mathematica_free' - use http://integrals.wolfram.com/

EXAMPLES:

```
sage: x = var('x')
sage: h = sin(x)/(cos(x))^2
sage: h.integral(x)
1/cos(x)

sage: f = x^2/(x+1)^3
sage: f.integral()
1/2*(4*x + 3)/(x^2 + 2*x + 1) + log(x + 1)

sage: f = x*cos(x^2)
sage: f.integral(x, 0, sqrt(pi))
0
sage: f.integral(a=-pi, b=pi)
0

sage: f(x) = sin(x)
sage: f.integral(x, 0, pi/2)
1
```

The variable and endpoints are both optional:

```
sage: integral(sin(x))
-cos(x)
sage: integral(sin(x), var('y'))
y*sin(x)
sage: integral(sin(x), pi, 2*pi)
-2
sage: integral(sin(x), var('y'), pi, 2*pi)
pi*sin(x)
```

Constraints are sometimes needed:

```
sage: var('x, n')
(x, n)
sage: integral(x^n,x)
...
TypeError: Computation failed since Maxima requested additional constraints (try the command 'as
Is  n+1  zero or nonzero?
sage: assume(n > 0)
sage: integral(x^n,x)
x^(n + 1)/(n + 1)
sage: forget()
```

Usually the constraints are of sign, but others are possible:

```
sage: assume(n==-1)
sage: integral(x^n,x)
log(x)
```

Note that an exception is raised when a definite integral is divergent.

```
sage: forget()

sage: integrate(1/x^3,x,0,1)
...
ValueError: Integral is divergent.
sage: integrate(1/x^3,x,-1,3)
...
ValueError: Integral is divergent.
```

Note: Above, putting assume(n == -1) does not yield the right behavior.

The examples in the Maxima documentation:

```
sage: var('x, y, z, b')
(x, y, z, b)
sage: integral(sin(x)^3)
1/3*cos(x)^3 - cos(x)
sage: integral(x/sqrt(b^2-x^2))
x*log(2*b + 2*sqrt(b^2 - x^2))
sage: integral(x/sqrt(b^2-x^2), x)
-sqrt(b^2 - x^2)
sage: integral(cos(x)^2 * exp(x), x, 0, pi)
3/5*e^pi - 3/5
sage: integral(x^2 * exp(-x^2), x, -oo, oo)
1/2*sqrt(pi)
```

We integrate the same function in both Mathematica and Sage (via Maxima):

```
sage: _ = var('x, y, z')
sage: f = sin(x^2) + y^z
sage: g = mathematica(f)                              # optional  -- requires mathematica
sage: print g                                         # optional -- requires mathematica
          z        2
         y  + Sin[x ]
sage: print g.Integrate(x)                            # optional -- requires mathematica
             z        Pi                 2
          x y  + Sqrt[--] FresnelS[Sqrt[--] x]
                   2                  Pi
sage: print f.integral(x)
y^z*x + 1/8*((I - 1)*sqrt(2)*erf((1/2*I - 1/2)*sqrt(2)*x) + (I + 1)*sqrt(2)*erf((1/2*I + 1/2)*sq
```

Alternatively, just use algorithm='mathematica_free' to integrate via Mathematica over the internet (deos NOT require a mathematica license!):

```
sage: _ = var('x, y, z')
sage: f = sin(x^2) + y^z
sage: f.integrate(algorithm="mathematica_free")       # optional -- requires internet
sqrt(pi)*sqrt(1/2)*fresnels(sqrt(2)*x/sqrt(pi)) + y^z*x
```

We can also use Sympy:

```
sage: _ = var('x, y, z')
sage: (x^y-z).integrate(y)
-y*z + x^y/log(x)
sage: (x^y-z).integrate(y,algorithm="sympy")
-y*z + x^y/log(x)
```

We integrate the above function in maple now:

```
sage: g = maple(f); g                                  # optional -- requires maple
sin(x^2)+y^z
sage: g.integrate(x)                                   # optional -- requires maple
1/2*2^(1/2)*Pi^(1/2)*FresnelS(2^(1/2)/Pi^(1/2)*x)+y^z*x
```

We next integrate a function with no closed form integral. Notice that the answer comes back as an expression that contains an integral itself.

```
sage: A = integral(1/ ((x-4) * (x^3+2*x+1)), x); A
1/73*log(x - 4) - 1/73*integrate((x^2 + 4*x + 18)/(x^3 + 2*x + 1), x)
```

We now show that floats are not converted to rationals automatically since we by default have keepfloat: true in maxima.

```
sage: integral(e^(-x^2),x, 0, 0.1)
0.0562314580091*sqrt(pi)
```

ALIASES: integral() and integrate() are the same.

EXAMPLES: Here is example where we have to use assume:

```
sage: a,b = var('a,b')
sage: integrate(1/(x^3 *(a+b*x)^(1/3)), x)
...
TypeError: Computation failed since Maxima requested additional constraints (try the command 'as
Is  a  positive or negative?
```

So we just assume that $a > 0$ and the integral works:

```
sage: assume(a>0)
sage: integrate(1/(x^3 *(a+b*x)^(1/3)), x)
2/9*sqrt(3)*b^2*arctan(1/3*(2*(b*x + a)^(1/3) + a^(1/3))*sqrt(3)/a^(1/3))/a^(7/3) + 2/9*b^2*log(
```

TESTS:

The following integral was broken prior to Maxima 5.15.0 - see #3013

```
sage: integrate(sin(x)*cos(10*x)*log(x))
1/18*log(x)*cos(9*x) - 1/22*log(x)*cos(11*x) - 1/18*integrate(cos(9*x)/x, x) + 1/22*integrate(cc
```

It is no longer possible to use certain functions without an explicit variable. Instead, evaluate the function at a variable, and then take the integral:

```
sage: integrate(sin)
...
TypeError

sage: integrate(sin(x))
-cos(x)
sage: integrate(sin(x), 0, 1)
-cos(1) + 1
```

integrate(*expression, v=None, a=None, b=None, algorithm='maxima'*)

Returns the indefinite integral with respect to the variable v, ignoring the constant of integration. Or, if endpoints a and b are specified, returns the definite integral over the interval $[a, b]$.

If `self` has only one variable, then it returns the integral with respect to that variable.

INPUT:

- v - (optional) a variable or variable name
- a - (optional) lower endpoint of definite integral
- b - (optional) upper endpoint of definite integral
- algorithm - (default: 'maxima') one of
  - 'maxima' - use maxima (the default)
  - 'sympy' - use sympy (also in Sage)
  - 'mathematica_free' - use http://integrals.wolfram.com/

EXAMPLES:

```
sage: x = var('x')
sage: h = sin(x)/(cos(x))^2
sage: h.integral(x)
1/cos(x)


sage: f = x^2/(x+1)^3
sage: f.integral()
1/2*(4*x + 3)/(x^2 + 2*x + 1) + log(x + 1)


sage: f = x*cos(x^2)
sage: f.integral(x, 0, sqrt(pi))
0
sage: f.integral(a=-pi, b=pi)
0


sage: f(x) = sin(x)
sage: f.integral(x, 0, pi/2)
1
```

The variable and endpoints are both optional:

```
sage: integral(sin(x))
-cos(x)
sage: integral(sin(x), var('y'))
y*sin(x)
sage: integral(sin(x), pi, 2*pi)
-2
sage: integral(sin(x), var('y'), pi, 2*pi)
pi*sin(x)
```

Constraints are sometimes needed:

```
sage: var('x, n')
(x, n)
sage: integral(x^n,x)
...
TypeError: Computation failed since Maxima requested additional constraints (try the command 'as
Is  n+1  zero or nonzero?
sage: assume(n > 0)
sage: integral(x^n,x)
x^(n + 1)/(n + 1)
sage: forget()
```

Usually the constraints are of sign, but others are possible:

```
sage: assume(n==-1)
sage: integral(x^n,x)
log(x)
```

Note that an exception is raised when a definite integral is divergent.

```
sage: forget()

sage: integrate(1/x^3,x,0,1)
...
ValueError: Integral is divergent.
sage: integrate(1/x^3,x,-1,3)
...
ValueError: Integral is divergent.
```

Note: Above, putting assume(n == -1) does not yield the right behavior.

The examples in the Maxima documentation:

```
sage: var('x, y, z, b')
(x, y, z, b)
sage: integral(sin(x)^3)
1/3*cos(x)^3 - cos(x)
sage: integral(x/sqrt(b^2-x^2))
x*log(2*b + 2*sqrt(b^2 - x^2))
sage: integral(x/sqrt(b^2-x^2), x)
-sqrt(b^2 - x^2)
sage: integral(cos(x)^2 * exp(x), x, 0, pi)
3/5*e^pi - 3/5
sage: integral(x^2 * exp(-x^2), x, -oo, oo)
1/2*sqrt(pi)
```

We integrate the same function in both Mathematica and Sage (via Maxima):

```
sage: _ = var('x, y, z')
sage: f = sin(x^2) + y^z
sage: g = mathematica(f)                              # optional  -- requires mathematica
sage: print g                                         # optional -- requires mathematica
          z        2
         y  + Sin[x ]
sage: print g.Integrate(x)                            # optional -- requires mathematica
            z        Pi                 2
         x y  + Sqrt[--] FresnelS[Sqrt[--] x]
                     2                  Pi
sage: print f.integral(x)
y^z*x + 1/8*((I - 1)*sqrt(2)*erf((1/2*I - 1/2)*sqrt(2)*x) + (I + 1)*sqrt(2)*erf((1/2*I + 1/2)*sq
```

Alternatively, just use algorithm='mathematica_free' to integrate via Mathematica over the internet (deos NOT require a mathematica license!):

```
sage: _ = var('x, y, z')
sage: f = sin(x^2) + y^z
sage: f.integrate(algorithm="mathematica_free")       # optional -- requires internet
sqrt(pi)*sqrt(1/2)*fresnels(sqrt(2)*x/sqrt(pi)) + y^z*x
```

We can also use Sympy:

```
sage: _ = var('x, y, z')
sage: (x^y-z).integrate(y)
-y*z + x^y/log(x)
sage: (x^y-z).integrate(y,algorithm="sympy")
-y*z + x^y/log(x)
```

We integrate the above function in maple now:

```
sage: g = maple(f); g                                  # optional -- requires maple
sin(x^2)+y^z
sage: g.integrate(x)                                   # optional -- requires maple
1/2*2^(1/2)*Pi^(1/2)*FresnelS(2^(1/2)/Pi^(1/2)*x)+y^z*x
```

We next integrate a function with no closed form integral. Notice that the answer comes back as an expression that contains an integral itself.

```
sage: A = integral(1/ ((x-4) * (x^3+2*x+1)), x); A
1/73*log(x - 4) - 1/73*integrate((x^2 + 4*x + 18)/(x^3 + 2*x + 1), x)
```

We now show that floats are not converted to rationals automatically since we by default have keepfloat: true in maxima.

```
sage: integral(e^(-x^2),x, 0, 0.1)
0.0562314580091*sqrt(pi)
```

ALIASES: integral() and integrate() are the same.

EXAMPLES: Here is example where we have to use assume:

```
sage: a,b = var('a,b')
sage: integrate(1/(x^3 *(a+b*x)^(1/3)), x)
...
TypeError: Computation failed since Maxima requested additional constraints (try the command 'as
Is  a  positive or negative?
```

So we just assume that $a > 0$ and the integral works:

```
sage: assume(a>0)
sage: integrate(1/(x^3 *(a+b*x)^(1/3)), x)
2/9*sqrt(3)*b^2*arctan(1/3*(2*(b*x + a)^(1/3) + a^(1/3))*sqrt(3)/a^(1/3))/a^(7/3) + 2/9*b^2*log(
```

TESTS:

The following integral was broken prior to Maxima 5.15.0 - see #3013

```
sage: integrate(sin(x)*cos(10*x)*log(x))
1/18*log(x)*cos(9*x) - 1/22*log(x)*cos(11*x) - 1/18*integrate(cos(9*x)/x, x) + 1/22*integrate(co
```

It is no longer possible to use certain functions without an explicit variable. Instead, evaluate the function at a variable, and then take the integral:

```
sage: integrate(sin)
...
TypeError

sage: integrate(sin(x))
-cos(x)
sage: integrate(sin(x), 0, 1)
-cos(1) + 1
```

inverse_laplace(*ex, t, s*)

Attempts to compute the inverse Laplace transform of `self` with respect to the variable t and transform parameter s. If this function cannot find a solution, a formal function is returned.

The function that is returned may be be viewed as a function of s.

DEFINITION: The inverse Laplace transform of a function $F(s)$, is the function $f(t)$ defined by

$$F(s) = \frac{1}{2\pi i}\int_{\gamma-i\infty}^{\gamma+i\infty} e^{st}F(s)dt,$$

where γ is chosen so that the contour path of integration is in the region of convergence of $F(s)$.

EXAMPLES:

```
sage: var('w, m')
(w, m)
sage: f = (1/(w^2+10)).inverse_laplace(w, m); f
1/10*sqrt(10)*sin(sqrt(10)*m)
sage: laplace(f, m, w)
1/(w^2 + 10)

sage: f(t) = t*cos(t)
sage: s = var('s')
sage: L = laplace(f, t, s); L
t |--> 2*s^2/(s^2 + 1)^2 - 1/(s^2 + 1)
sage: inverse_laplace(L, s, t)
t |--> t*cos(t)
sage: inverse_laplace(1/(s^3+1), s, t)
1/3*(sqrt(3)*sin(1/2*sqrt(3)*t) - cos(1/2*sqrt(3)*t))*e^(1/2*t) + 1/3*e^(-t)
```

No explicit inverse Laplace transform, so one is returned formally as a function `ilt`:

```
sage: inverse_laplace(cos(s), s, t)
ilt(cos(s), s, t)
```

laplace(*ex, t, s*)

Attempts to compute and return the Laplace transform of `self` with respect to the variable t and transform parameter s. If this function cannot find a solution, a formal function is returned.

The function that is returned may be be viewed as a function of s.

DEFINITION: The Laplace transform of a function $f(t)$, defined for all real numbers $t \geq 0$, is the function $F(s)$ defined by

$$F(s) = \int_0^\infty e^{-st}f(t)dt.$$

EXAMPLES: We compute a few Laplace transforms:

```
sage: var('x, s, z, t, t0')
(x, s, z, t, t0)
sage: sin(x).laplace(x, s)
1/(s^2 + 1)
sage: (z + exp(x)).laplace(x, s)
z/s + 1/(s - 1)
sage: log(t/t0).laplace(t, s)
 -(euler_gamma + log(s) + log(t0))/s
```

We do a formal calculation:

```
sage: f = function('f', x)
sage: g = f.diff(x); g
D[0](f)(x)
sage: g.laplace(x, s)
s*laplace(f(x), x, s) - f(0)
```

EXAMPLE: A BATTLE BETWEEN the X-women and the Y-men (by David Joyner): Solve

$$x' = -16y, x(0) = 270, y' = -x + 1, y(0) = 90.$$

This models a fight between two sides, the "X-women" and the "Y-men", where the X-women have 270 initially and the Y-men have 90, but the Y-men are better at fighting, because of the higher factor of "-16" vs "-1", and also get an occasional reinforcement, because of the "+1" term.

```
sage: var('t')
t
sage: t = var('t')
sage: x = function('x', t)
sage: y = function('y', t)
sage: de1 = x.diff(t) + 16*y
sage: de2 = y.diff(t) + x - 1
sage: de1.laplace(t, s)
s*laplace(x(t), t, s) + 16*laplace(y(t), t, s) - x(0)
sage: de2.laplace(t, s)
s*laplace(y(t), t, s) - 1/s + laplace(x(t), t, s) - y(0)
```

Next we form the augmented matrix of the above system:

```
sage: A = matrix([[s, 16, 270],[1, s, 90+1/s]])
sage: E = A.echelon_form()
sage: xt = E[0,2].inverse_laplace(s,t)
sage: yt = E[1,2].inverse_laplace(s,t)
sage: xt
629/2*e^(-4*t) - 91/2*e^(4*t) + 1
sage: yt
629/8*e^(-4*t) + 91/8*e^(4*t)
sage: p1 = plot(xt,0,1/2,rgbcolor=(1,0,0))
sage: p2 = plot(yt,0,1/2,rgbcolor=(0,1,0))
sage: (p1+p2).save()
```

Another example:

```
sage: var('a,s,t')
(a, s, t)
sage: f = exp (2*t + a) * sin(t) * t; f
t*e^(a + 2*t)*sin(t)
sage: L = laplace(f, t, s); L
2*(s - 2)*e^a/(s^2 - 4*s + 5)^2
sage: inverse_laplace(L, s, t)
t*e^(a + 2*t)*sin(t)
```

Unable to compute solution:

```
sage: laplace(1/s, s, t)
laplace(1/s, s, t)
```

lim(*ex, dir=None, taylor=False, algorithm='maxima', **argv*)

Return the limit as the variable v approaches a from the given direction.

```
expr.limit(x = a)
expr.limit(x = a, dir='above')
```

INPUT:

- `dir` - (default: None); dir may have the value 'plus' (or 'above') for a limit from above, 'minus' (or 'below') for a limit from below, or may be omitted (implying a two-sided limit is to be computed).
- `taylor` - (default: False); if True, use Taylor series, which allows more integrals to be computed (but may also crash in some obscure cases due to bugs in Maxima).
- `**argv` - 1 named parameter

Note: The output may also use 'und' (undefined), 'ind' (indefinite but bounded), and 'infinity' (complex infinity).

EXAMPLES:

```
sage: x = var('x')
sage: f = (1+1/x)^x
sage: f.limit(x = oo)
e
sage: f.limit(x = 5)
7776/3125
sage: f.limit(x = 1.2)
2.06961575467...
sage: f.limit(x = I, taylor=True)
(-I + 1)^I
sage: f(x=1.2)
2.0696157546720...
sage: f(x=I)
(-I + 1)^I
sage: CDF(f(x=I))
2.06287223508 + 0.74500706218*I
sage: CDF(f.limit(x = I))
2.06287223508 + 0.74500706218*I
```

More examples:

```
sage: limit(x*log(x), x = 0, dir='above')
0
sage: lim((x+1)^(1/x),x = 0)
e
sage: lim(e^x/x, x = oo)
+Infinity
sage: lim(e^x/x, x = -oo)
0
sage: lim(-e^x/x, x = oo)
-Infinity
sage: lim((cos(x))/(x^2), x = 0)
+Infinity
sage: lim(sqrt(x^2+1) - x, x = oo)
0
sage: lim(x^2/(sec(x)-1), x=0)
2
sage: lim(cos(x)/(cos(x)-1), x=0)
-Infinity
sage: lim(x*sin(1/x), x=0)
0
```

```
sage: f = log(log(x))/log(x)
sage: forget(); assume(x<-2); lim(f, x=0, taylor=True)
limit(log(log(x))/log(x), x, 0)
```

Here ind means "indefinite but bounded":

```
sage: lim(sin(1/x), x = 0)
ind
```

limit (*ex, dir=None, taylor=False, algorithm='maxima', **argv*)

Return the limit as the variable v approaches a from the given direction.

```
expr.limit(x = a)
expr.limit(x = a, dir='above')
```

INPUT:

- `dir` - (default: None); dir may have the value 'plus' (or 'above') for a limit from above, 'minus' (or 'below') for a limit from below, or may be omitted (implying a two-sided limit is to be computed).
- `taylor` - (default: False); if True, use Taylor series, which allows more integrals to be computed (but may also crash in some obscure cases due to bugs in Maxima).
- `**argv` - 1 named parameter

Note: The output may also use 'und' (undefined), 'ind' (indefinite but bounded), and 'infinity' (complex infinity).

EXAMPLES:

```
sage: x = var('x')
sage: f = (1+1/x)^x
sage: f.limit(x = oo)
e
sage: f.limit(x = 5)
7776/3125
sage: f.limit(x = 1.2)
2.06961575467...
sage: f.limit(x = I, taylor=True)
(-I + 1)^I
sage: f(x=1.2)
2.0696157546720...
sage: f(x=I)
(-I + 1)^I
sage: CDF(f(x=I))
2.06287223508 + 0.74500706218*I
sage: CDF(f.limit(x = I))
2.06287223508 + 0.74500706218*I
```

More examples:

```
sage: limit(x*log(x), x = 0, dir='above')
0
sage: lim((x+1)^(1/x),x = 0)
e
sage: lim(e^x/x, x = oo)
+Infinity
sage: lim(e^x/x, x = -oo)
0
sage: lim(-e^x/x, x = oo)
```

```
-Infinity
sage: lim((cos(x))/(x^2), x = 0)
+Infinity
sage: lim(sqrt(x^2+1) - x, x = oo)
0
sage: lim(x^2/(sec(x)-1), x=0)
2
sage: lim(cos(x)/(cos(x)-1), x=0)
-Infinity
sage: lim(x*sin(1/x), x=0)
0
```

```
sage: f = log(log(x))/log(x)
sage: forget(); assume(x<-2); lim(f, x=0, taylor=True)
limit(log(log(x))/log(x), x, 0)
```

Here ind means "indefinite but bounded":

```
sage: lim(sin(1/x), x = 0)
ind
```

mapped_opts(*v*)

Used internally when creating a string of options to pass to Maxima.

INPUT:

- v - an object

OUTPUT: a string.

The main use of this is to turn Python bools into lower case strings.

EXAMPLES:

```
sage: sage.calculus.calculus.mapped_opts(True)
'true'
sage: sage.calculus.calculus.mapped_opts(False)
'false'
sage: sage.calculus.calculus.mapped_opts('bar')
'bar'
```

maxima_options(***kwds*)

Used internally to create a string of options to pass to Maxima.

EXAMPLES:

```
sage: sage.calculus.calculus.maxima_options(an_option=True, another=False, foo='bar')
'an_option=true,foo=bar,another=false'
```

minpoly(*ex, var='x', algorithm=None, bits=None, degree=None, epsilon=0*)

Return the minimal polynomial of self, if possible.

INPUT:

- var - polynomial variable name (default 'x')
- algorithm - 'algebraic' or 'numerical' (default both, algebraic first)
- bits - the number of bits to use in numerical approx
- degree - the expected algebraic degree

- `epsilon` - return without error as long as f(self) epsilon, in the case that the result cannot be proven.

  All of the above parameters are optional, with epsilon=0, bits and degree tested up to 1000 and 24 by default respectively. The numerical algorithm will be faster if bits and/or degree are given explicitly. The algebraic algorithm ignores the last three parameters.

OUTPUT: The minimal polynomial of self. If the numerical algorithm is used then it is proved symbolically when epsilon=0 (default).

If the minimal polynomial could not be found, two distinct kinds of errors are raised. If no reasonable candidate was found with the given bit/degree parameters, a `ValueError` will be raised. If a reasonable candidate was found but (perhaps due to limits in the underlying symbolic package) was unable to be proved correct, a `NotImplementedError` will be raised.

ALGORITHM: Two distinct algorithms are used, depending on the algorithm parameter. By default, the algebraic algorithm is attempted first, then the numerical one.

Algebraic: Attempt to evaluate this expression in QQbar, using cyclotomic fields to resolve exponential and trig functions at rational multiples of pi, field extensions to handle roots and rational exponents, and computing compositums to represent the full expression as an element of a number field where the minimal polynomial can be computed exactly. The bits, degree, and epsilon parameters are ignored.

Numerical: Computes a numerical approximation of `self` and use PARI's algdep to get a candidate minpoly f. If $f($`self`$)$, evaluated to a higher precision, is close enough to 0 then evaluate $f($`self`$)$ symbolically, attempting to prove vanishing. If this fails, and `epsilon` is non-zero, return f if and only if $f($`self`$) <$ `epsilon`. Otherwise raise a `ValueError` (if no suitable candidate was found) or a `NotImplementedError` (if a likely candidate was found but could not be proved correct).

EXAMPLES: First some simple examples:

```
sage: sqrt(2).minpoly()
x^2 - 2
sage: minpoly(2^(1/3))
x^3 - 2
sage: minpoly(sqrt(2) + sqrt(-1))
x^4 - 2*x^2 + 9
sage: minpoly(sqrt(2)-3^(1/3))
x^6 - 6*x^4 + 6*x^3 + 12*x^2 + 36*x + 1
```

Works with trig and exponential functions too.

```
sage: sin(pi/3).minpoly()
x^2 - 3/4
sage: sin(pi/7).minpoly()
x^6 - 7/4*x^4 + 7/8*x^2 - 7/64
sage: minpoly(exp(I*pi/17))
x^16 - x^15 + x^14 - x^13 + x^12 - x^11 + x^10 - x^9 + x^8 - x^7 + x^6 - x^5 + x^4 - x^3 + x^2 -
```

Here we verify it gives the same result as the abstract number field.

```
sage: (sqrt(2) + sqrt(3) + sqrt(6)).minpoly()
x^4 - 22*x^2 - 48*x - 23
sage: K.<a,b> = NumberField([x^2-2, x^2-3])
sage: (a+b+a*b).absolute_minpoly()
x^4 - 22*x^2 - 48*x - 23
```

Here we solve a cubic and then recover it from its complicated radical expansion.

```
sage: f = x^3 - x + 1
sage: a = f.solve(x)[0].rhs(); a
-1/2*(I*sqrt(3) + 1)*(1/18*sqrt(3)*sqrt(23) - 1/2)^(1/3) - 1/6*(-I*sqrt(3) + 1)/(1/18*sqrt(3)*sq
```

```
sage: a.minpoly()
x^3 - x + 1
```

Note that simplification may be necessary to see that the minimal polynomial is correct.

```
sage: a = sqrt(2)+sqrt(3)+sqrt(5)
sage: f = a.minpoly(); f
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
sage: f(a)
((((sqrt(2) + sqrt(3) + sqrt(5))^2 - 40)*(sqrt(2) + sqrt(3) + sqrt(5))^2 + 352)*(sqrt(2) + sqrt(
sage: f(a).expand()
0
```

Here we show use of the `epsilon` parameter. That this result is actually exact can be shown using the addition formula for sin, but maxima is unable to see that.

```
sage: a = sin(pi/5)
sage: a.minpoly(algorithm='numerical')
...
NotImplementedError: Could not prove minimal polynomial x^4 - 5/4*x^2 + 5/16 (epsilon 0.00000000
sage: f = a.minpoly(algorithm='numerical', epsilon=1e-100); f
x^4 - 5/4*x^2 + 5/16
sage: f(a).numerical_approx(100)
0.00000000000000000000000000000
```

The degree must be high enough (default tops out at 24).

```
sage: a = sqrt(3) + sqrt(2)
sage: a.minpoly(algorithm='numerical', bits=100, degree=3)
...
ValueError: Could not find minimal polynomial (100 bits, degree 3).
sage: a.minpoly(algorithm='numerical', bits=100, degree=10)
x^4 - 10*x^2 + 1
```

There is a difference between algorithm='algebraic' and algorithm='numerical':

```
sage: cos(pi/22).minpoly(algorithm='algebraic')
x^10 - 11/4*x^8 + 11/4*x^6 - 77/64*x^4 + 55/256*x^2 - 11/1024
sage: cos(pi/22).minpoly(algorithm='numerical')
Traceback (most recent call last):
NotImplementedError: Could not prove minimal polynomial x^10 - 11/4*x^8 + 11/4*x^6 - 77/64*x^4 +
```

Sometimes it fails.

```
sage: sin(1).minpoly()
...
ValueError: Could not find minimal polynomial (1000 bits, degree 24).
```

Note: Failure to produce a minimal polynomial does not necessarily indicate that this number is transcendental.

AUTHORS:

- Robert Bradshaw (2007-10): numerical algorithm
- Robert Bradshaw (2008-10): algebraic algorithm

nintegral (*ex, x, a, b, desired_relative_error='1e-8', maximum_num_subintervals=200*)

Return a floating point machine precision numerical approximation to the integral of `self` from a to b, computed using floating point arithmetic via maxima.

INPUT:

- x - variable to integrate with respect to
- a - lower endpoint of integration
- b - upper endpoint of integration
- desired_relative_error - (default: '1e-8') the desired relative error
- maximum_num_subintervals - (default: 200) maxima number of subintervals

OUTPUT:

- float: approximation to the integral
- float: estimated absolute error of the approximation
- the number of integrand evaluations
- an error code:
  - 0 - no problems were encountered
  - 1 - too many subintervals were done
  - 2 - excessive roundoff error
  - 3 - extremely bad integrand behavior
  - 4 - failed to converge
  - 5 - integral is probably divergent or slowly convergent
  - 6 - the input is invalid

ALIAS: nintegrate is the same as nintegral

REMARK: There is also a function numerical_integral that implements numerical integration using the GSL C library. It is potentially much faster and applies to arbitrary user defined functions.

Also, there are limits to the precision to which Maxima can compute the integral to due to limitations in quadpack.

```
sage: f = x
sage: f = f.nintegral(x,0,1,1e-14)
...
ValueError: Maxima (via quadpack) cannot compute the integral to that precision
```

EXAMPLES:

```
sage: f(x) = exp(-sqrt(x))
sage: f.nintegral(x, 0, 1)
(0.52848223531423055, 4.163...e-11, 231, 0)
```

We can also use the numerical_integral function, which calls the GSL C library.

```
sage: numerical_integral(f, 0, 1)
(0.52848223225314706, 6.83928460...e-07)
```

Note that in exotic cases where floating point evaluation of the expression leads to the wrong value, then the output can be completely wrong:

```
sage: f = exp(pi*sqrt(163)) - 262537412640768744
```

Despite appearance, f is really very close to 0, but one gets a nonzero value since the definition of float(f) is that it makes all constants inside the expression floats, then evaluates each function and each arithmetic operation using float arithmetic:

```
sage: float(f)
-480.0
```

Computing to higher precision we see the truth:

```
sage: f.n(200)
-7.4992740280181431112064614366622348652078895136533593355718e-13
sage: f.n(300)
-7.4992740280181431112064614366266300913729246258962178935209506618170909557568196396710300
```

Now numerically integrating, we see why the answer is wrong:

```
sage: f.nintegrate(x,0,1)
(-480.00000000000011, 5.3290705182007538e-12, 21, 0)
```

It is just because every floating point evaluation of return -480.0 in floating point.

Important note: using GP/PARI one can compute numerical integrals to high precision:

```
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.5657285005610514829173563961304785900 E-127'    # 64-bit
sage: old_prec = gp.set_real_precision(50)
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.5657285005610514829173563961304785900147709554020 E-127'
sage: gp.set_real_precision(old_prec)
57
```

Note that the input function above is a string in PARI syntax.

nintegrate(*ex, x, a, b, desired_relative_error='1e-8', maximum_num_subintervals=200*)

Return a floating point machine precision numerical approximation to the integral of `self` from a to b, computed using floating point arithmetic via maxima.

INPUT:

- `x` - variable to integrate with respect to
- `a` - lower endpoint of integration
- `b` - upper endpoint of integration
- `desired_relative_error` - (default: '1e-8') the desired relative error
- `maximum_num_subintervals` - (default: 200) maxima number of subintervals

OUTPUT:

- float: approximation to the integral
- float: estimated absolute error of the approximation
- the number of integrand evaluations
- an error code:
  - `0` - no problems were encountered
  - `1` - too many subintervals were done
  - `2` - excessive roundoff error
  - `3` - extremely bad integrand behavior
  - `4` - failed to converge
  - `5` - integral is probably divergent or slowly convergent
  - `6` - the input is invalid

ALIAS: nintegrate is the same as nintegral

REMARK: There is also a function `numerical_integral` that implements numerical integration using the GSL C library. It is potentially much faster and applies to arbitrary user defined functions.

Also, there are limits to the precision to which Maxima can compute the integral to due to limitations in quadpack.

```
sage: f = x
sage: f = f.nintegral(x,0,1,1e-14)
...
ValueError: Maxima (via quadpack) cannot compute the integral to that precision
```

EXAMPLES:

```
sage: f(x) = exp(-sqrt(x))
sage: f.nintegral(x, 0, 1)
(0.52848223531423055, 4.163...e-11, 231, 0)
```

We can also use the `numerical_integral` function, which calls the GSL C library.

```
sage: numerical_integral(f, 0, 1)
(0.52848223225314706, 6.83928460...e-07)
```

Note that in exotic cases where floating point evaluation of the expression leads to the wrong value, then the output can be completely wrong:

```
sage: f = exp(pi*sqrt(163)) - 262537412640768744
```

Despite appearance, f is really very close to 0, but one gets a nonzero value since the definition of `float(f)` is that it makes all constants inside the expression floats, then evaluates each function and each arithmetic operation using float arithmetic:

```
sage: float(f)
-480.0
```

Computing to higher precision we see the truth:

```
sage: f.n(200)
-7.4992740280181431112064614366622348652078895136533593355718e-13
sage: f.n(300)
-7.4992740280181431112064614366266300913729246258962178935209506618170909557568196396710300
```

Now numerically integrating, we see why the answer is wrong:

```
sage: f.nintegrate(x,0,1)
(-480.00000000000011, 5.3290705182007538e-12, 21, 0)
```

It is just because every floating point evaluation of return -480.0 in floating point.

Important note: using GP/PARI one can compute numerical integrals to high precision:

```
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.5657285005610514829173563961304785900 E-127'        # 32-bit
'2.5657285005610514829173563961304785900 E-127'    # 64-bit
sage: old_prec = gp.set_real_precision(50)
sage: gp.eval('intnum(x=17,42,exp(-x^2)*log(x))')
'2.5657285005610514829173563961304785900147709554020 E-127'
sage: gp.set_real_precision(old_prec)
57
```

Note that the input function above is a string in PARI syntax.

symbolic_expression_from_maxima_element(*x, maxima=Maxima*)

Given an element of the calculus copy of the Maxima interface, create the corresponding Sage symbolic expression.

EXAMPLES:

```
sage: a = sage.calculus.calculus.maxima('x^(sqrt(y)+%pi) + sin(%e + %pi)')
sage: sage.calculus.calculus.symbolic_expression_from_maxima_element(a)
x^(pi + sqrt(y)) - sin(e)
sage: var('x, y')
(x, y)
sage: v = sage.calculus.calculus.maxima.vandermonde_matrix([x, y, 1/2])
sage: sage.calculus.calculus.symbolic_expression_from_maxima_element(v)
[  1   x x^2]
[  1   y y^2]
[  1 1/2 1/4]
```

symbolic_expression_from_maxima_string(*x, equals_sub=False, maxima=Maxima*)

Given a string representation of a Maxima expression, parse it and return the corresponding Sage symbolic expression.

INPUT:

- `x` - a string
- `equals_sub` - (default: False) if True, replace '=' by '==' in self
- `maxima` - (default: the calculus package's Maxima) the Maxima interpreter to use.

EXAMPLES:

```
sage: sage.calculus.calculus.symbolic_expression_from_maxima_string('x^%e + %e^%pi + %i + sin(0)')
x^e + e^pi + I
```

symbolic_expression_from_string(*s, syms=None, accept_sequence=False*)

var_cmp(*x, y*)

Return comparison of the two variables x and y, which is just the comparison of the underlying string representations of the variables. This is used internally by the Calculus package.

INPUT:

- `x, y` - symbolic variables

OUTPUT: Python integer; either -1, 0, or 1.

EXAMPLES:

```
sage: sage.calculus.calculus.var_cmp(x,x)
0
sage: sage.calculus.calculus.var_cmp(x,var('z'))
-1
sage: sage.calculus.calculus.var_cmp(x,var('a'))
1
```

3.5 Functional notation support for common calculus methods.

EXAMPLES: We illustrate each of the calculus functional functions.

```
sage: simplify(x - x)
0
sage: a = var('a')
sage: derivative(x^a + sin(x), x)
a*x^(a - 1) + cos(x)
sage: diff(x^a + sin(x), x)
a*x^(a - 1) + cos(x)
sage: derivative(x^a + sin(x), x)
a*x^(a - 1) + cos(x)
sage: integral(a*x*sin(x), x)
-(x*cos(x) - sin(x))*a
sage: integrate(a*x*sin(x), x)
-(x*cos(x) - sin(x))*a
sage: limit(a*sin(x)/x, x=0)
a
sage: taylor(a*sin(x)/x, x, 0, 4)
1/120*a*x^4 - 1/6*a*x^2 + a
sage: expand( (x-a)^3 )
-a^3 + 3*a^2*x - 3*a*x^2 + x^3
sage: laplace( e^(x+a), x, a)
e^a/(a - 1)
sage: inverse_laplace( e^a/(a-1), x, a)
ilt(e^a/(a - 1), x, a)
```

derivative (*f, *args, **kwds*)

The derivative of f.

Repeated differentiation is supported by the syntax given in the examples below.

ALIAS: diff

EXAMPLES: We differentiate a callable symbolic function:

```
sage: f(x,y) = x*y + sin(x^2) + e^(-x)
sage: f
(x, y) |--> x*y + e^(-x) + sin(x^2)
sage: derivative(f, x)
(x, y) |--> 2*x*cos(x^2) + y - e^(-x)
sage: derivative(f, y)
(x, y) |--> x
```

We differentiate a polynomial:

```
sage: t = polygen(QQ, 't')
sage: f = (1-t)^5; f
-t^5 + 5*t^4 - 10*t^3 + 10*t^2 - 5*t + 1
sage: derivative(f)
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
sage: derivative(f, t)
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
sage: derivative(f, t, t)
-20*t^3 + 60*t^2 - 60*t + 20
sage: derivative(f, t, 2)
-20*t^3 + 60*t^2 - 60*t + 20
sage: derivative(f, 2)
-20*t^3 + 60*t^2 - 60*t + 20
```

We differentiate a symbolic expression:

```
sage: var('a x')
(a, x)
sage: f = exp(sin(a - x^2))/x
sage: derivative(f, x)
-2*e^(sin(-x^2 + a))*cos(-x^2 + a) - e^(sin(-x^2 + a))/x^2
sage: derivative(f, a)
e^(sin(-x^2 + a))*cos(-x^2 + a)/x
```

Syntax for repeated differentiation:

```
sage: R.<u, v> = PolynomialRing(QQ)
sage: f = u^4*v^5
sage: derivative(f, u)
4*u^3*v^5
sage: f.derivative(u)   # can always use method notation too
4*u^3*v^5


sage: derivative(f, u, u)
12*u^2*v^5
sage: derivative(f, u, u, u)
24*u*v^5
sage: derivative(f, u, 3)
24*u*v^5


sage: derivative(f, u, v)
20*u^3*v^4
sage: derivative(f, u, 2, v)
60*u^2*v^4
sage: derivative(f, u, v, 2)
80*u^3*v^3
sage: derivative(f, [u, v, v])
80*u^3*v^3
```

diff (*f, *args, **kwds*)

The derivative of f.

Repeated differentiation is supported by the syntax given in the examples below.

ALIAS: diff

EXAMPLES: We differentiate a callable symbolic function:

```
sage: f(x,y) = x*y + sin(x^2) + e^(-x)
sage: f
(x, y) |--> x*y + e^(-x) + sin(x^2)
sage: derivative(f, x)
(x, y) |--> 2*x*cos(x^2) + y - e^(-x)
sage: derivative(f, y)
(x, y) |--> x
```

We differentiate a polynomial:

```
sage: t = polygen(QQ, 't')
sage: f = (1-t)^5; f
-t^5 + 5*t^4 - 10*t^3 + 10*t^2 - 5*t + 1
sage: derivative(f)
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
sage: derivative(f, t)
```

```
-5*t^4 + 20*t^3 - 30*t^2 + 20*t - 5
sage: derivative(f, t, t)
-20*t^3 + 60*t^2 - 60*t + 20
sage: derivative(f, t, 2)
-20*t^3 + 60*t^2 - 60*t + 20
sage: derivative(f, 2)
-20*t^3 + 60*t^2 - 60*t + 20
```

We differentiate a symbolic expression:

```
sage: var('a x')
(a, x)
sage: f = exp(sin(a - x^2))/x
sage: derivative(f, x)
-2*e^(sin(-x^2 + a))*cos(-x^2 + a) - e^(sin(-x^2 + a))/x^2
sage: derivative(f, a)
e^(sin(-x^2 + a))*cos(-x^2 + a)/x
```

Syntax for repeated differentiation:

```
sage: R.<u, v> = PolynomialRing(QQ)
sage: f = u^4*v^5
sage: derivative(f, u)
4*u^3*v^5
sage: f.derivative(u)   # can always use method notation too
4*u^3*v^5


sage: derivative(f, u, u)
12*u^2*v^5
sage: derivative(f, u, u, u)
24*u*v^5
sage: derivative(f, u, 3)
24*u*v^5


sage: derivative(f, u, v)
20*u^3*v^4
sage: derivative(f, u, 2, v)
60*u^2*v^4
sage: derivative(f, u, v, 2)
80*u^3*v^3
sage: derivative(f, [u, v, v])
80*u^3*v^3
```

expand(*x, *args, **kwds*)

EXAMPLES:

```
sage: a = (x-1)*(x^2 - 1); a
(x - 1)*(x^2 - 1)
sage: expand(a)
x^3 - x^2 - x + 1
```

You can also use expand on polynomial, integer, and other factorizations:

```
sage: x = polygen(ZZ)
sage: F = factor(x^12 - 1); F
(x - 1) * (x + 1) * (x^2 - x + 1) * (x^2 + 1) * (x^2 + x + 1) * (x^4 - x^2 + 1)
sage: expand(F)
```

```
x^12 - 1
sage: F.expand()
x^12 - 1
sage: F = factor(2007); F
3^2 * 223
sage: expand(F)
2007
```

Note: If you want to compute the expanded form of a polynomial arithmetic operation quickly and the coefficients of the polynomial all lie in some ring, e.g., the integers, it is vastly faster to create a polynomial ring and do the arithmetic there.

```
sage: x = polygen(ZZ)       # polynomial over a given base ring.
sage: f = sum(x^n for n in range(5))
sage: f*f                   # much faster, even if the degree is huge
x^8 + 2*x^7 + 3*x^6 + 4*x^5 + 5*x^4 + 4*x^3 + 3*x^2 + 2*x + 1
```

TESTS:

```
sage: t1 = (sqrt(3)-3)*(sqrt(3)+1)/6;
sage: tt1 = -1/sqrt(3);
sage: t2 = sqrt(3)/6;
sage: float(t1)
-0.577350269189625...
sage: float(tt1)
-0.577350269189625...
sage: float(t2)
0.28867513459481287
sage: float(expand(t1 + t2))
-0.288675134594812...
sage: float(expand(tt1 + t2))
-0.288675134594812...
```

integral (*f, *args, **kwds*)

The integral of f.

EXAMPLES:

```
sage: integral(sin(x), x)
-cos(x)
sage: integral(sin(x)^2, x, pi, 123*pi/2)
121/4*pi
sage: integral( sin(x), x, 0, pi)
2
```

We integrate a symbolic function:

```
sage: f(x,y,z) = x*y/z + sin(z)
sage: integral(f, z)
(x, y, z) |--> x*y*log(z) - cos(z)
```

```
sage: var('a,b')
(a, b)
sage: assume(b-a>0)
sage: integral( sin(x), x, a, b)
cos(a) - cos(b)
sage: forget()
```

```
sage: integral(x/(x^3-1), x)
1/3*sqrt(3)*arctan(1/3*(2*x + 1)*sqrt(3)) + 1/3*log(x - 1) - 1/6*log(x^2 + x + 1)


sage: integral( exp(-x^2), x )
1/2*sqrt(pi)*erf(x)
```

We define the Gaussian, plot and integrate it numerically and symbolically:

```
sage: f(x) = 1/(sqrt(2*pi)) * e^(-x^2/2)
sage: P = plot(f, -4, 4, hue=0.8, thickness=2)
sage: P.show(ymin=0, ymax=0.4)
sage: numerical_integral(f, -4, 4)                    # random output
(0.99993665751633376, 1.1101527003413533e-14)
sage: integrate(f, x)
x |--> 1/2*erf(1/2*sqrt(2)*x)
```

You can have Sage calculate multiple integrals. For example, consider the function $exp(y^2)$ on the region between the lines $x = y$, $x = 1$, and $y = 0$. We find the value of the integral on this region using the command:

```
sage: area = integral(integral(exp(y^2),x,0,y),y,0,1); area
1/2*e - 1/2
sage: float(area)
0.85914091422952255
```

We compute the line integral of $\sin(x)$ along the arc of the curve $x = y^4$ from $(1, -1)$ to $(1, 1)$:

```
sage: t = var('t')
sage: (x,y) = (t^4,t)
sage: (dx,dy) = (diff(x,t), diff(y,t))
sage: integral(sin(x)*dx, t,-1, 1)
0
sage: restore('x,y')    # restore the symbolic variables x and y
```

Sage is unable to do anything with the following integral:

```
sage: integral( exp(-x^2)*log(x), x )
integrate(e^(-x^2)*log(x), x)
```

Sage does not know how to compute this integral either:

```
sage: integral( exp(-x^2)*ln(x), x, 0, oo)
integrate(e^(-x^2)*log(x), x, 0, +Infinity)
```

This definite integral is easy:

```
sage: integral( ln(x)/x, x, 1, 2)
1/2*log(2)^2
```

Sage can't do this elliptic integral (yet):

```
sage: integral(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
integrate(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
```

A double integral:

```
sage: y = var('y')
sage: integral(integral(x*y^2, x, 0, y), y, -2, 2)
32/5
```

This illustrates using assumptions:

```
sage: integral(abs(x), x, 0, 5)
25/2
sage: integral(abs(x), x, 0, a)
integrate(abs(x), x, 0, a)
sage: assume(a>0)
sage: integral(abs(x), x, 0, a)
1/2*a^2
sage: forget()       # forget the assumptions.
```

We integrate and differentiate a huge mess:

```
sage: f = (x^2-1+3*(1+x^2)^(1/3))/(1+x^2)^(2/3)*x/(x^2+2)^2
sage: g = integral(f, x)
sage: h = f - diff(g, x)
```

```
sage: [float(h(i)) for i in range(5)] #random

[0.0,
 -1.1102230246251565e-16,
 -5.5511151231257827e-17,
 -5.5511151231257827e-17,
 -6.9388939039072284e-17]
sage: h.factor()
0
sage: bool(h == 0)
True
```

integrate(*f, *args, **kwds*)

The integral of f.

EXAMPLES:

```
sage: integral(sin(x), x)
-cos(x)
sage: integral(sin(x)^2, x, pi, 123*pi/2)
121/4*pi
sage: integral( sin(x), x, 0, pi)
2
```

We integrate a symbolic function:

```
sage: f(x,y,z) = x*y/z + sin(z)
sage: integral(f, z)
(x, y, z) |--> x*y*log(z) - cos(z)
```

```
sage: var('a,b')
(a, b)
sage: assume(b-a>0)
sage: integral( sin(x), x, a, b)
cos(a) - cos(b)
sage: forget()
```

```
sage: integral(x/(x^3-1), x)
1/3*sqrt(3)*arctan(1/3*(2*x + 1)*sqrt(3)) + 1/3*log(x - 1) - 1/6*log(x^2 + x + 1)
```

```
sage: integral( exp(-x^2), x )
1/2*sqrt(pi)*erf(x)
```

We define the Gaussian, plot and integrate it numerically and symbolically:

```
sage: f(x) = 1/(sqrt(2*pi)) * e^(-x^2/2)
sage: P = plot(f, -4, 4, hue=0.8, thickness=2)
sage: P.show(ymin=0, ymax=0.4)
sage: numerical_integral(f, -4, 4)                    # random output
(0.99993665751633376, 1.1101527003413533e-14)
sage: integrate(f, x)
x |--> 1/2*erf(1/2*sqrt(2)*x)
```

You can have Sage calculate multiple integrals. For example, consider the function $exp(y^2)$ on the region between the lines $x = y$, $x = 1$, and $y = 0$. We find the value of the integral on this region using the command:

```
sage: area = integral(integral(exp(y^2),x,0,y),y,0,1); area
1/2*e - 1/2
sage: float(area)
0.85914091422952255
```

We compute the line integral of $\sin(x)$ along the arc of the curve $x = y^4$ from $(1, -1)$ to $(1, 1)$:

```
sage: t = var('t')
sage: (x,y) = (t^4,t)
sage: (dx,dy) = (diff(x,t), diff(y,t))
sage: integral(sin(x)*dx, t,-1, 1)
0
sage: restore('x,y')    # restore the symbolic variables x and y
```

Sage is unable to do anything with the following integral:

```
sage: integral( exp(-x^2)*log(x), x )
integrate(e^(-x^2)*log(x), x)
```

Sage does not know how to compute this integral either:

```
sage: integral( exp(-x^2)*ln(x), x, 0, oo)
integrate(e^(-x^2)*log(x), x, 0, +Infinity)
```

This definite integral is easy:

```
sage: integral( ln(x)/x, x, 1, 2)
1/2*log(2)^2
```

Sage can't do this elliptic integral (yet):

```
sage: integral(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
integrate(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
```

A double integral:

```
sage: y = var('y')
sage: integral(integral(x*y^2, x, 0, y), y, -2, 2)
32/5
```

This illustrates using assumptions:

```
sage: integral(abs(x), x, 0, 5)
25/2
sage: integral(abs(x), x, 0, a)
integrate(abs(x), x, 0, a)
sage: assume(a>0)
sage: integral(abs(x), x, 0, a)
1/2*a^2
sage: forget()       # forget the assumptions.
```

We integrate and differentiate a huge mess:

```
sage: f = (x^2-1+3*(1+x^2)^(1/3))/(1+x^2)^(2/3)*x/(x^2+2)^2
sage: g = integral(f, x)
sage: h = f - diff(g, x)
```

```
sage: [float(h(i)) for i in range(5)] #random

[0.0,
 -1.1102230246251565e-16,
 -5.5511151231257827e-17,
 -5.5511151231257827e-17,
 -6.9388939039072284e-17]
sage: h.factor()
0
sage: bool(h == 0)
True
```

lim(*f, dir=None, taylor=False, **argv*)

Return the limit as the variable v approaches a from the given direction.

```
limit(expr, x = a)
limit(expr, x = a, dir='above')
```

INPUT:

- `dir` - (default: None); dir may have the value 'plus' (or 'above') for a limit from above, 'minus' (or 'below') for a limit from below, or may be omitted (implying a two-sided limit is to be computed).
- `taylor` - (default: False); if True, use Taylor series, which allows more integrals to be computed (but may also crash in some obscure cases due to bugs in Maxima).
- `\*\*argv` - 1 named parameter

ALIAS: You can also use lim instead of limit.

EXAMPLES:

```
sage: limit(sin(x)/x, x=0)
1
sage: limit(exp(x), x=oo)
+Infinity
sage: lim(exp(x), x=-oo)
0
```

```
sage: lim(1/x, x=0)
und
sage: limit(sqrt(x^2+x+1)+x, taylor=True, x=-oo)
-1/2
sage: limit((tan(sin(x)) - sin(tan(x)))/x^7, taylor=True, x=0)
1/30
```

Sage does not know how to do this limit (which is 0), so it returns it unevaluated:

```
sage: lim(exp(x^2)*(1-erf(x)), x=infinity)
limit(-e^(x^2)*erf(x) + e^(x^2), x, +Infinity)
```

limit (*f, dir=None, taylor=False, **argv*)

Return the limit as the variable v approaches a from the given direction.

```
limit(expr, x = a)
limit(expr, x = a, dir='above')
```

INPUT:

- `dir` - (default: None); dir may have the value 'plus' (or 'above') for a limit from above, 'minus' (or 'below') for a limit from below, or may be omitted (implying a two-sided limit is to be computed).
- `taylor` - (default: False); if True, use Taylor series, which allows more integrals to be computed (but may also crash in some obscure cases due to bugs in Maxima).
- `\*\*argv` - 1 named parameter

ALIAS: You can also use lim instead of limit.

EXAMPLES:

```
sage: limit(sin(x)/x, x=0)
1
sage: limit(exp(x), x=oo)
+Infinity
sage: lim(exp(x), x=-oo)
0
sage: lim(1/x, x=0)
und
sage: limit(sqrt(x^2+x+1)+x, taylor=True, x=-oo)
-1/2
sage: limit((tan(sin(x)) - sin(tan(x)))/x^7, taylor=True, x=0)
1/30
```

Sage does not know how to do this limit (which is 0), so it returns it unevaluated:

```
sage: lim(exp(x^2)*(1-erf(x)), x=infinity)
limit(-e^(x^2)*erf(x) + e^(x^2), x, +Infinity)
```

simplify (*f*)

Simplify the expression f.

EXAMPLES: We simplify the expression $i + x - x$.

```
sage: f = I + x - x; simplify(f)
I
```

In fact, printing f yields the same thing - i.e., the simplified form.

taylor (*f, v, a, n*)

Expands self in a truncated Taylor or Laurent series in the variable v around the point a, containing terms through $(x-a)^n$.

INPUT:

- v - variable
- a - number
- n - integer

EXAMPLES:

```
sage: var('x,k,n')
(x, k, n)
sage: taylor (sqrt (1 - k^2*sin(x)^2), x, 0, 6)
-1/720*(45*k^6 - 60*k^4 + 16*k^2)*x^6 - 1/24*(3*k^4 - 4*k^2)*x^4 - 1/2*k^2*x^2 + 1
sage: taylor ((x + 1)^n, x, 0, 4)
1/24*(n^4 - 6*n^3 + 11*n^2 - 6*n)*x^4 + 1/6*(n^3 - 3*n^2 + 2*n)*x^3 + 1/2*(n^2 - n)*x^2 + n*x +
```

3.6 A Sample Session using Sympy

In this first part, we do all of the examples in the Sympy tutorial (http://code.google.com/p/sympy/wiki/Tutorial), but using Sage instead of Sympy.

```
sage: a = Rational((1,2))
sage: a
1/2
sage: a*2
1
sage: Rational(2)^50 / Rational(10)^50
1/888178419700125232338905334472656 25
sage: 1.0/2
0.500000000000000
sage: 1/2
1/2
sage: pi^2
pi^2
sage: float(pi)
3.1415926535897931
sage: RealField(200)(pi)
3.14159265358979323846264338327950288419716939937510582097 49
sage: float(pi + exp(1))
5.85987448204883...
sage: oo != 2
True


sage: var('x y')
(x, y)
sage: x + y + x - y
2*x
sage: (x+y)^2
(x + y)^2
sage: ((x+y)^2).expand()
x^2 + 2*x*y + y^2
sage: ((x+y)^2).subs(x=1)
```

```
(y + 1)^2
sage: ((x+y)^2).subs(x=y)
4*y^2


sage: limit(sin(x)/x, x=0)
1
sage: limit(x, x=oo)
+Infinity
sage: limit((5^x + 3^x)^(1/x), x=oo)
5


sage: diff(sin(x), x)
cos(x)
sage: diff(sin(2*x), x)
2*cos(2*x)
sage: diff(tan(x), x)
tan(x)^2 + 1
sage: limit((tan(x+y) - tan(x))/y, y=0)
cos(x)^(-2)
sage: diff(sin(2*x), x, 1)
2*cos(2*x)
sage: diff(sin(2*x), x, 2)
-4*sin(2*x)
sage: diff(sin(2*x), x, 3)
-8*cos(2*x)


sage: cos(x).taylor(x,0,10)
-1/3628800*x^10 + 1/40320*x^8 - 1/720*x^6 + 1/24*x^4 - 1/2*x^2 + 1
sage: (1/cos(x)).taylor(x,0,10)
50521/3628800*x^10 + 277/8064*x^8 + 61/720*x^6 + 5/24*x^4 + 1/2*x^2 + 1


sage: matrix([[1,0], [0,1]])
[1 0]
[0 1]
sage: var('x y')
(x, y)
sage: A = matrix([[1,x], [y,1]])
sage: A
[1 x]
[y 1]
sage: A^2
[x*y + 1     2*x]
[    2*y x*y + 1]
sage: R.<x,y> = QQ[]
sage: A = matrix([[1,x], [y,1]])
sage: print A^10
[x^5*y^5 + 45*x^4*y^4 + 210*x^3*y^3 + 210*x^2*y^2 + 45*x*y + 1     10*x^5*y^4 + 120*x^4*y^3 + 252*x^
[    10*x^4*y^5 + 120*x^3*y^4 + 252*x^2*y^3 + 120*x*y^2 + 10*y x^5*y^5 + 45*x^4*y^4 + 210*x^3*y^3 + 2
sage: var('x y')
(x, y)
```

And here are some actual tests of sympy:

```
sage: from sympy import Symbol, cos, sympify, pprint
sage: from sympy.abc import x


sage: e = sympify(1)/cos(x)**3; e
cos(x)**(-3)
sage: f = e.series(x, 0, 10); f
1 + 3*x**2/2 + 11*x**4/8 + 241*x**6/240 + 8651*x**8/13440 + O(x**10)
```

And the pretty-printer:

```
sage: pprint(e)
       1
    -------
       3
    cos (x)
sage: pprint(f)
           2       4         6          8
        3*x    11*x    241*x    8651*x
    1 + ---- + ----- + ------ + ------- + O(x**10)
         2       8       240       13440
```

And the functionality to convert from sympy format to Sage format:

```
sage: e._sage_()
cos(x)^(-3)
sage: e._sage_().taylor(x._sage_(), 0, 8)
8651/13440*x^8 + 241/240*x^6 + 11/8*x^4 + 3/2*x^2 + 1
sage: f._sage_()
8651/13440*x^8 + 241/240*x^6 + 11/8*x^4 + 3/2*x^2 + 1
```

Mixing SymPy with Sage:

```
sage: import sympy
sage: sympy.sympify(var("y"))+sympy.Symbol("x")
x + y
sage: o = var("omega")
sage: s = sympy.Symbol("x")
sage: t1 = s + o
sage: t2 = o + s
sage: type(t1)
<class 'sympy.core.add.Add'>
sage: type(t2)
<type 'sage.symbolic.expression.Expression'>
sage: t1, t2
(omega + x, omega + x)
sage: e=sympy.sin(var("y"))+sage.all.cos(sympy.Symbol("x"))
sage: type(e)
<class 'sympy.core.add.Add'>
sage: e
cos(x) + sin(y)
sage: e=e._sage_()
sage: type(e)
<type 'sage.symbolic.expression.Expression'>
sage: e
sin(y) + cos(x)
sage: e = sage.all.cos(var("y")**3)**4+var("x")**2
```

```
sage: e = e._sympy_()
sage: e
x**2 + cos(y**3)**4
```

```
sage: a = sympy.Matrix([1, 2, 3])
sage: a[1]
2
```

```
sage: sympify(1.5)
1.50000000000000
sage: sympify(2)
2
sage: sympify(-2)
-2
```

3.7 Calculus Tests and Examples.

Compute the Christoffel symbol.

```
sage: var('r t theta phi')
(r, t, theta, phi)
sage: m = matrix(SR, [[(1-1/r),0,0,0],[0,-(1-1/r)^(-1),0,0],[0,0,-r^2,0],[0,0,0,-r^2*(sin(theta))^2]]
sage: print m
[          -1/r + 1                   0                   0                   0]
[                 0        1/(1/r - 1)                   0                   0]
[                 0                   0              -r^2                   0]
[                 0                   0                   0 -r^2*sin(theta)^2]
```

```
sage: def christoffel(i,j,k,vars,g):
...       s = 0
...       ginv = g^(-1)
...       for l in range(g.nrows()):
...           s = s + (1/2)*ginv[k,l]*(g[j,l].diff(vars[i])+g[i,l].diff(vars[j])-g[i,j].diff(vars[l]))
...       return s
```

```
sage: christoffel(3,3,2, [t,r,theta,phi], m)
-sin(theta)*cos(theta)
sage: X = christoffel(1,1,1,[t,r,theta,phi],m)
sage: X
 1/2/((1/r - 1)*r^2)
sage: X.rational_simplify()
 -1/2/(r^2 - r)
```

Some basic things:

```
sage: f(x,y) = x^3 + sinh(1/y)
sage: f
(x, y) |--> x^3 + sinh(1/y)
sage: f^3
(x, y) |--> (x^3 + sinh(1/y))^3
sage: (f^3).expand()
(x, y) |--> x^9 + 3*x^6*sinh(1/y) + 3*x^3*sinh(1/y)^2 + sinh(1/y)^3
```

A polynomial over a symbolic base ring:

```
sage: R = SR[x]
sage: f = R([1/sqrt(2), 1/(4*sqrt(2))])
sage: f
1/8*sqrt(2)*x + 1/2*sqrt(2)
sage: -f
-1/8*sqrt(2)*x - 1/2*sqrt(2)
sage: (-f).degree()
1
```

A big product. Notice that simplifying simplifies the product further:

```
sage: A = exp(I*pi/5)
sage: b = A*A*A*A*A*A*A*A*A*A
sage: b
1
```

We check a statement made at the beginning of Friedlander and Joshi's book on Distributions:

```
sage: f(x) = sin(x^2)
sage: g(x) = cos(x) + x^3
sage: u = f(x+t) + g(x-t)
sage: u
-(t - x)^3 + sin((t + x)^2) + cos(-t + x)
sage: u.diff(t,2) - u.diff(x,2)
0
```

Restoring variables after they have been turned into functions:

```
sage: x = function('x')
sage: type(x)
<type 'sage.symbolic.function.SFunction'>
sage: x(2/3)
x(2/3)
sage: restore('x')
sage: sin(x).variables()
(x,)
```

MATHEMATICA: Some examples of integration and differentiation taken from some Mathematica docs:

```
sage: var('x n a')
(x, n, a)
sage: diff(x^n, x)        # the output looks funny, but is correct
n*x^(n - 1)
sage: diff(x^2 * log(x+a), x)
2*x*log(a + x) + x^2/(a + x)
sage: derivative(arctan(x), x)
1/(x^2 + 1)
sage: derivative(x^n, x, 3)
(n - 2)*(n - 1)*n*x^(n - 3)
sage: derivative( function('f')(x), x)
D[0](f)(x)
sage: diff( 2*x*f(x^2), x)
4*x^2*D[0](f)(x^2) + 2*f(x^2)
sage: integrate( 1/(x^4 - a^4), x)
1/4*log(-a + x)/a^3 - 1/4*log(a + x)/a^3 - 1/2*arctan(x/a)/a^3
```

```
sage: expand(integrate(log(1-x^2), x))
x*log(-x^2 + 1) - 2*x - log(x - 1) + log(x + 1)
sage: integrate(log(1-x^2)/x, x)
log(-x^2 + 1)*log(x) + 1/2*polylog2(-x^2 + 1)
sage: integrate(exp(1-x^2),x)
1/2*sqrt(pi)*e*erf(x)
sage: integrate(sin(x^2),x)
1/8*((I - 1)*sqrt(2)*erf((1/2*I - 1/2)*sqrt(2)*x) + (I + 1)*sqrt(2)*erf((1/2*I + 1/2)*sqrt(2)*x))*sqr

sage: integrate((1-x^2)^n,x)
integrate((-x^2 + 1)^n, x)
sage: integrate(x^x,x)
integrate(x^x, x)
sage: integrate(1/(x^3+1),x)
1/3*sqrt(3)*arctan(1/3*(2*x - 1)*sqrt(3)) + 1/3*log(x + 1) - 1/6*log(x^2 - x + 1)
sage: integrate(1/(x^3+1), x, 0, 1)
1/9*pi*sqrt(3) + 1/3*log(2)


sage: forget()
sage: c = var('c')
sage: assume(c > 0)
sage: integrate(exp(-c*x^2), x, -oo, oo)
sqrt(pi)/sqrt(c)
sage: forget()
```

The following are a bunch of examples of integrals that Mathematica can do, but Sage currently can't do:

```
sage: integrate(sqrt(x + sqrt(x)), x)     # todo -- mathematica can do this
integrate(sqrt(x + sqrt(x)), x)
sage: integrate(log(x)*exp(-x^2))         # todo -- mathematica can do this
integrate(e^(-x^2)*log(x), x)
```

Todo - Mathematica can do this and gets $\pi^2/15$.

```
sage: integrate(log(1+sqrt(1+4*x)/2)/x, x, 0, 1)  # not tested
[boom!]
Integral is divergent


sage: integrate(ceil(x^2 + floor(x)), x, 0, 5)     # todo: mathematica can do this
integrate(ceil(x^2) + floor(x), x, 0, 5)
```

MAPLE: The basic differentiation and integration examples in the Maple documentation:

```
sage: diff(sin(x), x)
cos(x)
sage: diff(sin(x), y)
0
sage: diff(sin(x), x, 3)
-cos(x)
sage: diff(x*sin(cos(x)), x)
-x*sin(x)*cos(cos(x)) + sin(cos(x))
sage: diff(tan(x), x)
tan(x)^2 + 1
sage: f = function('f'); f
f
```

```
sage: diff(f(x), x)
D[0](f)(x)
sage: diff(f(x,y), x, y)
D[0, 1](f)(x, y)
sage: diff(f(x,y), x, y) - diff(f(x,y), y, x)
0
sage: g = function('g')
sage: var('x y z')
(x, y, z)
sage: diff(g(x,y,z), x,z,z)
D[0, 2, 2](g)(x, y, z)
sage: integrate(sin(x), x)
-cos(x)
sage: integrate(sin(x), x, 0, pi)
2


sage: var('a b')
(a, b)
sage: assume(b-a>0)      # annoying -- maple doesn't require this...
sage: print integrate(sin(x), x, a, b)
                               cos(a) - cos(b)
sage: forget()


sage: integrate( x/(x^3-1), x)
1/3*sqrt(3)*arctan(1/3*(2*x + 1)*sqrt(3)) + 1/3*log(x - 1) - 1/6*log(x^2 + x + 1)
sage: integrate(exp(-x^2), x)
1/2*sqrt(pi)*erf(x)
sage: integrate(exp(-x^2)*log(x), x)       # todo: maple can compute this exactly.
integrate(e^(-x^2)*log(x), x)
sage: f = exp(-x^2)*log(x)
sage: f.nintegral(x, 0, 999)
(-0.87005772672831..., 7.5584...e-10, 567, 0)
sage: integral(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)      # todo: maple can do this
integrate(1/sqrt(2*t^4 - 3*t^2 - 2), t, 2, 3)
sage: integral(integral(x*y^2, x, 0, y), y, -2, 2)
32/5
```

We verify several standard differentiation rules:

```
sage: function('f, g')
(f, g)
sage: diff(f(t)*g(t),t)
D[0](f)(t)*g(t) + f(t)*D[0](g)(t)
sage: diff(f(t)/g(t), t)
D[0](f)(t)/g(t) - f(t)*D[0](g)(t)/g(t)^2
sage: diff(f(t) + g(t), t)
D[0](f)(t) + D[0](g)(t)
sage: diff(c*f(t), t)
c*D[0](f)(t)
```

3.8 Conversion of symbolic expressions to other types

This module provides routines for converting new symbolic expressions to other types. Primarily, it provides a class `Converter` which will walk the expression tree and make calls to methods overridden by subclasses.

class AlgebraicConverter(*field*)

arithmetic(*ex, operator*)

Convert a symbolic expression to an algebraic number.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import AlgebraicConverter
sage: f = 2^(1/2)
sage: a = AlgebraicConverter(QQbar)
sage: a.arithmetic(f, f.operator())
1.414213562373095?
```

composition(*ex, operator*)

Coerce to an algebraic number.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import AlgebraicConverter
sage: a = AlgebraicConverter(QQbar)
sage: a.composition(exp(I*pi/3), exp)
0.500000000000000? + 0.866025403784439?*I
sage: a.composition(sin(pi/5), sin)
0.5877852522924731? + 0.?e-18*I
```

pyobject(*ex, obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import AlgebraicConverter
sage: a = AlgebraicConverter(QQbar)
sage: f = SR(2)
sage: a.pyobject(f, f.pyobject())
2
sage: _.parent()
Algebraic Field
```

class Converter(*use_fake_div=False*)

arithmetic(*ex, operator*)

The input to this method is a symbolic expression and the infix operator corresponding to that expression. Typically, one will convert all of the arguments and then perform the operation afterward.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: f = x + 2
sage: Converter().arithmetic(f, f.operator())
...
NotImplementedError: arithmetic
```

composition(*ex, operator*)

The input to this method is a symbolic expression and its operator. This method will get called when you have a symbolic function application.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: f = sin(2)
sage: Converter().composition(f, f.operator())
```

```
...
NotImplementedError: composition
```

derivative(*ex, operator*)

The input to this method is a symbolic expression which corresponds to a relation.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: a = function('f', x).diff(x); a
D[0](f)(x)
sage: Converter().derivative(a, a.operator())
...
NotImplementedError: derivative
```

get_fake_div(*ex*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: c = Converter(use_fake_div=True)
sage: c.get_fake_div(sin(x)/x)
FakeExpression([sin(x), x], <built-in function div>)
sage: c.get_fake_div(-1*sin(x))
FakeExpression([sin(x)], <built-in function neg>)
sage: c.get_fake_div(-x)
FakeExpression([x], <built-in function neg>)
sage: c.get_fake_div((2*x^3+2*x-1)/((x-2)*(x+1)))
FakeExpression([2*x^3 + 2*x - 1, FakeExpression([x - 2, x + 1], <built-in function mul>)], <
```

pyobject(*ex, obj*)

The input to this method is the result of calling `pyobject()` on a symbolic expression.

Note: Note that if a constant such as `pi` is encountered in the expression tree, its corresponding pyobject which is an instance of `sage.symbolic.constants.Pi` will be passed into this method. One cannot do arithmetic using such an object.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: f = SR(1)
sage: Converter().pyobject(f, f.pyobject())
...
NotImplementedError: pyobject
```

relation(*ex, operator*)

The input to this method is a symbolic expression which corresponds to a relation.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: import operator
sage: Converter().relation(x==3, operator.eq)
...
NotImplementedError: relation
sage: Converter().relation(x==3, operator.lt)
...
NotImplementedError: relation
```

symbol(*ex*)

The input to this method is a symbolic expression which corresponds to a single variable. For example, this method could be used to return a generator for a polynomial ring.

TESTS:

```
sage: from sage.symbolic.expression_conversions import Converter
sage: Converter().symbol(x)
...
NotImplementedError: symbol
```

class FakeExpression(*operands, operator*)

Pynac represents x/y as xy^{-1}. Often, tree-walkers would prefer to see divisions instead of multiplications and negative exponents. To allow for this (since Pynac internally doesn't have division at all), there is a possibility to pass use_fake_div=True; this will rewrite an Expression into a mixture of Expression and FakeExpression nodes, where the FakeExpression nodes are used to represent divisions. These nodes are intended to act sufficiently like Expression nodes that tree-walkers won't care about the difference.

operands()

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import FakeExpression
sage: import operator; x,y = var('x,y')
sage: f = FakeExpression([x, y], operator.div)
sage: f.operands()
[x, y]
```

operator()

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import FakeExpression
sage: import operator; x,y = var('x,y')
sage: f = FakeExpression([x, y], operator.div)
sage: f.operator()
<built-in function div>
```

pyobject()

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import FakeExpression
sage: import operator; x,y = var('x,y')
sage: f = FakeExpression([x, y], operator.div)
sage: f.pyobject()
...
TypeError: self must be a numeric expression
```

class FastCallableConverter(*ex, etb*)

arithmetic(*ex, operator*)

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x','y'])
sage: var('x,y')
(x, y)
sage: (x+y)._fast_callable_(etb)
add(v_0, v_1)
sage: (-x)._fast_callable_(etb)
neg(v_0)
sage: (x+y+x^2)._fast_callable_(etb)
add(add(ipow(v_0, 2), v_0), v_1)
```

TESTS:

```
sage: etb = ExpressionTreeBuilder(vars=['x'], domain=RDF)
sage: (x^7)._fast_callable_(etb)
ipow(v_0, 7)
```

composition(*ex, function*)

Given an ExpressionTreeBuilder, return an Expression representing this value.

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x','y'])
sage: x,y = var('x,y')
sage: sin(sqrt(x+y))._fast_callable_(etb)
sin(sqrt(add(v_0, v_1)))
```

pyobject(*ex, obj*)

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x'])
sage: pi._fast_callable_(etb)
pi
sage: etb = ExpressionTreeBuilder(vars=['x'], domain=RDF)
sage: pi._fast_callable_(etb)
3.14159265359
```

relation(*ex, operator*)

EXAMPLES:

```
sage: ff = fast_callable(x == 2, vars=['x'])
sage: ff(2)
0
sage: ff(4)
2
sage: ff = fast_callable(x < 2, vars=['x'])
...
NotImplementedError
```

symbol(*ex*)

Given an ExpressionTreeBuilder, return an Expression representing this value.

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x','y'])
sage: x, y, z = var('x,y,z')
sage: x._fast_callable_(etb)
v_0
sage: y._fast_callable_(etb)
v_1
sage: z._fast_callable_(etb)
...
ValueError: Variable 'z' not found
```

class **FastFloatConverter**(*ex, *vars*)

arithmetic(*ex, operator*)

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = x*x-y
```

```
sage: ff = f._fast_float_('x','y')
sage: ff(2,3)
1.0

sage: a = x + 2*y
sage: f = a._fast_float_('x', 'y')
sage: f(1,0)
1.0
sage: f(0,1)
2.0

sage: f = sqrt(x)._fast_float_('x'); f.op_list()
['load 0', 'call sqrt(1)']

sage: f = (1/2*x)._fast_float_('x'); f.op_list()
['load 0', 'push 0.5', 'mul']
```

composition(*ex, operator*)

EXAMPLES:

```
sage: f = sqrt(x)._fast_float_('x')
sage: f(2)
1.41421356237309...
sage: y = var('y')
sage: f = sqrt(x+y)._fast_float_('x', 'y')
sage: f(1,1)
1.41421356237309...

sage: f = sqrt(x+2*y)._fast_float_('x', 'y')
sage: f(2,0)
1.41421356237309...
sage: f(0,1)
1.41421356237309...
```

pyobject(*ex, obj*)

EXAMPLES:

```
sage: f = SR(2)._fast_float_()
sage: f(3)
2.0
```

relation(*ex, operator*)

EXAMPLES:

```
sage: ff = fast_float(x == 2, 'x')
sage: ff(2)
0.0
sage: ff(4)
2.0
sage: ff = fast_float(x < 2, 'x')
...
NotImplementedError
```

symbol(*ex*)

EXAMPLES:

```
sage: f = x._fast_float_('x', 'y')
sage: f(1,2)
1.0
sage: f = x._fast_float_('y', 'x')
```

```
sage: f(1,2)
2.0
```

class **InterfaceInit**(*interface*)

arithmetic(*ex, operator*)
EXAMPLES:

```
sage: import operator
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.arithmetic(x+2, operator.add)
'(x)+(2)'
```

composition(*ex, operator*)
EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.composition(sin(x), sin)
'sin(x)'
sage: m.composition(ceil(x), ceil)
'ceiling(x)'

sage: m = InterfaceInit(mathematica)
sage: m.composition(sin(x), sin)
'Sin[x]'
```

derivative(*ex, operator*)
EXAMPLES:

```
sage: import operator
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: a = function('f', x).diff(x); a
D[0](f)(x)
sage: print m.derivative(a, a.operator())
diff('f(x), x, 1)
sage: b = function('f', x).diff(x).diff(x)
sage: print m.derivative(b, b.operator())
diff('f(x), x, 2)
```

pyobject(*ex, obj*)
EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: ii = InterfaceInit(gp)
sage: f = 2+I
sage: ii.pyobject(f, f.pyobject())
'I + 2'

sage: ii.pyobject(SR(2), 2)
'2'

sage: ii.pyobject(pi, pi.pyobject())
'Pi'
```

relation(*ex, operator*)
EXAMPLES:

```
sage: import operator
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.relation(x==3, operator.eq)
'x = 3'
sage: m.relation(x==3, operator.lt)
'x < 3'
```

symbol(*ex*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import InterfaceInit
sage: m = InterfaceInit(maxima)
sage: m.symbol(x)
'x'
```

class **PolynomialConverter**(*ex, base_ring=None, ring=None*)

arithmetic(*ex, operator*)

EXAMPLES:

```
sage: import operator
sage: from sage.symbolic.expression_conversions import PolynomialConverter

sage: x, y = var('x, y')
sage: p = PolynomialConverter(x, base_ring=RR)
sage: p.arithmetic(pi+e, operator.add)
5.85987448204884
sage: p.arithmetic(x^2, operator.pow)
1.00000000000000*x^2

sage: p = PolynomialConverter(x+y, base_ring=RR)
sage: p.arithmetic(x*y+y^2, operator.add)
x*y + y^2
```

composition(*ex, operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import PolynomialConverter
sage: a = sin(2)
sage: p = PolynomialConverter(a*x, base_ring=RR)
sage: p.composition(a, a.operator())
0.909297426825682
```

pyobject(*ex, obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import PolynomialConverter
sage: p = PolynomialConverter(x, base_ring=QQ)
sage: f = SR(2)
sage: p.pyobject(f, f.pyobject())
2
sage: _.parent()
Rational Field
```

relation(*ex, op*)

EXAMPLES:

```
sage: import operator
sage: from sage.symbolic.expression_conversions import PolynomialConverter
```

```
sage: x, y = var('x, y')
sage: p = PolynomialConverter(x, base_ring=RR)

sage: p.relation(x==3, operator.eq)
1.00000000000000*x - 3.00000000000000
sage: p.relation(x==3, operator.lt)
...
ValueError: Unable to represent as a polynomial

sage: p = PolynomialConverter(x - y, base_ring=QQ)
sage: p.relation(x^2 - y^3 + 1 == x^3, operator.eq)
-x^3 - y^3 + x^2 + 1
```

symbol(*ex*)

Returns a variable in the polynomial ring.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import PolynomialConverter
sage: p = PolynomialConverter(x, base_ring=QQ)
sage: p.symbol(x)
x
sage: _.parent()
Univariate Polynomial Ring in x over Rational Field
```

class **RingConverter**(*R*, *subs_dict=None*)

arithmetic(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import RingConverter
sage: P.<z> = ZZ[]
sage: R = RingConverter(P, subs_dict={x:z})
sage: a = 2*x^2 + x + 3
sage: R(a)
2*z^2 + z + 3
```

composition(*ex*, *operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import RingConverter
sage: R = RingConverter(RIF)
sage: R(cos(2))
-0.4161468365471424?
```

pyobject(*ex*, *obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import RingConverter
sage: R = RingConverter(RIF)
sage: R(SR(5/2))
2.5000000000000000?
```

symbol(*ex*)

All symbols appearing in the expression must appear in *subs_dict* in order for the conversion to be successful.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import RingConverter
sage: R = RingConverter(RIF, subs_dict={x:2})
sage: R(x+pi)
5.141592653589794?

sage: R = RingConverter(RIF)
sage: R(x+pi)
...
TypeError
```

class SubstituteFunction (*ex, original, new*)

arithmetic (*ex, operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SubstituteFunction
sage: foo = function('foo'); bar = function('bar')
sage: s = SubstituteFunction(foo(x), foo, bar)
sage: f = x*foo(x) + pi/foo(x)
sage: s.arithmetic(f, f.operator())
x*bar(x) + pi/bar(x)
```

composition (*ex, operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SubstituteFunction
sage: foo = function('foo'); bar = function('bar')
sage: s = SubstituteFunction(foo(x), foo, bar)
sage: f = foo(x)
sage: s.composition(f, f.operator())
bar(x)
sage: f = foo(foo(x))
sage: s.composition(f, f.operator())
bar(bar(x))
sage: f = sin(foo(x))
sage: s.composition(f, f.operator())
sin(bar(x))
sage: f = foo(sin(x))
sage: s.composition(f, f.operator())
bar(sin(x))
```

derivative (*ex, operator*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SubstituteFunction
sage: foo = function('foo'); bar = function('bar')
sage: s = SubstituteFunction(foo(x), foo, bar)
sage: f = foo(x).diff(x)
sage: s.derivative(f, f.operator())
D[0](bar)(x)
```

pyobject (*ex, obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SubstituteFunction
sage: foo = function('foo'); bar = function('bar')
sage: s = SubstituteFunction(foo(x), foo, bar)
sage: f = SR(2)
sage: s.pyobject(f, f.pyobject())
```

```
2
sage: _.parent()
Symbolic Ring
```

relation(*ex, operator*)
EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SubstituteFunction
sage: foo = function('foo'); bar = function('bar')
sage: s = SubstituteFunction(foo(x), foo, bar)
sage: eq = foo(x) == x
sage: s.relation(eq, eq.operator())
bar(x) == x
```

symbol(*ex*)
EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SubstituteFunction
sage: foo = function('foo'); bar = function('bar')
sage: s = SubstituteFunction(foo(x), foo, bar)
sage: s.symbol(x)
x
```

class **SympyConverter**(*use_fake_div=False*)
Converts any expression to SymPy.

EXAMPLE:

```
sage: import sympy
sage: var('x,y')
(x, y)
sage: f = exp(x^2) - arcsin(pi+x)/y
sage: f._sympy_()
-asin(pi + x)/y + exp(x**2)
sage: _._sage_()
-arcsin(pi + x)/y + e^(x^2)

sage: sympy.sympify(x) # indirect doctest
x
```

arithmetic(*ex, operator*)
EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SympyConverter
sage: s = SympyConverter()
sage: f = x + 2
sage: s.arithmetic(f, f.operator())
2 + x
```

composition(*ex, operator*)
EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SympyConverter
sage: s = SympyConverter()
sage: f = sin(2)
sage: s.composition(f, f.operator())
sin(2)
sage: type(_)
sin
sage: f = arcsin(2)
```

```
sage: s.composition(f, f.operator())
asin(2)
```

pyobject(*ex*, *obj*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SympyConverter
sage: s = SympyConverter()
sage: f = SR(2)
sage: s.pyobject(f, f.pyobject())
2
sage: type(_)
<class 'sympy.core.numbers.Integer'>
```

symbol(*ex*)

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import SympyConverter
sage: s = SympyConverter()
sage: s.symbol(x)
x
sage: type(_)
<class 'sympy.core.symbol.Symbol'>
```

algebraic(*ex*, *field*)

Returns the symbolic expression *ex* as a element of the algebraic field *field*.

EXAMPLES:

```
sage: a = SR(5/6)
sage: AA(a)
5/6
sage: type(AA(a))
<class 'sage.rings.qqbar.AlgebraicReal'>
sage: QQbar(a)
5/6
sage: type(QQbar(a))
<class 'sage.rings.qqbar.AlgebraicNumber'>
sage: QQbar(i)
1*I
sage: AA(golden_ratio)
1.618033988749895?
sage: QQbar(golden_ratio)
1.618033988749895?
sage: QQbar(sin(pi/3))
0.866025403784439?

sage: QQbar(sqrt(2) + sqrt(8))
4.242640687119285?
sage: AA(sqrt(2)) ^ 4) == 4
True
sage: AA(-golden_ratio)
-1.618033988749895?
sage: QQbar((2*I)^(1/2))
1 + 1*I
sage: QQbar(e^(pi*I/3))
0.500000000000000? + 0.866025403784439?*I

sage: AA(x*sin(0))
```

```
0
sage: QQbar(x*sin(0))
0
```

fast_callable(*ex, etb*)

Given an ExpressionTreeBuilder *etb*, return an Expression representing the symbolic expression *ex*.

EXAMPLES:

```
sage: from sage.ext.fast_callable import ExpressionTreeBuilder
sage: etb = ExpressionTreeBuilder(vars=['x','y'])
sage: x,y = var('x,y')
sage: f = y+2*x^2
sage: f._fast_callable_(etb)
add(mul(ipow(v_0, 2), 2), v_1)

sage: f = (2*x^3+2*x-1)/((x-2)*(x+1))
sage: f._fast_callable_(etb)
div(add(add(mul(ipow(v_0, 3), 2), mul(v_0, 2)), -1), mul(add(v_0, -2), add(v_0, 1)))
```

fast_float(*ex, *vars*)

Returns an object which provides fast floating point evaluation of the symbolic expression *ex*.

See `sage.ext.fast_eval` for more information.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import fast_float
sage: f = sqrt(x+1)
sage: ff = fast_float(f, 'x')
sage: ff(1.0)
1.4142135623730951
```

polynomial(*ex, base_ring=None, ring=None*)

Returns a polynomial from the symbolic expression *ex*. Either a base ring *base_ring* or a polynomial ring *ring* can be specified for the parent of result. If just a base ring is given, then the variables of the base ring will be the variables of the expression *ex*.

EXAMPLES:

```
sage: from sage.symbolic.expression_conversions import polynomial
sage: f = x^2 + 2
sage: polynomial(f, base_ring=QQ)
x^2 + 2
sage: _.parent()
Univariate Polynomial Ring in x over Rational Field

sage: polynomial(f, ring=QQ['x,y'])
x^2 + 2
sage: _.parent()
Multivariate Polynomial Ring in x, y over Rational Field

sage: x, y = var('x, y')
sage: polynomial(x + y^2, ring=QQ['x,y'])
y^2 + x
sage: _.parent()
Multivariate Polynomial Ring in x, y over Rational Field
```

The polynomials can have arbitrary (constant) coefficients so long as they coerce into the base ring:

```
sage: polynomial(2^sin(2)*x^2 + exp(3), base_ring=RR)
1.87813065119873*x^2 + 20.0855369231877
```

3.9 Further examples from Wester's paper

These are all the problems at http://yacas.sourceforge.net/essaysmanual.html

They come from the 1994 paper "Review of CAS mathematical capabilities", by Michael Wester, who put forward 123 problems that a reasonable computer algebra system should be able to solve and tested the then current versions of various commercial CAS on this list. Sage can do most of the problems natively now, i.e., with no explicit calls to maxima or other systems.

```
sage: factorial(50)
30414093201713378043612608166064768844377641568960512000000000000
sage: factor(factorial(50))
2^47 * 3^22 * 5^12 * 7^8 * 11^4 * 13^3 * 17^2 * 19^2 * 23^2 * 29 * 31 * 37 * 41 * 43 * 47
```

```
sage: # 1/2+...+1/10 = 4861/2520
sage: sum(1/n for n in range(2,10+1)) == 4861/2520
True
```

```
sage: # Evaluate  e^(Pi*Sqrt(163)) to 50 decimal digits
sage: a = e^(pi*sqrt(163)); a
e^(pi*sqrt(163))
sage: print RealField(150)(a)
2.6253741264076874399999999999992500725971982 0e17
```

```
sage: # Evaluate the Bessel function J[2] numerically at z=1+I.
sage: bessel_J (2, 1+I)
0.0415798869439621 + 0.247397641513306*I
```

```
sage: # Obtain period of decimal fraction 1/7=0.(142857).
sage: a = 1/7
sage: print a
1/7
sage: print a.period()
6
```

```
sage: # Continued fraction of 3.1415926535
sage: a = 3.1415926535
sage: continued_fraction(a)
[3, 7, 15, 1, 292, 1, 1, 6, 2, 13, 4]
```

```
sage: # (not exactly ok) Sqrt(2*Sqrt(3)+4)=1+Sqrt(3).
sage: # The maxima backend equality checker fails this; maybe it *should*, since
sage: # the equality only holds for one choice of sign.
sage: a = sqrt(2*sqrt(3) + 4); b = 1 + sqrt(3)
sage: print float(a-b)
0.0
sage: print bool(a == b)
False
```

```
sage: # We can, of course, do this in a quadratic field
sage: k.<sqrt3> = QuadraticField(3)
sage: asqr = 2*sqrt3 + 4
sage: b = 1+sqrt3
sage: asqr == b^2
True


sage: # (not exactly ok) Sqrt(14+3*Sqrt(3+2*Sqrt(5-12*Sqrt(3-2*Sqrt(2)))))=3+Sqrt(2).
sage: a = sqrt(14+3*sqrt(3+2*sqrt(5-12*sqrt(3-2*sqrt(2)))))
sage: b = 3+sqrt(2)
sage: a, b
(sqrt(3*sqrt(2*sqrt(-12*sqrt(-2*sqrt(2) + 3) + 5) + 3) + 14), sqrt(2) + 3)
sage: bool(a==b)
False
sage: abs(float(a-b)) < 1e-10
True
sage: # 2*Infinity-3=Infinity.
sage: 2*infinity-3 == infinity
True


sage: # (YES) Standard deviation of the sample (1, 2, 3, 4, 5).
sage: v = vector(RDF, 5, [1,2,3,4,5])
sage: v.standard_deviation()
1.58113883008


sage: # (NO) Hypothesis testing with t-distribution.
sage: # (NO) Hypothesis testing with chi^2 distribution


sage: # (YES) (x^2-4)/(x^2+4*x+4)=(x-2)/(x+2).
sage: R.<x> = QQ[]
sage: (x^2-4)/(x^2+4*x+4) == (x-2)/(x+2)
True
sage: restore('x')


sage: # (YES -- Maxima doesn't consider them equal,
sage: # but Sage does additional testing to show that they are)
sage: # (Exp(x)-1)/(Exp(x/2)+1)=Exp(x/2)-1.
sage: f = (exp(x)-1)/(exp(x/2)+1)
sage: g = exp(x/2)-1
sage: f
(e^x - 1)/(e^(1/2*x) + 1)
sage: g
e^(1/2*x) - 1
sage: f(x=10.0).n(53), g(x=10.0).n(53)
(147.413159102577, 147.413159102577)
sage: bool(f == g)
True


sage: # (YES) Expand (1+x)^20, take derivative and factorize.
sage: # first do it using algebraic polys
sage: R.<x> = QQ[]
sage: f = (1+x)^20; f
x^20 + 20*x^19 + 190*x^18 + 1140*x^17 + 4845*x^16 + 15504*x^15 + 38760*x^14 + 77520*x^13 + 125970*x^1
```

```
sage: deriv = f.derivative()
sage: deriv
20*x^19 + 380*x^18 + 3420*x^17 + 19380*x^16 + 77520*x^15 + 232560*x^14 + 542640*x^13 + 1007760*x^12 +
sage: deriv.factor()
(20) * (x + 1)^19
sage: restore('x')
sage: # next do it symbolically
sage: var('y')
y
sage: f = (1+y)^20; f
(y + 1)^20
sage: g = f.expand(); g
y^20 + 20*y^19 + 190*y^18 + 1140*y^17 + 4845*y^16 + 15504*y^15 + 38760*y^14 + 77520*y^13 + 125970*y^1
sage: deriv = g.derivative(); deriv
20*y^19 + 380*y^18 + 3420*y^17 + 19380*y^16 + 77520*y^15 + 232560*y^14 + 542640*y^13 + 1007760*y^12 +
sage: deriv.factor()
20*(y + 1)^19
```

```
sage: # (YES) Factorize x^100-1.
sage: factor(x^100-1)
(x - 1)*(x + 1)*(x^2 + 1)*(x^4 - x^3 + x^2 - x + 1)*(x^4 + x^3 + x^2 + x + 1)*(x^8 - x^6 + x^4 - x^2
sage: # Also, algebraically
sage: x = polygen(QQ)
sage: factor(x^100 - 1)
(x - 1) * (x + 1) * (x^2 + 1) * (x^4 - x^3 + x^2 - x + 1) * (x^4 + x^3 + x^2 + x + 1) * (x^8 - x^6 +
sage: restore('x')
```

```
sage: # (YES) Factorize  x^4-3*x^2+1 in the field of rational numbers extended by roots of  x^2-x-1.
sage: k.< a> = NumberField(x^2 - x -1)
sage: R.< y> = k[]
sage: f = y^4 - 3*y^2 + 1
sage: f
y^4 - 3*y^2 + 1
sage: factor(f)
(y - a) * (y - a + 1) * (y + a - 1) * (y + a)
```

```
sage: # (YES) Factorize  x^4-3*x^2+1 mod 5.
sage: k.< x > = GF(5) [ ]
sage: f = x^4 - 3*x^2 + 1
sage: f.factor()
(x + 2)^2 * (x + 3)^2
sage: # Alternatively, from symbol x as follows:
sage: reset('x')
sage: f = x^4 - 3*x^2 + 1
sage: f.polynomial(GF(5)).factor()
(x + 2)^2 * (x + 3)^2
```

```
sage: # (YES) Partial fraction decomposition of (x^2+2*x+3)/(x^3+4*x^2+5*x+2)
sage: f = (x^2+2*x+3)/(x^3+4*x^2+5*x+2); f
(x^2 + 2*x + 3)/(x^3 + 4*x^2 + 5*x + 2)
```

```
sage: f.partial_fraction()
-2/(x + 1) + 2/(x + 1)^2 + 3/(x + 2)
```

```
sage: # (BUG?) Assuming  x>=y,  y>=z,  z>=x, deduce  x=z.
sage: # Maxima doesn't agree that x==z is a conclusion...
sage: forget()
sage: var('x,y,z')
(x, y, z)
sage: assume(x>=y, y>=z,z>=x)
sage: print bool(x==z)
False


sage: # (YES) Assuming x>y, y>0, deduce 2*x^2>2*y^2.
sage: forget()
sage: assume(x>y, y>0)
sage: print list(sorted(assumptions()))
[x > y, y > 0]
sage: print bool(2*x^2 > 2*y^2)
True
sage: forget()
sage: print assumptions()
[]
sage: # Solve the inequality Abs(x-1)>2.


sage: # (NO) Maxima doesn't solve inequalities:
sage: eqn = abs(x-1) > 2
sage: print eqn
                                abs(x - 1) > 2


sage: # (NO) Solve the inequality (x-1)*...*(x-5)<0.
sage: eqn = prod(x-i for i in range(1,5 +1)) < 0
sage: # but don't know how to solve
sage: eqn
(x - 5)*(x - 4)*(x - 3)*(x - 2)*(x - 1) < 0


sage: # (YES) Cos(3*x)/Cos(x)=Cos(x)^2-3*Sin(x)^2 or similar equivalent combination.
sage: f = cos(3*x)/cos(x)
sage: g = cos(x)^2 - 3*sin(x)^2
sage: h = f-g
sage: print h.trig_simplify()
                                        0


sage: # (YES) Cos(3*x)/Cos(x)=2*Cos(2*x)-1.
sage: f = cos(3*x)/cos(x)
sage: g = 2*cos(2*x) - 1
sage: h = f-g
sage: print h.trig_simplify()
                                        0
sage: # (NO) Define rewrite rules to match  Cos(3*x)/Cos(x)=Cos(x)^2-3*Sin(x)^2.
sage: # Sage has no notion of "rewrite rules".


sage: # (YES) Sqrt(997)-(997^3)^(1/6)=0
sage: a = sqrt(997) - (997^3)^(1/6)
sage: a.simplify()
0
```

```
sage: bool(a == 0)
True


sage: # (YES) Sqrt(99983)-99983^3^(1/6)=0
sage: a = sqrt(99983) - (99983^3)^(1/6)
sage: bool(a==0)
True
sage: float(a)
1.1368683772...e-13
sage: print 13*7691
99983


sage: # (YES) (2^(1/3) + 4^(1/3))^3 - 6*(2^(1/3) + 4^(1/3))-6 = 0
sage: ## same issue as above -- can only do using number fields
sage: a = (2^(1/3) + 4^(1/3))^3 - 6*(2^(1/3) + 4^(1/3)) - 6; a
(2^(1/3) + 4^(1/3))^3 - 6*2^(1/3) - 6*4^(1/3) - 6
sage: bool(a==0)
True
sage: abs(float(a)) < 1e-10
True
sage: ## but we can do it using number fields.
sage: reset('x')
sage: k.<b> = NumberField(x^3-2)
sage: a = (b  + b^2)^3 - 6*(b  + b^2) - 6
sage: print a
0


sage: # (YES) Ln(Tan(x/2+Pi/4))-ArcSinh(Tan(x))=0
sage: # Yes, in that the thing is clearly not equal to 0!
sage: f = log(tan(x/2 + pi/4)) - arcsin(tan(x))
sage: bool(f == 0)
False
sage: [float(f(x=i/10)) for i in range(1,5)]
[-0.0003367004075408...,
 -0.002777800409662...,
 -0.00989099409140...,
 -0.025411145508414...]


sage: # (YES) Numerically, the expression Ln(Tan(x/2+Pi/4))-ArcSinh(Tan(x))=0 and its derivative at
sage: g = f.derivative()
sage: abs(float(f(x=0))) < 1e-10
True
sage: abs(float(g(x=0))) < 1e-10
True
sage: g
-(tan(x)^2 + 1)/sqrt(-tan(x)^2 + 1) + 1/2*(tan(1/4*pi + 1/2*x)^2 + 1)/tan(1/4*pi + 1/2*x)


sage: # (NO?) Ln((2*Sqrt(r) + 1)/Sqrt(4*r 4*Sqrt(r) 1))=0.
sage: var('r')
r
sage: f = log( (2*sqrt(r) + 1) / sqrt(4*r  + 4*sqrt(r) +  1))
sage: f
log((2*sqrt(r) + 1)/sqrt(4*r + 4*sqrt(r) + 1))
sage: bool(f == 0)
```

```
False
sage: [abs(float(f(r=i))) < 1e-10 for i in [0.1,0.3,0.5]]
[True, True, True]


sage: # (NO, except numerically)
sage: # (4*r+4*Sqrt(r)+1)^(Sqrt(r)/(2*Sqrt(r)+1))*(2*Sqrt(r)+1)^(2*Sqrt(r)+1)^(-1)-2*Sqrt(r)-1=0, ass
sage: assume(r>0)
sage: f = (4*r+4*sqrt(r)+1)^(sqrt(r)/(2*sqrt(r)+1))*(2*sqrt(r)+1)^(2*sqrt(r)+1)^(-1)-2*sqrt(r)-1
sage: f
(2*sqrt(r) + 1)^(1/(2*sqrt(r) + 1))*(4*r + 4*sqrt(r) + 1)^(sqrt(r)/(2*sqrt(r) + 1)) - 2*sqrt(r) - 1
sage: bool(f == 0)
False
sage: [abs(float(f(r=i))) < 1e-10 for i in [0.1,0.3,0.5]]
[True, True, True]


sage: # (YES) Obtain real and imaginary parts of Ln(3+4*I).
sage: a = log(3+4*I); a
log(4*I + 3)
sage: a.real()
log(5)
sage: a.imag()
arctan(4/3)


sage: # (YES) Obtain real and imaginary parts of Tan(x+I*y)
sage: z = var('z')
sage: a = tan(z); a
tan(z)
sage: a.real()
tan(real_part(z))/(tan(real_part(z))^2*tan(imag_part(z))^2 + 1)
sage: a.imag()
tanh(imag_part(z))/(tan(real_part(z))^2*tan(imag_part(z))^2 + 1)


sage: # (YES) Simplify Ln(Exp(z)) to z for -Pi<Im(z)<=Pi.
sage: assume(-pi < imag(z))
sage: assume(imag(z) <= pi)
sage: f = log(exp(z)); f
log(e^z)
sage: f.simplify()
z
sage: forget()


sage: # (YES) Assuming Re(x)>0, Re(y)>0, deduce x^(1/n)*y^(1/n)-(x*y)^(1/n)=0.
sage: assume(real(x) > 0, real(y) > 0)
sage: n = var('n')
sage: f = x^(1/n)*y^(1/n)-(x*y)^(1/n)
sage: f.simplify()
0
sage: forget()


sage: # (YES) Transform equations, (x==2)/2+(1==1)=>x/2+1==2.
sage: eq1 = x == 2
sage: eq2 = SR(1) == SR(1)
```

```
sage: eq1/2 + eq2
1/2*x + 1 == 2


sage: # (SOMEWHAT) Solve Exp(x)=1 and get all solutions.
sage: solve(exp(x) == 1, x)
[x == 0]


sage: # (SOMEWHAT) Solve Tan(x)=1 and get all solutions.
sage: solve(tan(x) == 1, x)
[x == 1/4*pi]


sage: # (YES) Solve a degenerate 3x3 linear system.
sage: # x+y+z==6,2*x+y+2*z==10,x+3*y+z==10
sage: # First symbolically:
sage: solve([x+y+z==6, 2*x+y+2*z==10, x+3*y+z==10], x,y,z)
[[x == -r1 + 4, y == 2, z == r1]]


sage: # (YES) Invert a 2x2 symbolic matrix.
sage: # [[a,b],[1,a*b]]
sage: # Using multivariate poly ring -- much nicer
sage: R.<a,b> = QQ[]
sage: m = matrix(2,2,[a,b,  1, a*b])
sage: zz = m^(-1)
sage: print zz
[     a/(a^2 - 1)    (-1)/(a^2 - 1)]
[(-1)/(a^2*b - b)     a/(a^2*b - b)]


sage: # (YES) Compute and factor the determinant of the 4x4 Vandermonde matrix in a, b, c, d.
sage: var('a,b,c,d')
(a, b, c, d)
sage: m = matrix(SR, 4, 4, [[z^i for i in range(4)] for z in [a,b,c,d]])
sage: print m
[  1   a a^2 a^3]
[  1   b b^2 b^3]
[  1   c c^2 c^3]
[  1   d d^2 d^3]
sage: d = m.determinant()
sage: d.factor()
(c - d)*(b - d)*(b - c)*(a - d)*(a - c)*(a - b)


sage: # (YES) Compute and factor the determinant of the 4x4 Vandermonde matrix in a, b, c, d.
sage: # Do it instead in a multivariate ring
sage: R.<a,b,c,d> = QQ[]
sage: m = matrix(R, 4, 4, [[z^i for i in range(4)] for z in [a,b,c,d]])
sage: print m
[  1   a a^2 a^3]
[  1   b b^2 b^3]
[  1   c c^2 c^3]
[  1   d d^2 d^3]
sage: d = m.determinant()
sage: print d
a^3*b^2*c - a^2*b^3*c - a^3*b*c^2 + a*b^3*c^2 + a^2*b*c^3 - a*b^2*c^3 - a^3*b^2*d + a^2*b^3*d + a^3*
```

```
sage: print d.factor()
(-1) * (c - d) * (b - d) * (b - c) * (-a + b) * (a - d) * (a - c)


sage: # Find the eigenvalues of a 3x3 integer matrix.
sage: m = matrix(QQ, 3, [5,-3,-7, -2,1,2, 2,-3,-4])
sage: m.eigenspaces()
[
(3, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1  0 -1]),
(1, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1  1 -1]),
(-2, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[0 1 1])
]


sage: # OK Verify some standard limits found by L'Hopital's rule:
sage: #   Verify(Limit(x,Infinity) (1+1/x)^x, Exp(1));
sage: #   Verify(Limit(x,0) (1-Cos(x))/x^2, 1/2);
sage: limit( (1+1/x)^x, x = oo)
e
sage: limit( (1-cos(x))/(x^2), x = 1/2)
-4*cos(1/2) + 4


sage: # (OK-ish) D(x)Abs(x)
sage: #    Verify(D(x) Abs(x), Sign(x));
sage: diff(abs(x))
x/abs(x)


sage: # (NO) (Integrate(x)Abs(x))=Abs(x)*x/2
sage: integral(abs(x), x)
integrate(abs(x), x)


sage: #  (YES) Compute derivative of Abs(x), piecewise defined.
sage: #     Verify(D(x)if(x<0) (-x) else x,
sage: #        Simplify(if(x<0) -1 else 1))
Piecewise defined function with 2 parts, [[(-10, 0), -1], [(0, 10), 1]]
sage: #  (NOT really) Integrate Abs(x), piecewise defined.
sage: #      Verify(Simplify(Integrate(x)
sage: #        if(x<0) (-x) else x),
sage: #        Simplify(if(x<0) (-x^2/2) else x^2/2));
sage: f = piecewise([ [[-10,0], -x], [[0,10], x]])
sage: f.integral(definite=True)
100


sage: # (YES) Taylor series of 1/Sqrt(1-v^2/c^2) at v=0.
sage: var('v,c')
(v, c)
sage: taylor(1/sqrt(1-v^2/c^2), v, 0, 7)
1/2*v^2/c^2 + 3/8*v^4/c^4 + 5/16*v^6/c^6 + 1
```

```
sage: # (OK-ish) (Taylor expansion of Sin(x))/(Taylor expansion of Cos(x)) = (Taylor expansion of Ta
sage: #      TestYacas(Taylor(x,0,5)(Taylor(x,0,5)Sin(x))/
sage: #        (Taylor(x,0,5)Cos(x)), Taylor(x,0,5)Tan(x));
sage: f = taylor(sin(x), x, 0, 8)
sage: g = taylor(cos(x), x, 0, 8)
sage: h = taylor(tan(x), x, 0, 8)
sage: f = f.power_series(QQ)
sage: g = g.power_series(QQ)
sage: h = h.power_series(QQ)
sage: f - g*h
O(x^8)


sage: # (YES) Taylor expansion of Ln(x)^a*Exp(-b*x) at x=1.
sage: a,b = var('a,b')
sage: taylor(log(x)^a*exp(-b*x), x, 1, 3)
-1/48*(x - 1)^3*((6*b + 5)*(x - 1)^a*a^2 + (x - 1)^a*a^3 + 8*(x - 1)^a*b^3 + 2*(6*b^2 + 5*b + 3)*(x -


sage: # (YES) Taylor expansion of Ln(Sin(x)/x) at x=0.
sage: taylor(log(sin(x)/x), x, 0, 10)
-1/467775*x^10 - 1/37800*x^8 - 1/2835*x^6 - 1/180*x^4 - 1/6*x^2


sage: # (NO) Compute n-th term of the Taylor series of Ln(Sin(x)/x) at x=0.
sage: # need formal functions


sage: # (NO) Compute n-th term of the Taylor series of Exp(-x)*Sin(x) at x=0.
sage: # (Sort of, with some work)
sage: # Solve x=Sin(y)+Cos(y) for y as Taylor series in x at x=1.
sage: #      TestYacas(InverseTaylor(y,0,4) Sin(y)+Cos(y),
sage: #        (y-1)+(y-1)^2/2+2*(y-1)^3/3+(y-1)^4);
sage: #       Note that InverseTaylor does not give the series in terms of x but in terms of y which
sage: # wrong. But other CAS do the same.
sage: f = sin(y) + cos(y)
sage: g = f.taylor(y, 0, 10)
sage: h = g.power_series(QQ)
sage: k = (h - 1).reversion()
sage: print k
y + 1/2*y^2 + 2/3*y^3 + y^4 + 17/10*y^5 + 37/12*y^6 + 41/7*y^7 + 23/2*y^8 + 1667/72*y^9 + 3803/80*y^


sage: # [OK] Compute Legendre polynomials directly from Rodrigues's formula, P[n]=1/(2^n*n!) *(Deriv
sage: #      P(n,x) := Simplify( 1/(2*n)!! *
sage: #        Deriv(x,n) (x^2-1)^n );
sage: #      TestYacas(P(4,x), (35*x^4)/8+(-15*x^2)/4+3/8);
sage: P = lambda n, x: simplify(diff((x^2-1)^n,x,n) / (2^n * factorial(n)))
sage: P(4,x).expand()
35/8*x^4 - 15/4*x^2 + 3/8


sage: # (YES) Define the polynomial p=Sum(i,1,5,a[i]*x^i).
sage: # symbolically
sage: ps = sum(var('a%s'%i)*x^i for i in range(1,6)); ps
a5*x^5 + a4*x^4 + a3*x^3 + a2*x^2 + a1*x
sage: ps.parent()
Symbolic Ring
```

```
sage: # algebraically
sage: R = PolynomialRing(QQ,5,names='a')
sage: S.<x> = PolynomialRing(R)
sage: p = S(list(R.gens()))*x; p
a4*x^5 + a3*x^4 + a2*x^3 + a1*x^2 + a0*x
sage: p.parent()
Univariate Polynomial Ring in x over Multivariate Polynomial Ring in a0, a1, a2, a3, a4 over Rational
```

```
sage: # (YES) Convert the above to Horner's form.
sage: #      Verify(Horner(p, x), ((((a[5]*x+a[4])*x
sage: #        +a[3])*x+a[2])*x+a[1])*x);
sage: # We use the trick of evaluating the algebraic poly at a symbolic variable:
sage: restore('x')
sage: p(x)
((((a4*x + a3)*x + a2)*x + a1)*x + a0)*x
```

```
sage: # (NO) Convert the result of problem 127 to Fortran syntax.
sage: #      CForm(Horner(p, x));
```

```
sage: # (YES) Verify that True And False=False.
sage: (True and False) == False
True
```

```
sage: # (YES) Prove x Or Not x.
sage: for x in [True, False]:
...    print x or (not x)
True
True
```

```
sage: # (YES) Prove x Or y Or x And y=>x Or y.
sage: for x in [True, False]:
...   for y in [True, False]:
...       if x or y or x and y:
...           if not (x or y):
...              print "failed!"
```

CHAPTER

FOUR

2D GRAPHICS

4.1 2D Plotting

Sage provides extensive 2D plotting functionality. The underlying rendering is done using the matplotlib Python library.

The following graphics primitives are supported:

- arrow - an arrow from a min point to a max point.
- circle - a circle with given radius
- disk - a filled disk (i.e. a sector or wedge of a circle)
- line - a line determined by a sequence of points (this need not be straight!)
- point - a point
- text - some text
- polygon - a filled polygon

The following plotting functions are supported:

- plot - plot of a function or other Sage object (e.g., elliptic curve).
- parametric_plot
- polar_plot
- list_plot
- bar_chart
- contour_plot
- plot_vector_field
- matrix_plot
- graphics_array

The following miscellaneous Graphics functions are included:

- Graphics

- is_Graphics
- rgbcolor
- hue

Type ? after each primitive in Sage for help and examples.

EXAMPLES: We construct a plot involving several graphics objects:

```
sage: G = plot(cos, -5, 5, thickness=5, rgbcolor=(0.5,1,0.5))
sage: P = polygon([[1,2], [5,6], [5,0]], rgbcolor=(1,0,0))
sage: P   # show it
```

We draw a circle and a curve:

```
sage: circle((1,1), 1) + plot(x^2, (0,5))
```

Notice that the above circle is not round, because the aspect ratio of the coordinate system is not 1:1. The `aspect_ratio` option to show allows us to fix this:

```
sage: show(circle((1,1), 1) + plot(x^2, (0,5)), aspect_ratio=1)
```

With an aspect ratio of 2 the circle is squashed half way down (it looks twice as wide as it does tall):

```
sage: show(circle((1,1), 1) + plot(x^2, (0,5)), aspect_ratio=2)
```

Use figsize to set the actual aspect ratio of the rendered image (i.e., of the frame). For example, this image is twice as many pixels wide as it is tall:

```
sage: show(circle((1,1), 1) + plot(x^2, (0,5)), figsize=[8,4])
```

Next we construct the reflection of the above polygon about the y-axis by iterating over the list of first-coordinates of the first graphic element of P (which is the actual Polygon; note that P is a Graphics object, which consists of a single polygon):

```
sage: Q = polygon([(-x,y) for x,y in P[0]], rgbcolor=(0,0,1))
sage: Q   # show it
```

We combine together different graphics objects using "+":

```
sage: H = G + P + Q
sage: print H
Graphics object consisting of 3 graphics primitives
sage: type(H)
<class 'sage.plot.plot.Graphics'>
sage: H[1]
Polygon defined by 3 points
sage: list(H[1])
[(1.0, 2.0), (5.0, 6.0), (5.0, 0.0)]
sage: H       # show it
```

We can put text in a graph:

```
sage: L = [[cos(pi*i/100)^3,sin(pi*i/100)] for i in range(200)]
sage: p = line(L, rgbcolor=(1/4,1/8,3/4))
sage: t = text('A Bulb', (1.5, 0.25))
sage: x = text('x axis', (1.5,-0.2))
sage: y = text('y axis', (0.4,0.9))
sage: g = p+t+x+y
sage: g.show(xmin=-1.5, xmax=2, ymin=-1, ymax=1)
```

We plot the Riemann zeta function along the critical line and see the first few zeros:

```
sage: i = CDF.0      # define i this way for maximum speed.
sage: p1 = plot(lambda t: arg(zeta(0.5+t*i)), 1,27,rgbcolor=(0.8,0,0))
sage: p2 = plot(lambda t: abs(zeta(0.5+t*i)), 1,27,rgbcolor=hue(0.7))
sage: print p1 + p2
Graphics object consisting of 2 graphics primitives
sage: p1 + p2    # display it
```

Many concentric circles shrinking toward the origin:

```
sage: show(sum(circle((i,0), i, hue=sin(i/10)) for i in [10,9.9,..,0]), aspect_ratio=1)
```

Here is a pretty graph:

```
sage: g = Graphics()
sage: for i in range(60):
...     p = polygon([(i*cos(i),i*sin(i)), (0,i), (i,0)],\
...                 rgbcolor=hue(i/40+0.4), alpha=0.2)
...     g = g + p
...
sage: g.show(dpi=200, axes=False)
```

Another graph:

```
sage: x = var('x')
sage: P = plot(sin(x)/x, -4,4, rgbcolor=(0,0,1)) + \
...       plot(x*cos(x), -4,4, rgbcolor=(1,0,0)) + \
...       plot(tan(x),-4,4,rgbcolor=(0,1,0))
...
sage: P.show(ymin=-pi,ymax=pi)
```

PYX EXAMPLES: These are some examples of plots similar to some of the plots in the PyX (http://pyx.sourceforge.net) documentation:

Symbolline:

```
sage: y(x) = x*sin(x^2)
sage: v = [(x, y(x)) for x in [-3,-2.95,..,3]]
sage: show(points(v, rgbcolor=(0.2,0.6, 0.1), pointsize=30) + plot(spline(v), -3.1, 3))
```

Cycliclink:

```
sage: x = var('x')
sage: g1 = plot(cos(20*x)*exp(-2*x), 0, 1)
sage: g2 = plot(2*exp(-30*x) - exp(-3*x), 0, 1)
sage: show(graphics_array([g1, g2], 2, 1), xmin=0)
```

Pi Axis: In the PyX manual, the point of this example is to show labeling the X-axis using rational multiples of Pi. Sage currently has no support for controlling how the ticks on the x and y axes are labeled, so this is really a bad example:

```
sage: g1 = plot(sin(x), 0, 2*pi)
sage: g2 = plot(cos(x), 0, 2*pi, linestyle = "--")
sage: g1 + g2     # show their sum
```

An illustration of integration:

```
sage: f(x) = (x-3)*(x-5)*(x-7)+40
sage: P = line([(2,0),(2,f(2))], rgbcolor=(0,0,0))
sage: P += line([(8,0),(8,f(8))], rgbcolor=(0,0,0))
sage: P += polygon([(2,0),(2,f(2))] + [(x, f(x)) for x in [2,2.1,..,8]] + [(8,0),(2,0)],  rgbcolor=(
sage: P += text("$\\int_{a}^b f(x) dx$", (5, 20), fontsize=16, rgbcolor=(0,0,0))
sage: P += plot(f, 1, 8.5, thickness=3)
sage: P     # show the result
```

NUMERICAL PLOTTING:

Sage also provides 2D plotting with an interface that is a likely very familiar to people doing numerical computation. For example,

```
sage: from pylab import *
sage: t = arange(0.0, 2.0, 0.01)
sage: s = sin(2*pi*t)
sage: P = plot(t, s, linewidth=1.0)
sage: xl = xlabel('time (s)')
sage: yl = ylabel('voltage (mV)')
sage: t = title('About as simple as it gets, folks')
sage: grid(True)
sage: savefig('sage.png')
```

Since the above overwrites many Sage plotting functions, we reset the state of Sage, so that the examples below work!

```
sage: reset()
```

See http://matplotlib.sourceforge.net for complete documentation about how to use Matplotlib.

TESTS: We test dumping and loading a plot.

```
sage: p = plot(sin(x), (x, 0,2*pi))
sage: Q = loads(dumps(p))
```

AUTHORS:

- Alex Clemesha and William Stein (2006-04-10): initial version
- David Joyner: examples
- Alex Clemesha (2006-05-04) major update
- Willaim Stein (2006-05-29): fine tuning, bug fixes, better server integration
- William Stein (2006-07-01): misc polish
- Alex Clemesha (2006-09-29): added contour_plot, frame axes, misc polishing

- Robert Miller (2006-10-30): tuning, NetworkX primitive
- Alex Clemesha (2006-11-25): added plot_vector_field, matrix_plot, arrow, bar_chart, Axes class usage (see axes.py)
- Bobby Moretti and William Stein (2008-01): Change plot to specify ranges using the (varname, min, max) notation.
- William Stein (2008-01-19): raised the documentation coverage from a miserable 12 percent to a 'wopping' 35 percent, and fixed and clarified numerous small issues.

class Graphics()

The Graphics object is an empty list of graphics objects It is useful to use this object when intializing a for loop where different graphics object will be added to the empty object.

EXAMPLES:

```
sage: G = Graphics(); print G
Graphics object consisting of 0 graphics primitives
sage: c = circle((1,1), 1)
sage: G+=c; print G
Graphics object consisting of 1 graphics primitive
```

Here we make a graphic of embedded isosceles triangles, coloring each one with a different color as we go:

```
sage: h=10; c=0.4; p=0.1;
sage: G = Graphics()
sage: for x in srange(1,h+1):
...         l = [[0,x*sqrt(3)],[-x/2,-x*sqrt(3)/2],[x/2,-x*sqrt(3)/2],[0,x*sqrt(3)]]
...         G+=line(l,rgbcolor=hue(c + p*(x/h)))
sage: G.show(figsize=[5,5])
```

add_primitive(*primitive*)

Adds a primitive to this graphics object.

aspect_ratio()

Get the current aspect ratio.

OUTPUT: either None if the aspect ratio hasn't been set or a positive float

EXAMPLES:

```
sage: P = circle((1,1), 1)
sage: P.aspect_ratio() is None
True
sage: P.set_aspect_ratio(2)
sage: P.aspect_ratio()
2.0
```

axes(*show=None*)

Set whether or not the x and y axes are shown by default.

INPUT:

- `show` - bool

If called with no input, return the current axes setting.

EXAMPLES:

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
```

By default the axes are displayed.

```
sage: L.axes()
True
```

But we turn them off, and verify that they are off

```
sage: L.axes(False)
sage: L.axes()
False
```

Displaying L now shows a triangle but no axes.

```
sage: L
```

axes_color(*c=None*)

Set the axes color.

If called with no input, return the current axes_color setting.

INPUT:

- c - an rgb color 3-tuple, where each tuple entry is a float between 0 and 1

EXAMPLES: We create a line, which has like everything a default axes color of black.

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
sage: L.axes_color()
(0, 0, 0)
```

We change the axes color to red and verify the change.

```
sage: L.axes_color((1,0,0))
sage: L.axes_color()
(1.0, 0.0, 0.0)
```

When we display the plot, we'll see a blue triangle and bright red axes.

```
sage: L
```

axes_label_color(*c=None*)

Set the color of the axes labels.

The axes labels are placed at the edge of the x and y axes, and are not on by default (use the `axes_labels` command to set them; see the example below). This function just changes their color.

INPUT:

- c - an rgb 3-tuple of numbers between 0 and 1

If called with no input, return the current axes_label_color setting.

EXAMPLES: We create a plot, which by default has axes label color black.

```
sage: p = plot(sin, (-1,1))
sage: p.axes_label_color()
(0, 0, 0)
```

We change the labels to be red, and confirm this:

```
sage: p.axes_label_color((1,0,0))
sage: p.axes_label_color()
(1.0, 0.0, 0.0)
```

We set labels, since otherwise we won't see anything.

```
sage: p.axes_labels(['$x$ axis', '$y$ axis'])
```

In the plot below, notice that the labels are red:

```
sage: p
```

axes_labels(*l=None*)

Set the axes labels.

INPUT:

- `l` - (default: None) a list of two strings or None

OUTPUT: a 2-tuple of strings

If l is None, returns the current `axes_labels`, which is itself by default None. The default labels are both empty.

EXAMPLES: We create a plot and put x and y axes labels on it.

```
sage: p = plot(sin(x), (x, 0, 10))
sage: p.axes_labels(['x','y'])
sage: p.axes_labels()
('x', 'y')
```

Now when you plot p, you see x and y axes labels:

```
sage: p
```

axes_range(*xmin=None, xmax=None, ymin=None, ymax=None*)

Set the ranges of the x and y axes.

INPUT:

- `xmin, xmax, ymin, ymax` - floats

EXAMPLES:

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
sage: L.set_axes_range(-1, 20, 0, 2)
sage: d = L.get_axes_range()
sage: d['xmin'], d['xmax'], d['ymin'], d['ymax']
(-1.0, 20.0, 0.0, 2.0)
```

axes_width(*w=None*)

Set the axes width. Use this to draw a plot with really fat or really thin axes.

INPUT:

- `w` - a float

If called with no input, return the current `axes_width` setting.

EXAMPLE: We create a plot, see the default axes width (with funny Python float rounding), then reset the width to 10 (very fat).

```
sage: p = plot(cos, (-3,3))
sage: p.axes_width()
0.80000000000000004
sage: p.axes_width(10)
sage: p.axes_width()
10.0
```

Finally we plot the result, which is a graph with very fat axes.

```
sage: p
```

fontsize(*s=None*)

Set the font size of axes labels and tick marks.

INPUT:

- `s` - integer, a font size in points.

If called with no input, return the current fontsize.

EXAMPLES:

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
sage: L.fontsize()
10
sage: L.fontsize(20)
sage: L.fontsize()
20
```

All the numbers on the axes will be very large in this plot:

```
sage: L
```

get_axes_range()

Returns a dictionary of the range of the axes for this graphics object. This is fall back to the ranges in get_minmax_data() for any value which the user has not explicitly set.

> **Warning:** Changing the dictionary returned by this function does not change the axes range for this object. To do that, use the set_axes_range() method.

EXAMPLES:

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
sage: list(sorted(L.get_axes_range().items()))
[('xmax', 3.0), ('xmin', 1.0), ('ymax', 5.0), ('ymin', -4.0)]
sage: L.set_axes_range(xmin=-1)
sage: list(sorted(L.get_axes_range().items()))
[('xmax', 3.0), ('xmin', -1.0), ('ymax', 5.0), ('ymin', -4.0)]
```

get_minmax_data()

Return a dictionary whose keys give the xmin, xmax, ymin, and ymax data for this graphic.

> **Warning:** The returned dictionary is mutable, but changing it does not change the xmin/xmax/ymin/ymax data. The minmax data is a function of the primitives which make up this Graphics object. To change the range of the axes, call methods xmin(), xmax(), ymin(), ymax(), or set_axes_range().

EXAMPLES:

```
sage: g = line([(-1,1), (3,2)])
sage: list(sorted(g.get_minmax_data().items()))
[('xmax', 3.0), ('xmin', -1.0), ('ymax', 2.0), ('ymin', 1.0)]
```

Note that changing ymax doesn't change the output of get_minmax_data:

```
sage: g.ymax(10)
sage: list(sorted(g.get_minmax_data().items()))
[('xmax', 3.0), ('xmin', -1.0), ('ymax', 2.0), ('ymin', 1.0)]
```

plot(**args, **kwds*)

Draw a 2d plot this graphics object, which just returns this object since this is already a 2d graphics object.

EXAMPLES:

```
sage: S = circle((0,0), 2)
sage: S.plot() is S
True
```

plot3d(*z=0, **kwds*)

Returns an embedding of this 2D plot into the xy-plane of 3D space, as a 3D plot object. An optional parameter z can be given to specify the z-coordinate.

EXAMPLES:

```
sage: sum([plot(z*sin(x), 0, 10).plot3d(z) for z in range(6)]) #long
```

save(*filename=None, xmin=None, xmax=None, ymin=None, ymax=None, figsize=(6, 3.7082039324993699), figure=None, sub=None, savenow=True, dpi=100, axes=None, axes_labels=None, fontsize=None, frame=False, verify=True, aspect_ratio=None, gridlines=None, gridlinesstyle=None, vgridlinesstyle=None, hgridlinesstyle=None*)

Save the graphics to an image file of type: PNG, PS, EPS, SVG, SOBJ, depending on the file extension you give the filename. Extension types can be: `.png`, `.ps`, `.eps`, `.svg`, and `.sobj` (for a Sage object you can load later).

EXAMPLES:

```
sage: c = circle((1,1),1,rgbcolor=(1,0,0))
sage: c.show(xmin=-1,xmax=3,ymin=-1,ymax=3)
```

To correct the apect ratio of certain graphics, it is necessary to show with a '`figsize`' of square dimensions.

```
sage: c.show(figsize=[5,5],xmin=-1,xmax=3,ymin=-1,ymax=3)

sage: point((-1,1),pointsize=30, rgbcolor=(1,0,0))
```

set_aspect_ratio(*ratio*)

Set the aspect ratio.

INPUT:

- `ratio` - a positive real number

EXAMPLES: We create a plot of a circle, and it doesn't look quite round:

```
sage: P = circle((1,1), 1); P
```

So we set the aspect ratio and now it is round:

```
sage: P.set_aspect_ratio(1)
sage: P
```

Note that the aspect ratio is inherited upon addition (which takes the max of aspect ratios of objects whose aspect ratio has been set):

```
sage: P + circle((0,0), 0.5)            # still square
```

In the following example, both plots produce a circle that looks twice as wide as tall:

```
sage: Q = circle((0,0), 0.5); Q.set_aspect_ratio(2)
sage: P + Q
sage: Q + P
```

set_axes_range(*xmin=None, xmax=None, ymin=None, ymax=None*)

Set the ranges of the x and y axes.

INPUT:

- `xmin, xmax, ymin, ymax` - floats

EXAMPLES:

```
sage: L = line([(1,2), (3,-4), (2, 5), (1,2)])
sage: L.set_axes_range(-1, 20, 0, 2)
sage: d = L.get_axes_range()
sage: d['xmin'], d['xmax'], d['ymin'], d['ymax']
(-1.0, 20.0, 0.0, 2.0)
```

show(*xmin=None, xmax=None, ymin=None, ymax=None, figsize=(6, 3.7082039324993699), filename=None, dpi=100, axes=None, axes_labels=None, frame=False, fontsize=None, aspect_ratio=None, gridlines=None, gridlinesstyle=None, vgridlinesstyle=None, hgridlinesstyle=None, linkmode=False*)

Show this graphics image with the default image viewer.

OPTIONAL INPUT:

- `filename` - (default: None) string
- `dpi` - dots per inch
- `figsize` - [width, height]
- `aspect_ratio` - the perceived width divided by the perceived height. If the aspect ratio is set to 1, circles will look round. If it is set to 2 they will look twice as wide as they are tall. This is the aspect_ratio of the image, not of the frame that contains it. If you want to set the aspect ratio of the frame, use figsize.
- `axes` - (default: True)
- `axes_labels` - (default: None) list (or tuple) of two strings; the first is used as the label for the horizontal axis, and the second for the vertical axis.
- `fontsize` - (default: current setting – 10) positive integer; used for axes labels; if you make this very large, you may have to increase figsize to see all labels.
- `frame` - (default: False) draw a frame around the image
- `gridlines` - (default: None) can be any of the following:
  - None, False: do not add grid lines.
  - True, "automatic", "major": add grid lines at major ticks of the axes.
  - "minor": add grid at major and minor ticks.
  - [xlist,ylist]: a tuple or list containing two elements, where xlist (or ylist) can be any of the following.
    - None, False: don't add horizontal (or vertical) lines.
    - True, "automatic", "major": add horizontal (or vertical) grid lines at the major ticks of the axes.
    - "minor": add horizontal (or vertical) grid lines at major and minor ticks of axes.
    - an iterable yielding numbers n or pairs (n,opts), where n is the coordinate of the line and opt is a dictionary of MATPLOTLIB options for rendering the line.
- `gridlinesstyle, hgridlinesstyle, vgridlinesstyle` - (default: None) a dictionary of MATPLOTLIB options for the rendering of the grid lines, the horizontal grid lines or the vertical grid lines, respectively.
- **`linkmode` - (default: False) If True a string containing a link** to the produced file is returned.

EXAMPLES:

```
sage: c = circle((1,1), 1, rgbcolor=(1,0,0))
sage: c.show(xmin=-1, xmax=3, ymin=-1, ymax=3)
```

To correct the apect ratio of certain graphics, it is necessary to show with a '`figsize`' of square dimensions.

```
sage: c.show(figsize=[5,5], xmin=-1, xmax=3, ymin=-1, ymax=3)
```

You can turn off the drawing of the axes:

```
sage: show(plot(sin,-4,4), axes=False)
```

You can also label the axes:

```
sage: show(plot(sin,-4,4), axes_labels=('x','y'))
```

You can turn on the drawing of a frame around the plots:

```
sage: show(plot(sin,-4,4), frame=True)
```

Add grid lines at the major ticks of the axes.

```
sage: c = circle((0,0), 1)
sage: c.show(gridlines=True)
sage: c.show(gridlines="automatic")
sage: c.show(gridlines="major")
```

Add grid lines at the major and minor ticks of the axes.

```
sage: u,v = var('u v')
sage: f = exp(-(u^2+v^2))
sage: p = plot_vector_field(f.gradient(), (u,-2,2), (v,-2,2))
sage: p.show(gridlines="minor")
```

Add only horizontal or vertical grid lines.

```
sage: p = plot(sin,-10,20)
sage: p.show(gridlines=[None, "automatic"])
sage: p.show(gridlines=["minor", False])
```

Add grid lines at specific positions (using lists/tuples).

```
sage: x, y = var('x, y')
sage: p = implicit_plot((y^2-x^2)*(x-1)*(2*x-3)-4*(x^2+y^2-2*x)^2,              ...
sage: p.show(gridlines=[[1,0],[-1,0,1]])
```

Add grid lines at specific positions (using iterators).

```
sage: def maple_leaf(t):
...     return (100/(100+(t-pi/2)^8))*(2-sin(7*t)-cos(30*t)/2)
sage: p = polar_plot(maple_leaf, -pi/4, 3*pi/2, rgbcolor="red",plot_points=1000) #long
sage: p.show(gridlines=( [-3,-2.75,..,3], xrange(-1,5,2) )) #long
```

Add grid lines at specific positions (using functions).

```
sage: y = x^5 + 4*x^4 - 10*x^3 - 40*x^2 + 9*x + 36
sage: p = plot(y, -4.1, 1.1)
sage: xlines = lambda a,b: [z for z,m in y.roots()]
sage: p.show(gridlines=[xlines, [0]], frame=True, axes=False)
```

Change the style of all the grid lines.

```
sage: b = bar_chart([-3,5,-6,11], rgbcolor=(1,0,0))
sage: b.show(gridlines=([-1,-0.5,..,4],True),              ...     gridlinesstyle=dict(color=
```

Change the style of the horizontal or vertical grid lines separately.

```
sage: p = polar_plot(2 + 2*cos(x), 0, 2*pi, rgbcolor=hue(0.3))
sage: p.show(gridlines=True,              ...     hgridlinesstyle=dict(color="orange", linewi
```

Change the style of each grid line individually.

```
sage: x, y = var('x, y')
sage: p = implicit_plot((y^2-x^2)*(x-1)*(2*x-3)-4*(x^2+y^2-2*x)^2,              ...
sage: p.show(gridlines=(
...     [
...      (1,{"color":"red","linestyle":":"}),
...      (0,{"color":"blue","linestyle":"--"})
...     ],
...     [
...      (-1,{"rgbcolor":"red","linestyle":":"}),
...      (0,{"color":"blue","linestyle":"--"}),
```

```
...       (1,{"rgbcolor":"red","linestyle":":"}),
...      ]
...      ),
...      gridlinesstyle=dict(marker='x',rgbcolor="black"))
```

Grid lines can be added to contour plots.

```
sage: f = sin(x^2 + y^2)*cos(x)*sin(y)
sage: c = contour_plot(f, (x, -4, 4), (y, -4, 4), plot_points=100)
sage: c.show(gridlines=True, gridlinesstyle={'linestyle':':','linewidth':1, 'rgbcolor':'red'
```

Grid lines can be added to matrix plots.

```
sage: M = MatrixSpace(QQ,10).random_element()
sage: matrix_plot(M).show(gridlines=True)
```

tick_label_color(*c=None*)

Set the color of the axes tick labels.

INPUT:

- c - an rgb 3-tuple of numbers between 0 and 1

If called with no input, return the current tick_label_color setting.

EXAMPLES:

```
sage: p = plot(cos, (-3,3))
sage: p.tick_label_color()
(0, 0, 0)
sage: p.tick_label_color((1,0,0))
sage: p.tick_label_color()
(1.0, 0.0, 0.0)
sage: p
```

xmax(*xmax=None*)

EXAMPLES:

```
sage: g = line([(-1,1), (3,2)])
sage: g.xmax()
3.0
sage: g.xmax(10)
sage: g.xmax()
10.0
```

xmin(*xmin=None*)

EXAMPLES:

```
sage: g = line([(-1,1), (3,2)])
sage: g.xmin()
-1.0
sage: g.xmin(-3)
sage: g.xmin()
-3.0
```

ymax(*ymax=None*)

EXAMPLES:

```
sage: g = line([(-1,1), (3,2)])
sage: g.ymax()
2.0
sage: g.ymax(10)
sage: g.ymax()
10.0
```

ymin (*ymin=None*)

EXAMPLES:

```
sage: g = line([(-1,1), (3,2)])
sage: g.ymin()
1.0
sage: g.ymin(-3)
sage: g.ymin()
-3.0
```

class GraphicsArray (*array*)

GraphicsArray takes a (m x n) list of lists of graphics objects and plots them all on one canvas.

append (*g*)

ncols ()

nrows ()

save (*filename=None, dpi=100, figsize=(6, 3.7082039324993699), axes=None, **args*)

save the `graphics_array` to (for now) a png called 'filename'.

show (*filename=None, dpi=100, figsize=(6, 3.7082039324993699), axes=None, **args*)

Show this graphics array using the default viewer.

OPTIONAL INPUT:

- `filename` - (default: None) string
- `dpi` - dots per inch
- `figsize` - [width, height] (same for square aspect)
- `axes` - (default: True)
- `fontsize` - positive integer
- `frame` - (default: False) draw a frame around the image

EXAMPLES: This draws a graphics array with four trig plots and no axes in any of the plots.

```
sage: G = graphics_array([[plot(sin), plot(cos)], [plot(tan), plot(sec)]])
sage: G.show(axes=False)
```

adaptive_refinement (*f, p1, p2, adaptive_tolerance=0.01, adaptive_recursion=5, level=0*)

The adaptive refinement algorithm for plotting a function f. See the docstring for plot for a description of the algorithm.

INPUT:

- `f` - a function of one variable
- `p1, p2` - two points to refine between
- `adaptive_recursion` - (default: 5) how many levels of recursion to go before giving up when doing adaptive refinement. Setting this to 0 disables adaptive refinement.
- `adaptive_tolerance` - (default: 0.01) how large a relative difference should be before the adaptive refinement code considers it significant; see documentation for generate_plot_points for more information. See the documentation for plot() for more information on how the adaptive refinement algorithm works.

OUTPUT:

- `list` - a list of points to insert between p1 and p2 to get a better linear approximation between them

TESTS:

```
sage: from sage.plot.plot import adaptive_refinement
sage: adaptive_refinement(sin, (0,0), (pi,0), adaptive_tolerance=0.01, adaptive_recursion=0)
[]
sage: adaptive_refinement(sin, (0,0), (pi,0), adaptive_tolerance=0.01)
[(0.125000000000000*pi, 0.38268343236508978), (0.187500000000000*pi, 0.55557023301960218), (0.25
```

This shows that lowering adaptive_tolerance and raising adaptive_recursion both increase the number of subdivision points, though which one creates more points is heavily dependent upon the function being plotted.

```
sage: x = var('x')
sage: f(x) = sin(1/x)
sage: n1 = len(adaptive_refinement(f, (0,0), (pi,0), adaptive_tolerance=0.01)); n1
15
sage: n2 = len(adaptive_refinement(f, (0,0), (pi,0), adaptive_recursion=10, adaptive_tolerance=0
79
sage: n3 = len(adaptive_refinement(f, (0,0), (pi,0), adaptive_tolerance=0.001)); n3
26
```

adjust_figsize_for_aspect_ratio(*figsize, aspect_ratio, xmin, xmax, ymin, ymax*)

Adjust the figsize in case the user also specifies an aspect ratio.

INPUTS: figsize - a sequence of two positive real numbers aspect_ratio - a positive real number xmin, xmax, ymin, ymax - real numbers

EXAMPLES: This function is used mainly internally by plotting code so we explicitly import it:

```
sage: from sage.plot.plot import adjust_figsize_for_aspect_ratio
```

This returns (5,5), since the requested aspect ratio is 1 and the x and y ranges are the same, so that's the right size rendered image to produce a 1:1 ratio internally. 5 is used instead of 3 since the image size is always adjusted to the larger of the figsize dimensions.

```
sage: adjust_figsize_for_aspect_ratio([3,5], 1, 0, 2, 0, 2)
(5, 5)
```

Here we give a scalar figsize, which is automatically converted to the figsize `(figsize, figsize/golden_ratio)`.

```
sage: adjust_figsize_for_aspect_ratio(3, 1, 0, 2, 0, 2)
(3, 3)
```

Here we omit the aspect ratio so the figsize is just returned.

```
sage: adjust_figsize_for_aspect_ratio([5,6], None, 0, 2, 0, 2)
[5, 6]
```

Here we have an aspect ratio of 2, and since the x and y ranges are the same the returned figsize is twice as wide as tall:

```
sage: adjust_figsize_for_aspect_ratio([3,5], 2, 0, 2, 0, 2)
(5, 5/2)
```

Here the x range is rather large, so to get an aspect ratio where circles look twice as wide as they are tall, we have to shrink the y size of the image.

```
sage: adjust_figsize_for_aspect_ratio([3,5], 2, 0, 10, 0, 2)
(5, 1/2)
```

float_to_html(*r, g, b*)

This is a function to present tuples of RGB floats as HTML-happy hex for matplotlib. This may not seem necessary, but there are some odd cases where matplotlib is just plain schizophrenic – for an example, do

EXAMPLES:

```
sage: vertex_colors = {(1.0, 0.8571428571428571, 0.0): [4, 5, 6], (0.28571428571428559, 0.0, 1.0
sage: graphs.DodecahedralGraph().show(vertex_colors=vertex_colors)
```

Notice how the colors don't respect the partition at all.....

generate_plot_points(*f, xrange, plot_points=5, adaptive_tolerance=0.01, adaptive_recursion=5, randomize=True*)

Calculate plot points for a function f in the interval xrange. The adaptive refinement algorithm is also automatically invoked with a *relative* adaptive tolerance of adaptive_tolerance; see below.

INPUT:

- `f` - a function of one variable
- `p1, p2` - two points to refine between
- `plot_points` - (default: 5) the minimal number of plot points. (Note however that in any actual plot a number is passed to this, with default value 200.)
- `adaptive_recursion` - (default: 5) how many levels of recursion to go before giving up when doing adaptive refinement. Setting this to 0 disables adaptive refinement.
- `adaptive_tolerance` - (default: 0.01) how large the relative difference should be before the adaptive refinement code considers it significant. If the actual difference is greater than adaptive_tolerance*delta, where delta is the initial subinterval size for the given xrange and plot_points, then the algorithm will consider it significant.

OUTPUT:

- a list of points (x, f(x)) in the interval xrange, which aproximate the function f.

TESTS:

```
sage: from sage.plot.plot import generate_plot_points
sage: generate_plot_points(sin, (0, pi), plot_points=2, adaptive_recursion=0)
[(0.0, 0.0), (3.1415926535897931, 1.2246...e-16)]

sage: generate_plot_points(sin(x).function(x), (-pi, pi), randomize=False)
[(-3.1415926535897931, -1.2246...e-16), (-2.748893571891069,
-0.3826834323650898...), (-2.3561944901923448, -0.707106781186547...),
(-2.1598449493429825, -0.831469612302545...), (-1.9634954084936207,
-0.92387953251128674), (-1.7671458676442586, -0.98078528040323043),
(-1.5707963267948966, -1.0), (-1.3744467859455345,
-0.98078528040323043), (-1.1780972450961724, -0.92387953251128674),
(-0.98174770424681035, -0.831469612302545...), (-0.78539816339744828,
-0.707106781186547...), (-0.39269908169872414, -0.38268343236508978),
(0.0, 0.0), (0.39269908169872414, 0.38268343236508978),
(0.78539816339744828, 0.707106781186547...), (0.98174770424681035,
0.831469612302545...), (1.1780972450961724, 0.92387953251128674),
(1.3744467859455345, 0.98078528040323043), (1.5707963267948966, 1.0),
(1.7671458676442586, 0.98078528040323043), (1.9634954084936207,
0.92387953251128674), (2.1598449493429825, 0.831469612302545...),
(2.3561944901923448, 0.707106781186547...), (2.748893571891069,
0.3826834323650898...), (3.1415926535897931, 1.2246...e-16)]
```

This shows that lowering adaptive_tolerance and raising adaptive_recursion both increase the number of subdivision points. (Note that which creates more points is heavily dependent on the particular function plotted.)

```
sage: x = var('x')
sage: f(x) = sin(1/x)
sage: [len(generate_plot_points(f, (-pi, pi), plot_points=16, adaptive_tolerance=i, randomize=Fa
```

```
[97, 161, 275]

sage: [len(generate_plot_points(f, (-pi, pi), plot_points=16, adaptive_recursion=i, randomize=Fa
[97, 499, 2681]
```

graphics_array (*array, n=None, m=None*)

graphics_array take a list of lists (or tuples) of graphics objects and plots them all on one canvas (single plot).

INPUT:

- array - a list of lists or tuples
- n, m - (optional) integers - if n and m are given then the input array is flattened and turned into an n x m array, with blank graphics objects padded at the end, if necessary.

EXAMPLE: Make some plots of sin functions:

```
sage: f(x) = sin(x)
sage: g(x) = sin(2*x)
sage: h(x) = sin(4*x)
sage: p1 = plot(f,-2*pi,2*pi,rgbcolor=hue(0.5))
sage: p2 = plot(g,-2*pi,2*pi,rgbcolor=hue(0.9))
sage: p3 = parametric_plot((f,g),0,2*pi,rgbcolor=hue(0.6))
sage: p4 = parametric_plot((f,h),0,2*pi,rgbcolor=hue(1.0))
```

Now make a graphics array out of the plots; then you can type either: ga.show() or ga.save().

```
sage: graphics_array(((p1,p2),(p3,p4)))
```

Here we give only one row:

```
sage: p1 = plot(sin,-4,4)
sage: p2 = plot(cos,-4,4)
sage: g = graphics_array([p1, p2]); print g
Graphics Array of size 1 x 2
sage: g.show()
```

hue (*h, s=1, v=1*)

hue(h,s=1,v=1) where 'h' stands for hue, 's' stands for saturation, 'v' stands for value. hue returns a tuple of rgb intensities (r, g, b) All values are in the range 0 to 1.

INPUT:

- h, s, v - real numbers between 0 and 1. Note that if any are not in this range they are automatically normalized to be in this range by reducing them modulo 1.

OUTPUT: A valid RGB tuple.

EXAMPLES:

```
sage: hue(0.6)
(0.0, 0.40000000000000036, 1.0)
```

hue is an easy way of getting a broader range of colors for graphics

```
sage: plot(sin, -2, 2, rgbcolor=hue(0.6))
```

is_Graphics(*x*)

Return True if x is a Graphics object.

EXAMPLES:

```
sage: from sage.plot.plot import is_Graphics
sage: is_Graphics(1)
False
sage: is_Graphics(disk((0.0, 0.0), 1, (0, pi/2)))
True
```

list_plot(*data, plotjoined=False, **kwargs*)

`list_plot` takes a single list of data, in which case it forms a list of tuples (i, di) where i goes from 0 to $\text{len}(data) - 1$ and di is the i^{th} data value, and puts points at those tuple values.

`list_plot` also takes a list of tuples (dxi, dyi) where dxi is the i^{th} data representing the x-value, and dyi is the i^{th} y-value. If `plotjoined=True` , then a line spanning all the data is drawn instead.

EXAMPLES:

```
sage: list_plot([i^2 for i in range(5)])
```

Here are a bunch of random red points:

```
sage: r = [(random(),random()) for _ in range(20)]
sage: list_plot(r,rgbcolor=(1,0,0))
```

This gives all the random points joined in a purple line:

```
sage: list_plot(r, plotjoined=True, rgbcolor=(1,0,1))
```

If you have separate lists of x values and y values which you want to plot against each other, use the `zip` command to make a single list whose entries are pairs of (x, y) values, and feed the result into `list_plot`:

```
sage: x_coords = [cos(t)^3 for t in srange(0, 2*pi, 0.02)]
sage: y_coords = [sin(t)^3 for t in srange(0, 2*pi, 0.02)]
sage: list_plot(zip(x_coords, y_coords))
```

TESTS: We check to see that the x/y min/max data are set correctly.

```
sage: d = list_plot([(100,100), (120, 120)]).get_minmax_data()
sage: d['xmin']
100.0
sage: d['ymin']
100.0
```

minmax_data(*xdata, ydata, dict=False*)

Returns the minimums and maximums of xdata and ydata.

If dict is False, then minmax_data returns the tuple (xmin, xmax, ymin, ymax); otherwise, it returns a dictionary whose keys are 'xmin', 'xmax', 'ymin', and 'ymax' and whose values are the corresponding values.

EXAMPLES:

```
sage: from sage.plot.plot import minmax_data
sage: minmax_data([], [])
(-1, 1, -1, 1)
sage: minmax_data([-1, 2], [4, -3])
(-1, 2, -3, 4)
sage: d = minmax_data([-1, 2], [4, -3], dict=True)
```

```
sage: list(sorted(d.items()))
[('xmax', 2), ('xmin', -1), ('ymax', 4), ('ymin', -3)]
```

parametric_plot (*funcs, *args, **kwargs*)

parametric_plot takes two or three functions as a list or a tuple and makes a plot with the first function giving the x coordinates, the second function giving the y coordinates, and the third function (if present) giving the z coordinates.

In the 2d case, parametric_plot is equivalent to the plot command with the option parametric=True. In the 3d case, parametric_plot is equivalent to parametric_plot3d. See each of these functions for more help and examples.

INPUT:

- funcs - 2 or 3-tuple of functions
- other options - passed to plot or parametric_plot3d

EXAMPLES: We draw some 2d parametric plots:

```
sage: t = var('t')
sage: parametric_plot( (sin(t), sin(2*t)), (t, 0, 2*pi), rgbcolor=hue(0.6) )
sage: parametric_plot((1, t), (t, 0, 4))
```

Note that in parametric_plot, there is only fill or no fill.

```
sage: parametric_plot((t, t^2), (t, -4, 4), fill = True)
```

A filled Hypotrochoid:

```
sage: parametric_plot([cos(x) + 2 * cos(x/4), sin(x) - 2 * sin(x/4)], 0, 8*pi, fill = True)

sage: parametric_plot( (5*cos(x), 5*sin(x), x), (x,-12, 12), plot_points=150, color="red")

sage: y=var('y')
sage: parametric_plot( (5*cos(x), x*y, cos(x*y)), (x, -4,4), (y,-4,4))
```

TESTS:

```
sage: parametric_plot((x, t^2), (x, -4, 4))
...
ValueError: the number of functions and the number of free variables is not a possible combinati

sage: parametric_plot((1, x+t), (x, -4, 4))
...
ValueError: the number of functions and the number of free variables is not a possible combinati

sage: parametric_plot((-t, x+t), (x, -4, 4))
...
ValueError: the number of functions and the number of free variables is not a possible combinati

sage: parametric_plot((1, x+t, y), (x, -4, 4), (t, -4, 4))
...
ValueError: the number of functions and the number of free variables is not a possible combinati
```

plot (**args, **kwds*)

Use plot by writing

```
plot(X, ...)
```

where X is a Sage object (or list of Sage objects) that either is callable and returns numbers that can be coerced to floats, or has a plot method that returns a `GraphicPrimitive` object.

Type `plot.options` for a dictionary of the default options for plots. You can change this to change the defaults for all future plots. Use `plot.reset()` to reset to the default options.

PLOT OPTIONS:

- `plot_points` - (default: 200) the minimal number of plot points.
- `adaptive_recursion` - (default: 5) how many levels of recursion to go before giving up when doing adaptive refinement. Setting this to 0 disables adaptive refinement.
- `adaptive_tolerance` - (default: 0.01) how large a difference should be before the adaptive refinement code considers it significant. See the documentation further below for more information, starting at "the algorithm used to insert".
- `xmin` - starting x value
- `xmax` - ending x value
- `color` - an rgb-tuple (r,g,b) with each of r,g,b between 0 and 1, or a color name as a string (e.g., 'purple'), or an HTML color such as '#aaff0b'.
- `detect_poles` - (Default: False) If set to True poles are detected. If set to "show" vertical asymptotes are drawn.

APPEARANCE OPTIONS:

The following options affect the appearance of the line through the points on the graph of X (these are the same as for the line function):

INPUT:

- `alpha` - How transparent the line is
- `thickness` - How thick the line is
- `rgbcolor` - The color as an rgb tuple
- `hue` - The color given as a hue

Any MATPLOTLIB line option may also be passed in. E.g.,

- `linestyle` - The style of the line, which is one of '–' (dashed), '-.' (dash dot), '-' (solid), 'steps', ':' (dotted)
- `marker` - "'0' (tickleft), '1' (tickright), '2' (tickup), '3' (tickdown), '' (nothing), ' ' (nothing), '+' (plus), ',' (pixel), '.' (point), '1' (tri_down), '3' (tri_left), '2' (tri_up), '4' (tri_right), '<' (triangle_left), '>' (triangle_right), 'None' (nothing), 'D' (diamond), 'H' (hexagon2), '_' (hline), '^' (triangle_up), 'd' (thin_diamond), 'h' (hexagon1), 'o' (circle), 'p' (pentagon), 's' (square), 'v' (triangle_down), 'x' (x), 'l' (vline)"
- `markersize` - the size of the marker in points
- `markeredgecolor` - the markerfacecolor can be any color arg
- `markeredgewidth` - the size of the marker edge in points

FILLING OPTIONS: INPUT:

- `fill` - (Default: None) One of:
  - "axis" or True: Fill the area between the function and the x-axis.
  - "min": Fill the area between the function and its minimal value.
  - "max": Fill the area between the function and its maximal value.
  - a number c: Fill the area between the function and the horizontal line y = c.
  - a function g: Fill the area between the function that is plotted and g.

– a dictonary d (only if a list of functions are plotted): The keys of the dictionary should be integers. The value of d[i] specifies the fill options for the i-th function in the list. If d[i] == [j]: Fill the area between the i-th and the j-th function in the list. (But if d[i] == j: Fill the area between the i-th function in the list and the horizontal line y = j.)

- `fillcolor` - (default: 'automatic') The color of the fill. Either 'automatic' or a color.
- `fillalpha` - (default: 0.5) How transparent the fill is. A number between 0 and 1.

Note that this function does NOT simply sample equally spaced points between xmin and xmax. Instead it computes equally spaced points and add small perturbations to them. This reduces the possibility of, e.g., sampling sin only at multiples of 2π, which would yield a very misleading graph.

EXAMPLES: We plot the sin function:

```
sage: P = plot(sin, (0,10)); print P
Graphics object consisting of 1 graphics primitive
sage: len(P)      # number of graphics primitives
1
sage: len(P[0])   # how many points were computed (random)
225
sage: P           # render
```

```
sage: P = plot(sin, (0,10), plot_points=10); print P
Graphics object consisting of 1 graphics primitive
sage: len(P[0])   # random output
32
sage: P           # render
```

We plot with randomize=False, which makes the initial sample points evenly spaced (hence always the same). Adaptive plotting might insert other points, however, unless adaptive_recursion=0.

```
sage: p=plot(1, (x,0,3), plot_points=4, randomize=False, adaptive_recursion=0)
sage: list(p[0])
[(0.0, 1.0), (1.0, 1.0), (2.0, 1.0), (3.0, 1.0)]
```

Some colored functions:

```
sage: plot(sin, 0, 10, rgbcolor='#ff00ff')
sage: plot(sin, 0, 10, rgbcolor='purple')
```

We plot several functions together by passing a list of functions as input:

```
sage: plot([sin(n*x) for n in [1..4]], (0, pi))
```

The function $\sin(1/x)$ wiggles wildly near 0. Sage adapts to this and plots extra points near the origin.

```
sage: plot(sin(1/x), (x, -1, 1))
```

Note that the independent variable may be omitted if there is no ambiguity:

```
sage: plot(sin(1/x), (-1, 1))
```

The algorithm used to insert extra points is actually pretty simple. On the picture drawn by the lines below:

```
sage: p = plot(x^2, (-0.5, 1.4)) + line([(0,0), (1,1)], rgbcolor='green')
sage: p += line([(0.5, 0.5), (0.5, 0.5^2)], rgbcolor='purple')
sage: p += point(((0, 0), (0.5, 0.5), (0.5, 0.5^2), (1, 1)), rgbcolor='red', pointsize=20)
sage: p += text('A', (-0.05, 0.1), rgbcolor='red')
sage: p += text('B', (1.01, 1.1), rgbcolor='red')
sage: p += text('C', (0.48, 0.57), rgbcolor='red')
sage: p += text('D', (0.53, 0.18), rgbcolor='red')
sage: p.show(axes=False, xmin=-0.5, xmax=1.4, ymin=0, ymax=2)
```

You have the function (in blue) and its approximation (in green) passing through the points A and B. The algorithm finds the midpoint C of AB and computes the distance between C and D. If that distance exceeds the adaptive_tolerance threshold (*relative* to the size of the initial plot subintervals), the point D is added to the curve. If D is added to the curve, then the algorithm is applied recursively to the points A and D, and D and B. It is repeated adaptive_recursion times (5, by default).

The actual sample points are slightly randomized, so the above plots may look slightly different each time you draw them.

We draw the graph of an elliptic curve as the union of graphs of 2 functions.

```
sage: def h1(x): return abs(sqrt(x^3  - 1))
sage: def h2(x): return -abs(sqrt(x^3  - 1))
sage: P = plot([h1, h2], 1,4)
sage: P          # show the result
```

We can also directly plot the elliptic curve:

```
sage: E = EllipticCurve([0,-1])
sage: plot(E, (1, 4), rgbcolor=hue(0.6))
```

We can change the line style to one of '–' (dashed), '-.' (dash dot), '-' (solid), 'steps', ':' (dotted):

```
sage: plot(sin(x), 0, 10, linestyle='-.')
```

Sage currently ignores points that cannot be evaluated

```
sage: set_verbose(-1)
sage: plot(-x*log(x), (x,0,1))  # this works fine since the failed endpoint is just skipped.
sage: set_verbose(0)
```

This prints out a warning and plots where it can (we turn off the warning by setting the verbose mode temporarily to -1.)

```
sage: set_verbose(-1)
sage: plot(x^(1/3), (x,-1,1))
sage: set_verbose(0)
```

To plot the negative real cube root, use something like the following:

```
sage: plot(lambda x : RR(x).nth_root(3), (x,-1, 1))
```

We can detect the poles of a function:: sage: plot(gamma, (-3, 4), detect_poles = True).show(ymin = -5, ymax = 5)

We draw the Gamma-Function with its poles highlighted:: sage: plot(gamma, (-3, 4), detect_poles = 'show').show(ymin = -5, ymax = 5)

The basic options for filling a plot:

```
sage: p1 = plot(sin(x), -pi, pi, fill = 'axis')
sage: p2 = plot(sin(x), -pi, pi, fill = 'min')
sage: p3 = plot(sin(x), -pi, pi, fill = 'max')
sage: p4 = plot(sin(x), -pi, pi, fill = 0.5)
sage: graphics_array([[p1, p2], [p3, p4]]).show(frame=True, axes=False)

sage: plot([sin(x), cos(2*x)*sin(4*x)], -pi, pi, fill = {0: 1}, fillcolor = 'red', fillalpha = 1
```

A example about the growth of prime numbers:

```
sage: plot(1.13*log(x), 1, 100, fill = lambda x: nth_prime(x)/floor(x), fillcolor = 'red')
```

Fill the area between a function and its asymptote:

```
sage: f = (2*x^3+2*x-1)/((x-2)*(x+1))
sage: plot([f, 2*x+2], -7,7, fill = {0: [1]}, fillcolor='#ccc').show(ymin=-20, ymax=20)
```

Fill the area between a list of functions and the x-axis:

```
sage: def b(n): return lambda x: bessel_J(n, x)
sage: plot([b(n) for n in [1..5]], 0, 20, fill = 'axis')
```

Note that to fill between the ith and jth functions, you must use dictionary key-value pairs i:[j]; key-value pairs like i:j will fill between the ith function and the line y=j:

```
sage: def b(n): return lambda x: bessel_J(n, x) + 0.5*(n-1)
sage: plot([b(c) for c in [1..5]], 0, 40, fill = dict([(i, [i+1]) for i in [0..3]]))
sage: plot([b(c) for c in [1..5]], 0, 40, fill = dict([(i, i+1) for i in [0..3]]))
```

TESTS:

We do not randomize the endpoints:

```
sage: p = plot(x, (x,-1,1))
sage: p[0].xdata[0] == -1
True
sage: p[0].xdata[-1] == 1
True
```

We check to make sure that the x/y min/max data get set correctly when there are multiple functions.

```
sage: d = plot([sin(x), cos(x)], 100, 120).get_minmax_data()
sage: d['xmin']
100.0
sage: d['xmax']
120.0
```

We check various combinations of tuples and functions, ending with tests that lambda functions work properly with explicit variable declaration, without a tuple.

```
sage: p = plot(lambda x: x,(x,-1,1))
sage: p = plot(lambda x: x,-1,1)
sage: p = plot(x,x,-1,1)
sage: p = plot(x,-1,1)
sage: p = plot(x^2,x,-1,1)
sage: p = plot(x^2,xmin=-1,xmax=2)
sage: p = plot(lambda x: x,x,-1,1)
sage: p = plot(lambda x: x^2,x,-1,1)
```

```
sage: p = plot(lambda x: 1/x,x,-1,1)
sage: f(x) = sin(x+3)-.1*x^3
sage: p = plot(lambda x: f(x),x,-1,1)
```

We check to handle cases where the function gets evaluated at a point which causes an 'inf' or '-inf' result to be produced.

```
sage: p = plot(1/x, 0, 1)
sage: p = plot(-1/x, 0, 1)
```

Bad options now give better errors:

```
sage: P = plot(sin(1/x), (x,-1,3), foo=10)
...
RuntimeError: Error in line(): option 'foo' not valid.
```

polar_plot (*funcs, *args, **kwds*)

`polar_plot` takes a single function or a list or tuple of functions and plots them with polar coordinates in the given domain.

This function is equivalent to the plot command with the option polar=True. For more help on options, see the documentation for plot.

INPUT:

- `funcs` - a function
- other options are passed to plot

EXAMPLES:

Here is a blue 8-leaved petal:

```
sage: polar_plot(sin(5*x)^2, (x, 0, 2*pi), color='blue')
```

A red figure-8:

```
sage: polar_plot(abs(sqrt(1 - sin(x)^2)), (x, 0, 2*pi), color='red')
```

A green limacon of Pascal:

```
sage: polar_plot(2 + 2*cos(x), (x, 0, 2*pi), rgbcolor=hue(0.3))
```

Several polar plots:

```
sage: polar_plot([2*sin(x), 2*cos(x)], (x, 0, 2*pi))
```

A filled spiral:

```
sage: polar_plot(sqrt, 0, 2 * pi, fill = True)
```

Fill the area between two functions:

```
sage: polar_plot(cos(4*x) + 1.5, 0, 2*pi, fill=0.5 * cos(4*x) + 2.5, fillcolor='orange').show(as
```

Fill the area between several spirals:

```
sage: polar_plot([(1.2+k*0.2)*log(x) for k in range(6)], 1, 3 * pi, fill = {0: [1], 2: [3], 4: [
```

rainbow(*n, format='hex'*)

Given an integer n, returns a list of colors, represented in HTML hex, that changes smoothly in hue from one end of the spectrum to the other. Written in order to easily represent vertex partitions on graphs.

AUTHORS:

- Robert L. Miller

EXAMPLE:

```
sage: from sage.plot.plot import rainbow
sage: rainbow(7)
['#ff0000', '#ffda00', '#48ff00', '#00ff91', '#0091ff', '#4800ff', '#ff00da']
sage: rainbow(7, 'rgbtuple')
[(1.0, 0.0, 0.0), (1.0, 0.8571428571428571, 0.0), (0.28571428571428581, 1.0, 0.0), (0.0, 1.0, 0.
```

reshape(*v, n, m*)

setup_for_eval_on_grid(*v, xrange, yrange, plot_points*)

INPUT:

- `v` - a list of functions
- `xrange` - 2 or 3 tuple (if 3, first is a variable)
- `yrange` - 2 or 3 tuple
- `plot_points` - a positive integer

OUTPUT:

- `g` - tuple of fast callable functions
- `xstep` - step size in xdirection
- `ystep` - step size in ydirection
- `xrange` - tuple of 2 floats
- `yrange` - tuple of 2 floats

EXAMPLES:

```
sage: x,y = var('x,y')
sage: sage.plot.plot.setup_for_eval_on_grid([x^2 + y^2], (x,0,5), (y,0,pi), 11)
([<sage.ext... object at ...>],
 0.5,
 0.31415926535897931,
 (0.0, 5.0),
 (0.0, 3.1415926535897931))
```

We always plot at least two points; one at the beginning and one at the end of the ranges.

```
sage: sage.plot.plot.setup_for_eval_on_grid([x^2+y^2], (x,0,1), (y,-1,1), 1)
([<sage.ext... object at ...>],
1.0,
2.0,
(0.0, 1.0),
(-1.0, 1.0))
```

show_default(*default=None*)

Set the default for showing plots using any plot commands. If called with no arguments, returns the current default.

If this is `True` (the default) then any plot object when displayed will be displayed as an actual plot instead of text, i.e., the show command is not needed.

EXAMPLES: The default starts out as `True`:

```
sage: show_default()
True
```

We set it to `False`.

```
sage: show_default(False)
```

We see that it is `False`.

```
sage: show_default()
False
```

Now plot commands will not display their plots by default.

Turn back on default display.

```
sage: show_default(True)
```

to_float_list(*v*)

Given a list or tuple or iterable v, coerce each element of v to a float and make a list out of the result.

EXAMPLES:

```
sage: from sage.plot.plot import to_float_list
sage: to_float_list([1,1/2,3])
[1.0, 0.5, 3.0]
```

var_and_list_of_values(*v, plot_points*)

INPUT:

- `v` - (v0, v1) or (var, v0, v1); if the former return the range of values between v0 and v1 taking plot_points steps; if var is given, also return var.
- `plot_points` - integer = 2 (the endpoints)

OUTPUT:

- `var` - a variable or None
- `list` - a list of floats

EXAMPLES:

```
sage: from sage.plot.plot import var_and_list_of_values
sage: var_and_list_of_values((var('theta'), 2, 5),  5)
(theta, [2.0, 2.75, 3.5, 4.25, 5.0])
sage: var_and_list_of_values((2, 5),  5)
(None, [2.0, 2.75, 3.5, 4.25, 5.0])
sage: var_and_list_of_values((var('theta'), 2, 5),  2)
(theta, [2.0, 5.0])
sage: var_and_list_of_values((2, 5),  2)
(None, [2.0, 5.0])
```

xydata_from_point_list (*points*)

Returns two lists (xdata, ydata), each coerced to a list of floats, which correspond to the x-coordinates and the y-coordinates of the points.

The points parameter can be a list of 2-tuples or some object that yields a list of one or two numbers.

This function can potentially be very slow for large point sets.

4.2 Animated plots

EXAMPLES: We plot a circle shooting up to the right:

```
sage: a = animate([circle((i,i), 1-1/(i+1), hue=i/10) for i in srange(0,2,0.2)],
...                xmin=0,ymin=0,xmax=2,ymax=2,figsize=[2,2])
sage: a.show() # optional -- requires convert command
```

class Animation (*v, **kwds*)

Return an animation of a sequence of plots of objects.

INPUT:

- `v` - list of Sage objects. These should preferably be graphics objects, but if they aren't then plot is called on them.
- `xmin, xmax, ymin, ymax` - the ranges of the x and y axes.
- `**kwds` - all additional inputs are passed onto the rendering command. E.g., use figsize to adjust the resolution and aspect ratio.

EXAMPLES:

```
sage: a = animate([sin(x + float(k)) for k in srange(0,2*pi,0.3)],
...                xmin=0, xmax=2*pi, figsize=[2,1])
sage: a
Animation with 21 frames
sage: a[:5]
Animation with 5 frames
sage: a.show()          # optional -- requires convert command
sage: a[:5].show()      # optional
```

The `show` function takes arguments to specify the delay between frames (measured in hundredths of a second, default value 20) and the number of iterations (default value 0, which means to iterate forever). To iterate 4 times with half a second between each frame:

```
sage: a.show(delay=50, iterations=4) # optional
```

An animation of drawing a parabola:

```
sage: step = 0.1
sage: L = Graphics()
sage: v = []
sage: for i in srange(0,1,step):
...       L += line([(i,i^2),(i+step,(i+step)^2)], rgbcolor=(1,0,0), thickness=2)
...       v.append(L)
sage: a = animate(v, xmin=0, ymin=0)
sage: a.show() # optional -- requires convert command
sage: show(L)
```

TESTS: This illustrates that ticket #2066 is fixed (setting axes ranges when an endpoint is 0):

```
sage: animate([plot(sin, -1,1)], xmin=0, ymin=0)._Animation__kwds['xmin']
0
```

gif(*delay=20, savefile=None, iterations=0, show_path=False*)

Returns an animated gif composed from rendering the graphics objects in self.

This function will only work if the ImageMagick command line tools package is installed, i.e., you have the "convert" command.

INPUT:

- delay - (default: 20) delay in hundredths of a second between frames
- savefile - file that the animated gif gets saved to
- iterations - integer (default: 0); number of iterations of animation. If 0, loop forever.
- show_path - boolean (default: False); if True, print the path to the saved file

If savefile is not specified: in notebook mode, display the animation; otherwise, save it to a default file name.

EXAMPLES:

```
sage: a = animate([sin(x + float(k)) for k in srange(0,2*pi,0.7)],
...                xmin=0, xmax=2*pi, figsize=[2,1])
sage: a.gif()               # optional -- requires convert command
sage: a.gif(delay=35, iterations=3)       # optional
sage: a.gif(savefile='my_animation.gif')  # optional
sage: a.gif(savefile='my_animation.gif', show_path=True) # optional
Animation saved to .../my_animation.gif.
```

Note: If ImageMagick is not installed, you will get an error message like this:

```
/usr/local/share/sage/local/bin/sage-native-execute: 8: convert:
not found

Error: ImageMagick does not appear to be installed. Saving an
animation to a GIF file or displaying an animation requires
ImageMagick, so please install it and try again.
```

See www.imagemagick.org, for example.

AUTHORS:

- William Stein

graphics_array(*ncols=3*)

Return a graphics array with the given number of columns with plots of the frames of this animation.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: v = [E.change_ring(GF(p)).plot(pointsize=30) for p in [97, 101, 103, 107]]
sage: a = animate(v, xmin=0, ymin=0)
sage: a
Animation with 4 frames
sage: a.show()        # optional -- requires convert command

sage: g = a.graphics_array()
sage: print g
Graphics Array of size 1 x 3
sage: g.show(figsize=[4,1]) # optional

sage: g = a.graphics_array(ncols=2)
sage: print g
Graphics Array of size 2 x 2
sage: g.show('sage.png')         # optional
```

png (*dir=None*)

Return the absolute path to a temp directory that contains the rendered png's of all the images in this animation.

EXAMPLES:

```
sage: a = animate([plot(x^2 + n) for n in range(4)])
sage: d = a.png()
sage: v = os.listdir(d); v.sort(); v
['00000000.png', '00000001.png', '00000002.png', '00000003.png']
```

save (*filename=None, show_path=False*)

Save this animation into a gif or sobj file.

INPUT:

- `filename` - (default: None) name of save file
- `show_path` - boolean (default: False); if True, print the path to the saved file

If filename is None, then in notebook mode, display the animation; othewise, save the animation to a gif file. If filename ends in '.gif', save to a gif file. If filename ends in '.sobj', save to an sobj file.

In all other cases, print an error.

EXAMPLES:

```
sage: a = animate([sin(x + float(k)) for k in srange(0,2*pi,0.7)],
...                xmin=0, xmax=2*pi, figsize=[2,1])
sage: a.save()          # optional -- requires convert command
sage: a.save('wave.gif')    # optional
sage: a.save('wave.gif', show_path=True)    # optional
Animation saved to file .../wave.gif.
sage: a.save('wave0.sobj')  # optional
sage: a.save('wave1.sobj', show_path=True)  # optional
Animation saved to file wave1.sobj.
```

show (*delay=20, iterations=0*)

Show this animation.

INPUT:

- `delay` - (default: 20) delay in hundredths of a second between frames
- `iterations` - integer (default: 0); number of iterations of animation. If 0, loop forever.

Note: Currently this is done using an animated gif, though this could change in the future. This requires that the ImageMagick command line tools package be installed, i.e., that you have the `convert` command.

EXAMPLES:

```
sage: a = animate([sin(x + float(k)) for k in srange(0,2*pi,0.7)],
...                xmin=0, xmax=2*pi, figsize=[2,1])
sage: a.show()        # optional -- requires convert command
```

The preceding will loop the animation forever. If you want to show only three iterations instead:

```
sage: a.show(iterations=3)     # optional
```

To put a half-second delay between frames:

```
sage: a.show(delay=50)         # optional
```

Note: If ImageMagick is not installed, you will get an error message like this:

```
/usr/local/share/sage/local/bin/sage-native-execute: 8: convert:
not found
```

```
  Error: ImageMagick does not appear to be installed. Saving an
  animation to a GIF file or displaying an animation requires
  ImageMagick, so please install it and try again.

See www.imagemagick.org, for example.
```

CHAPTER

FIVE

3D GRAPHICS

5.1 Introduction.

EXAMPLES:

```
sage: x, y = var('x y')
sage: W = plot3d(sin(pi*((x)^2+(y)^2))/2,(x,-1,1),(y,-1,1), frame=False, color='purple', opacity=0.8)
sage: S = sphere((0,0,0),size=0.3, color='red', aspect_ratio=[1,1,1])
sage: show(W + S, figsize=8)
```

5.2 Parametric Plots

adapt_if_symbolic(*f*)

If f is symbolic find the variables u, v to substitute into f. Otherwise raise a TypeError.

This function is used internally by the plot commands for efficiency reasons only.

adapt_to_callable(*f, nargs=None*)

Tries to make every function in f into a (fast) callable function, returning a new list of functions and the expected arguments.

INPUT:

- **f – a list of functions; these can be symbolic expressions,** polynomials, etc
- nargs – number of input args to have in the output functions

OUTPUT: functions, expected arguments

parametric_plot3d(*f, urange, vrange=None, plot_points='automatic', boundary_style=None, **kwds*)

Return a parametric three-dimensional space curve or surface.

There are four ways to call this function:

- parametric_plot3d([f_x, f_y, f_z], (u_min, u_max)): f_x, f_y, f_z are three functions and $u_{\min}$ and $u_{\max}$ are real numbers
- parametric_plot3d([f_x, f_y, f_z], (u, u_min, u_max)): f_x, f_y, f_z can be viewed as functions of u
- parametric_plot3d([f_x, f_y, f_z], (u_min, u_max), (v_min, v_max)): f_x, f_y, f_z are each functions of two variables
- parametric_plot3d([f_x, f_y, f_z], (u, u_min, u_max), (v, v_min, v_max)): f_x, f_y, f_z can be viewed as functions of u and v

INPUT:

- `f` - a 3-tuple of functions or expressions
- `urange` - a 2-tuple (u_min, u_max) or a 3-tuple (u, u_min, u_max)
- `vrange` - (optional - only used for surfaces) a 2-tuple (v_min, v_max) or a 3-tuple (v, v_min, v_max)
- `plot_points` - (default: "automatic", which is 75 for curves and [40,40] for surfaces) initial number of sample points in each parameter; an integer for a curve, and a pair of integers for a surface.
- `boundary_style` - (default: None, no boundary) a dict that describes how to draw the boundaries of regions by giving options that are passed to the line3d command.
- `mesh` - bool (default: False) whether to display mesh grid lines
- `dots` - bool (default: False) whether to display dots at mesh grid points

Note:

1. By default for a curve any points where f_x, f_y, or f_z do not evaluate to a real number are skipped.
2. Currently for a surface f_x, f_y, and f_z have to be defined everywhere. This will change.
3. mesh and dots are not supported when using the Tachyon raytracer renderer.

EXAMPLES: We demonstrate each of the four ways to call this function.

1. A space curve defined by three functions of 1 variable:

```
sage: parametric_plot3d( (sin, cos, lambda u: u/10), (0, 20))
```

   Note above the lambda function, which creates a callable Python function that sends u to $u/10$.

2. Next we draw the same plot as above, but using symbolic functions:

```
sage: u = var('u')
sage: parametric_plot3d( (sin(u), cos(u), u/10), (u, 0, 20))
```

3. We draw a parametric surface using 3 Python functions (defined using lambda):

```
sage: f = (lambda u,v: cos(u), lambda u,v: sin(u)+cos(v), lambda u,v: sin(v))
sage: parametric_plot3d(f, (0, 2*pi), (-pi, pi))
```

4. The surface, but with a mesh:

```
sage: u, v = var('u,v')
sage: parametric_plot3d((cos(u), sin(u) + cos(v), sin(v)), (u, 0, 2*pi), (v, -pi, pi), mesh=
```

5. The same surface, but where the defining functions are symbolic:

```
sage: u, v = var('u,v')
sage: parametric_plot3d((cos(u), sin(u) + cos(v), sin(v)), (u, 0, 2*pi), (v, -pi, pi))
```

   We increase the number of plot points, and make the surface green and transparent:

```
sage: parametric_plot3d((cos(u), sin(u) + cos(v), sin(v)), (u, 0, 2*pi), (v, -pi, pi), color
```

We call the space curve function but with polynomials instead of symbolic variables.

```
sage: R.<t> = RDF[]
sage: parametric_plot3d( (t, t^2, t^3), (t, 0, 3) )
```

Next we plot the same curve, but because we use (0, 3) instead of (t, 0, 3), each polynomial is viewed as a callable function of one variable:

```
sage: parametric_plot3d( (t, t^2, t^3), (0, 3) )
```

We do a plot but mix a symbolic input, and an integer:

```
sage: t = var('t')
sage: parametric_plot3d( (1, sin(t), cos(t)), (t, 0, 3) )
```

We specify a boundary style to show us the values of the function at its extrema:

```
sage: u, v = var('u,v')
sage: parametric_plot3d((cos(u), sin(u) + cos(v), sin(v)), (u, 0, pi), (v, 0, pi), \
...                     boundary_style={"color": "black", "thickness": 2})
```

Any options you would normally use to specify the appearance of a curve are valid as entries in the boundary_style dict.

MANY MORE EXAMPLES:

We plot two interlinked tori:

```
sage: u, v = var('u,v')
sage: f1 = (4+(3+cos(v))*sin(u), 4+(3+cos(v))*cos(u), 4+sin(v))
sage: f2 = (8+(3+cos(v))*cos(u), 3+sin(v), 4+(3+cos(v))*sin(u))
sage: p1 = parametric_plot3d(f1, (u,0,2*pi), (v,0,2*pi), texture="red")
sage: p2 = parametric_plot3d(f2, (u,0,2*pi), (v,0,2*pi), texture="blue")
sage: p1 + p2
```

A cylindrical Star of David:

```
sage: u,v = var('u v')
sage: f_x = cos(u)*cos(v)*(abs(cos(3*v/4))^500 + abs(sin(3*v/4))^500)^(-1/260)*(abs(cos(4*u/4))^
sage: f_y = cos(u)*sin(v)*(abs(cos(3*v/4))^500 + abs(sin(3*v/4))^500)^(-1/260)*(abs(cos(4*u/4))^
sage: f_z = sin(u)*(abs(cos(4*u/4))^200 + abs(sin(4*u/4))^200)^(-1/200)
sage: parametric_plot3d([f_x, f_y, f_z], (u, -pi, pi), (v, 0, 2*pi))
```

Double heart:

```
sage: u, v = var('u,v')
sage: f_x = ( abs(v) - abs(u) - abs(tanh((1/sqrt(2))*u)/(1/sqrt(2))) + abs(tanh((1/sqrt(2))*v)/(
sage: f_y = ( abs(v) - abs(u) - abs(tanh((1/sqrt(2))*u)/(1/sqrt(2))) - abs(tanh((1/sqrt(2))*v)/(
sage: f_z = sin(u)*(abs(cos(4*u/4))^1 + abs(sin(4*u/4))^1)^(-1/1)
sage: parametric_plot3d([f_x, f_y, f_z], (u, 0, pi), (v, -pi, pi))
```

Heart:

```
sage: u, v = var('u,v')
sage: f_x = cos(u)*(4*sqrt(1-v^2)*sin(abs(u))^abs(u))
sage: f_y = sin(u) *(4*sqrt(1-v^2)*sin(abs(u))^abs(u))
sage: f_z = v
sage: parametric_plot3d([f_x, f_y, f_z], (u, -pi, pi), (v, -1, 1), frame=False, color="red")
```

Green bowtie:

```
sage: u, v = var('u,v')
sage: f_x = sin(u) / (sqrt(2) + sin(v))
sage: f_y = sin(u) / (sqrt(2) + cos(v))
sage: f_z = cos(u) / (1 + sqrt(2))
sage: parametric_plot3d([f_x, f_y, f_z], (u, -pi, pi), (v, -pi, pi), frame=False, color="green")
```

Boy's surface http://en.wikipedia.org/wiki/Boy's_surface

```
sage: u, v = var('u,v')
sage: fx = 2/3* (cos(u)* cos(2*v) + sqrt(2)* sin(u)* cos(v))* cos(u) / (sqrt(2) - sin(2*u)* sin(
sage: fy = 2/3* (cos(u)* sin(2*v) - sqrt(2)* sin(u)* sin(v))* cos(u) / (sqrt(2) - sin(2*u)* sin(
sage: fz = sqrt(2)* cos(u)* cos(u) / (sqrt(2) - sin(2*u)* sin(3*v))
sage: parametric_plot3d([fx, fy, fz], (u, -2*pi, 2*pi), (v, 0, pi), plot_points = [90,90], frame
```

Maeder's_Owl (pretty but can't find an internet reference):

```
sage: u, v = var('u,v')
sage: fx = v *cos(u) - 0.5* v^2 * cos(2* u)
sage: fy = -v *sin(u) - 0.5* v^2 * sin(2* u)
sage: fz = 4 *v^1.5 * cos(3 *u / 2) / 3
sage: parametric_plot3d([fx, fy, fz], (u, -2*pi, 2*pi), (v, 0, 1),plot_points = [90,90], frame=F
```

Bracelet:

```
sage: u, v = var('u,v')
sage: fx = (2 + 0.2*sin(2*pi*u))*sin(pi*v)
sage: fy = 0.2*cos(2*pi*u) *3*cos(2*pi*v)
sage: fz = (2 + 0.2*sin(2*pi*u))*cos(pi*v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, pi/2), (v, 0, 3*pi/4), frame=False, color="gray")
```

Green goblet

```
sage: u, v = var('u,v')
sage: fx = cos(u)*cos(2*v)
sage: fy = sin(u)*cos(2*v)
sage: fz = sin(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, pi), frame=False, color="green")
```

Funny folded surface - with square projection:

```
sage: u, v = var('u,v')
sage: fx = cos(u)*sin(2*v)
sage: fy = sin(u)*cos(2*v)
sage: fz = sin(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="green")
```

Surface of revolution of figure 8:

```
sage: u, v = var('u,v')
sage: fx = cos(u)*sin(2*v)
sage: fy = sin(u)*sin(2*v)
sage: fz = sin(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="green")
```

Yellow Whitney's umbrella http://en.wikipedia.org/wiki/Whitney_umbrella:

```
sage: u, v = var('u,v')
sage: fx = u*v
sage: fy = u
sage: fz = v^2
sage: parametric_plot3d([fx, fy, fz], (u, -1, 1), (v, -1, 1), frame=False, color="yellow")
```

Cross cap http://en.wikipedia.org/wiki/Cross-cap:

```
sage: u, v = var('u,v')
sage: fx = (1+cos(v))*cos(u)
sage: fy = (1+cos(v))*sin(u)
sage: fz = -tanh((2/3)*(u-pi))*sin(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="red")
```

Twisted torus:

```
sage: u, v = var('u,v')
sage: fx = (3+sin(v)+cos(u))*cos(2*v)
sage: fy = (3+sin(v)+cos(u))*sin(2*v)
sage: fz = sin(u)+2*cos(v)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="red")
```

Four intersecting discs:

```
sage: u, v = var('u,v')
sage: fx = v *cos(u) -0.5*v^2*cos(2*u)
sage: fy = -v*sin(u) -0.5*v^2*sin(2*u)
sage: fz = 4* v^1.5 *cos(3* u / 2) / 3
sage: parametric_plot3d([fx, fy, fz], (u, 0, 4*pi), (v, 0,2*pi), frame=False, color="red", opaci
```

Steiner surface/Roman's surface (see http://en.wikipedia.org/wiki/Roman_surface and http://en.wikipedia.org/wiki/Steiner_surface):

```
sage: u, v = var('u,v')
sage: fx = (sin(2 * u) * cos(v) * cos(v))
sage: fy = (sin(u) * sin(2 * v))
sage: fz = (cos(u) * sin(2 * v))
sage: parametric_plot3d([fx, fy, fz], (u, -pi/2, pi/2), (v, -pi/2,pi/2), frame=False, color="red
```

Klein bottle? (see http://en.wikipedia.org/wiki/Klein_bottle):

```
sage: u, v = var('u,v')
sage: fx = (3*(1+sin(v)) + 2*(1-cos(v)/2)*cos(u))*cos(v)
sage: fy = (4+2*(1-cos(v)/2)*cos(u))*sin(v)
sage: fz = -2*(1-cos(v)/2) * sin(u)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="green")
```

A Figure 8 embedding of the Klein bottle (see http://en.wikipedia.org/wiki/Klein_bottle):

```
sage: u, v = var('u,v')
sage: fx = (2 + cos(v/2)* sin(u) - sin(v/2)* sin(2 *u))* cos(v)
sage: fy = (2 + cos(v/2)* sin(u) - sin(v/2)* sin(2 *u))* sin(v)
sage: fz = sin(v/2)* sin(u) + cos(v/2) *sin(2* u)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0, 2*pi), frame=False, color="red")
```

Enneper's surface (see http://en.wikipedia.org/wiki/Enneper_surface):

```
sage: u, v = var('u,v')
sage: fx = u -u^3/3  + u*v^2
sage: fy = v -v^3/3  + v*u^2
sage: fz = u^2 - v^2
sage: parametric_plot3d([fx, fy, fz], (u, -2, 2), (v, -2, 2), frame=False, color="red")
```

Henneberg's surface (see http://xahlee.org/surface/gallery_m.html)

```
sage: u, v = var('u,v')
sage: fx = 2*sinh(u)*cos(v) -(2/3)*sinh(3*u)*cos(3*v)
sage: fy = 2*sinh(u)*sin(v) +(2/3)*sinh(3*u)*sin(3*v)
sage: fz = 2*cosh(2*u)*cos(2*v)
sage: parametric_plot3d([fx, fy, fz], (u, -1, 1), (v, -pi/2, pi/2), frame=False, color="red")
```

Dini's spiral

```
sage: u, v = var('u,v')
sage: fx = cos(u)*sin(v)
sage: fy = sin(u)*sin(v)
sage: fz = (cos(v)+log(tan(v/2))) + 0.2*u
sage: parametric_plot3d([fx, fy, fz], (u, 0, 12.4), (v, 0.1, 2),frame=False, color="red")
```

Catalan's surface (see http://xahlee.org/surface/catalan/catalan.html):

```
sage: u, v = var('u,v')
sage: fx = u-sin(u)*cosh(v)
sage: fy = 1-cos(u)*cosh(v)
sage: fz = 4*sin(1/2*u)*sinh(v/2)
sage: parametric_plot3d([fx, fy, fz], (u, -pi, 3*pi), (v, -2, 2), frame=False, color="red")
```

A Conchoid:

```
sage: u, v = var('u,v')
sage: k = 1.2; k_2 = 1.2; a = 1.5
sage: f = (k^u*(1+cos(v))*cos(u), k^u*(1+cos(v))*sin(u), k^u*sin(v)-a*k_2^u)
sage: parametric_plot3d(f, (u,0,6*pi), (v,0,2*pi), plot_points=[40,40], texture=(0,0.5,0))
```

A Mobius strip:

```
sage: u,v = var("u,v")
sage: parametric_plot3d([cos(u)*(1+v*cos(u/2)), sin(u)*(1+v*cos(u/2)), 0.2*v*sin(u/2)], (u,0, 4*
```

A Twisted Ribon

```
sage: u, v = var('u,v')
sage: parametric_plot3d([3*sin(u)*cos(v), 3*sin(u)*sin(v), cos(v)], (u,0, 2*pi), (v, 0, pi),plot
```

An Ellipsoid:

```
sage: u, v = var('u,v')
sage: parametric_plot3d([3*sin(u)*cos(v), 2*sin(u)*sin(v), cos(u)], (u,0, 2*pi), (v, 0, 2*pi),pl
```

A Cone:

```
sage: u, v = var('u,v')
sage: parametric_plot3d([u*cos(v), u*sin(v), u], (u, -1, 1), (v, 0, 2*pi+0.5), plot_points=[50,5
```

A Paraboloid:

```
sage: u, v = var('u,v')
sage: parametric_plot3d([u*cos(v), u*sin(v), u^2], (u, 0, 1), (v, 0, 2*pi+0.4), plot_points=[50,
```

A Hyperboloid:

```
sage: u, v = var('u,v')
sage: plot3d(u^2-v^2, (u, -1, 1), (v, -1, 1), plot_points=[50,50])
```

A weird looking surface - like a Mobius band but also an O:

```
sage: u, v = var('u,v')
sage: parametric_plot3d([sin(u)*cos(u)*log(u^2)*sin(v), (u^2)^(1/6)*(cos(u)^2)^(1/4)*cos(v), sin
```

A heart, but not a cardioid (for my wife):

```
sage: u, v = var('u,v')
sage: p1 = parametric_plot3d([sin(u)*cos(u)*log(u^2)*v*(1-v)/2, ((u^6)^(1/20)*(cos(u)^2)^(1/4)-1
sage: p2 = parametric_plot3d([-sin(u)*cos(u)*log(u^2)*v*(1-v)/2, ((u^6)^(1/20)*(cos(u)^2)^(1/4)-
sage: show(p1+p2, frame=False)
```

A Hyperhelicoidal:

```
sage: u = var("u")
sage: v = var("v")
sage: fx = (sinh(v)*cos(3*u))/(1+cosh(u)*cosh(v))
sage: fy = (sinh(v)*sin(3*u))/(1+cosh(u)*cosh(v))
sage: fz = (cosh(v)*sinh(u))/(1+cosh(u)*cosh(v))
sage: parametric_plot3d([fx, fy, fz], (u, -pi, pi), (v, -pi, pi), plot_points = [50,50], frame=F
```

A Helicoid (lines through a helix, http://en.wikipedia.org/wiki/Helix):

```
sage: u, v = var('u,v')
sage: fx = sinh(v)*sin(u)
sage: fy = -sinh(v)*cos(u)
sage: fz = 3*u
sage: parametric_plot3d([fx, fy, fz], (u, -pi, pi), (v, -pi, pi), plot_points = [50,50], frame=F
```

Kuen's surface (http://www.math.umd.edu/research/bianchi/Gifccsurfs/ccsurfs.html):

```
sage: fx = (2*(cos(u) + u*sin(u))*sin(v))/(1+ u^2*sin(v)^2)
sage: fy = (2*(sin(u) - u*cos(u))*sin(v))/(1+ u^2*sin(v)^2)
sage: fz = log(tan(1/2 *v)) + (2*cos(v))/(1+ u^2*sin(v)^2)
sage: parametric_plot3d([fx, fy, fz], (u, 0, 2*pi), (v, 0.01, pi-0.01), plot_points = [50,50], f
```

A 5-pointed star:

```
sage: fx = cos(u)*cos(v)*(abs(cos(1*u/4))^0.5 + abs(sin(1*u/4))^0.5)^(-1/0.3)*(abs(cos(5*v/4))^1
sage: fy = cos(u)*sin(v)*(abs(cos(1*u/4))^0.5 + abs(sin(1*u/4))^0.5)^(-1/0.3)*(abs(cos(5*v/4))^1
sage: fz = sin(u)*(abs(cos(1*u/4))^0.5 + abs(sin(1*u/4))^0.5)^(-1/0.3)
sage: parametric_plot3d([fx, fy, fz], (u, -pi/2, pi/2), (v, 0, 2*pi), plot_points = [50,50], fra
```

A cool self-intersecting surface (Eppener surface?):

```
sage: fx = u - u^3/3 + u*v^2
sage: fy = v - v^3/3 + v*u^2
sage: fz = u^2 - v^2
sage: parametric_plot3d([fx, fy, fz], (u, -25, 25), (v, -25, 25), plot_points = [50,50], frame=F
```

The breather surface (http://en.wikipedia.org/wiki/Breather_surface):

```
sage: fx = (2*sqrt(0.84)*cosh(0.4*u)*(-(sqrt(0.84)*cos(v)*cos(sqrt(0.84)*v)) - sin(v)*sin(sqrt(0
sage: fy = (2*sqrt(0.84)*cosh(0.4*u)*(-(sqrt(0.84)*sin(v)*cos(sqrt(0.84)*v)) + cos(v)*sin(sqrt(0
sage: fz = -u + (2*0.84*cosh(0.4*u)*sinh(0.4*u))/(0.4*((sqrt(0.84)*cosh(0.4*u))^2 + (0.4*sin(sqr
sage: parametric_plot3d([fx, fy, fz], (u, -13.2, 13.2), (v, -37.4, 37.4), plot_points = [90,90],
```

TESTS:

```
sage: u, v = var('u,v')
sage: plot3d(u^2-v^2, (u, -1, 1), (u, -1, 1))
...
ValueError: plot variables should be distinct, but both are u.
```

From Trac #2858:

```
sage: parametric_plot3d((u,-u,v), (u,-10,10),(v,-10,10))
sage: f(u)=u; g(v)=v^2; parametric_plot3d((g,f,f), (-10,10),(-10,10))
```

From Trac #5368:

```
sage: x, y = var('x,y')
sage: plot3d(x*y^2 - sin(x), (x,-1,1), (y,-1,1))
```

parametric_plot3d_curve (*f, urange, plot_points, **kwds*)
This function is used internally by the `parametric_plot3d` command.

parametric_plot3d_surface (*f, urange, vrange, plot_points, boundary_style, **kwds*)
This function is used internally by the `parametric_plot3d` command.

5.3 Implicit Plots

implicit_plot3d (*f, xrange, yrange, zrange, **kwds*)
Plots an isosurface of a function.

INPUT:

- `f` - function
- `xrange` - a 2-tuple (x_min, x_max) or a 3-tuple (x, x_min, x_max)
- `yrange` - a 2-tuple (y_min, y_may) or a 3-tuple (y, y_min, y_may)
- `zrange` - a 2-tuple (z_min, z_maz) or a 3-tuple (z, z_min, z_maz)
- `plot_points` - (default: "automatic", which is 50) the number of function evaluations in each direction. (The number of cubes in the marching cubes algorithm will be one less than this). Can be a triple of integers, to specify a different resolution in each of x,y,z.
- `contour` - (default: 0) plot the isosurface f(x,y,z)==contour. Can be a list, in which case multiple contours are plotted.
- `region` - (default: None) If region is given, it must be a Python callable. Only segments of the surface where region(x,y,z) returns a number >0 will be included in the plot. (Note that returning a Python boolean is acceptable, since True == 1 and False == 0).

EXAMPLES:

```
sage: var('x,y,z')
(x, y, z)
```

A simple sphere:

```
sage: implicit_plot3d(x^2+y^2+z^2==4, (x, -3, 3), (y, -3,3), (z, -3,3))
```

A nested set of spheres with a hole cut out:

```
sage: implicit_plot3d((x^2 + y^2 + z^2), (x, -2, 2), (y, -2, 2), (z, -2, 2), plot_points=60, con
...                   region=lambda x,y,z: x<=0.2 or y>=0.2 or z<=0.2).show(viewer='tachyon')
```

A very pretty example from http://iat.ubalt.edu/summers/math/platsol.htm:

```
sage: T = RDF(golden_ratio)
sage: p = 2 - (cos(x + T*y) + cos(x - T*y) + cos(y + T*z) + cos(y - T*z) + cos(z - T*x) + cos(z
sage: r = 4.77
sage: implicit_plot3d(p, (x, -r, r), (y, -r, r), (z, -r, r), plot_points=40).show(viewer='tachyc
```

As I write this (but probably not as you read it), it's almost Valentine's day, so let's try a heart (from http://mathworld.wolfram.com/HeartSurface.html)

```
sage: p = (x^2+9/4*y^2+z^2-1)^3-x^2*z^3-9/(80)*y^2*z^3
sage: r = 1.5
sage: implicit_plot3d(p, (x, -r,r), (y, -r,r), (z, -r,r), plot_points=80, color='red', smooth=Fa
```

The same examples also work with the default Jmol viewer; for example:

```
sage: T = RDF(golden_ratio)
sage: p = 2 - (cos(x + T*y) + cos(x - T*y) + cos(y + T*z) + cos(y - T*z) + cos(z - T*x) + cos(z
sage: r = 4.77
sage: implicit_plot3d(p, (x, -r, r), (y, -r, r), (z, -r, r), plot_points=40).show()
```

Here we use smooth=True with a Tachyon graph:

```
sage: implicit_plot3d(x^2 + y^2 + z^2, (x, -2, 2), (y, -2, 2), (z, -2, 2), contour=4, smooth=Tru
```

We explicitly specify a gradient function (in conjunction with smooth=True) and invert the normals:

```
sage: gx = lambda x, y, z: -(2*x + y^2 + z^2)
sage: gy = lambda x, y, z: -(x^2 + 2*y + z^2)
sage: gz = lambda x, y, z: -(x^2 + y^2 + 2*z)
sage: implicit_plot3d(x^2+y^2+z^2, (x, -2, 2), (y, -2, 2), (z, -2, 2), contour=4, \
...       plot_points=40, smooth=True, gradient=(gx, gy, gz)).show(viewer='tachyon')
```

A graph of two metaballs interacting with each other:

```
sage: def metaball(x0, y0, z0): return 1 / ((x-x0)^2 + (y-y0)^2 + (z-z0)^2)
sage: implicit_plot3d(metaball(-0.6, 0, 0) + metaball(0.6, 0, 0), (x, -2, 2), (y, -2, 2), (z, -2
```

MANY MORE EXAMPLES:

A kind of saddle:

```
sage: implicit_plot3d(x^3 + y^2 - z^2, (x, -2, 2), (y, -2, 2), (z, -2, 2), plot_points=60, contc
```

A smooth surface with six radial openings:

```
sage: implicit_plot3d(-(cos(x) + cos(y) + cos(z)), (x, -4, 4), (y, -4, 4), (z, -4, 4))
```

A cube composed of eight conjoined blobs:

```
sage: implicit_plot3d(x^2 + y ^2 + z^2 +cos(4*x)+cos(4*y)+cos(4*z)-0.2, (x, -2, 2), (y, -2, 2),
```

A variation of the blob cube featuring heterogeneously sized blobs:

```
sage: implicit_plot3d(x^2 + y ^2 + z^2 +sin(4*x) + sin(4*y) + sin(4*z) -1, (x, -2, 2), (y, -2, 2
```

A klein bottle:

```
sage: implicit_plot3d((x^2+y^2+z^2+2*y-1)*((x^2+y^2+z^2-2*y-1)^2-8*z^2)+16*x*z*(x^2+y^2+z^2-2*y-
```

A leminescate:

```
sage: implicit_plot3d(4*x^2*(x^2+y^2+z^2+z)+y^2*(y^2+z^2-1), (x, -0.5, 0.5), (y, -1, 1), (z, -1,
```

Drope:

```
sage: implicit_plot3d(z - 4*x*exp(-x^2-y^2), (x, -2, 2), (y, -2, 2), (z, -1.7, 1.7))
```

A cube with a circular aperture on each face:

```
sage: implicit_plot3d(((1/2.3)^2 *(x^2 + y^2 + z^2))^-6 + ( (1/2)^8 * (x^8 + y^8 + z^8) )^6 -1,
```

A simple hyperbolic surface:

```
sage: implicit_plot3d(x*x + y - z*z, (x, -1, 1), (y, -1, 1), (z, -1, 1))
```

A hyperboloid:

```
sage: implicit_plot3d(x^2 + y^2 - z^2 -0.3, (x, -2, 2), (y, -2, 2), (z, -1.8, 1.8))
```

Duplin cycloid:

```
sage: implicit_plot3d((2^2 - 0^2 - (2 + 2.1)^2) * (2^2 - 0^2 - (2 - 2.1)^2)*(x^4+y^4+z^4)+ 2*((2
```

Sinus:

```
sage: implicit_plot3d(sin(pi*((x)^2+(y)^2))/2 +z, (x, -1, 1), (y, -1, 1), (z, -1, 1))
```

A torus:

```
sage: implicit_plot3d((sqrt(x*x+y*y)-3)^2 + z*z - 1, (x, -4, 4), (y, -4, 4), (z, -1, 1))
```

An octahedron:

```
sage: implicit_plot3d(abs(x)+abs(y)+abs(z) - 1, (x, -1, 1), (y, -1, 1), (z, -1, 1))
```

A cube:

```
sage: implicit_plot3d(x^100 + y^100 + z^100 -1, (x, -2, 2), (y, -2, 2), (z, -2, 2))
```

Toupie:

```
sage: implicit_plot3d((sqrt(x*x+y*y)-3)^3 + z*z - 1, (x, -4, 4), (y, -4, 4), (z, -6, 6))
```

A cube with rounded edges:

```
sage: implicit_plot3d(x^4 + y^4 + z^4 - (x^2 + y^2 + z^2), (x, -2, 2), (y, -2, 2), (z, -2, 2))
```

Chmutov:

```
sage: implicit_plot3d(x^4 + y^4 + z^4 - (x^2 + y^2 + z^2-0.3), (x, -1.5, 1.5), (y, -1.5, 1.5), (
```

Further Chutmov:

```
sage: implicit_plot3d(2*(x^2*(3-4*x^2)^2+y^2*(3-4*y^2)^2+z^2*(3-4*z^2)^2) -3, (x, -1.3, 1.3), (y
```

Clebsch:

```
sage: implicit_plot3d(81*(x^3+y^3+z^3)-189*(x^2*y+x^2*z+y^2*x+y^2*z+z^2*x+z^2*y) +54*x*y*z+126*(
```

Looks like a water droplet:

```
sage: implicit_plot3d(x^2 +y^2 -(1-z)*z^2, (x, -1.5, 1.5), (y, -1.5, 1.5), (z, -1, 1))
```

Sphere in a cage:

```
sage: implicit_plot3d((x^8 + z^30 + y^8 - (x^4 + z^50 + y^4 -0.3))*(x^2 + y^2 + z^2 -0.5), (x, -
```

Ortho circle:

```
sage: implicit_plot3d(((x^2 + y^2 - 1)^2 + z^2)* ((y^2 + z^2 - 1)^2 + x^2)* ((z^2 + x^2 - 1)^2 +
```

Cube sphere:

```
sage: implicit_plot3d(12 - ((1/2.3)^2 *(x^2 + y^2 + z^2))^-6 - ( (1/2)^8 * (x^8 + y^8 + z^8) )^6
```

Two cylinders intersect to make a cross:

```
sage: implicit_plot3d((x^2 + y^2 - 1) * ( x^2 + z^2 - 1) - 1, (x, -3, 3), (y, -3, 3), (z, -3, 3)
```

Three cylinders intersect in a similar fashion:

```
sage: implicit_plot3d((x^2 + y^2 - 1) * ( x^2 + z^2 - 1)* ( y^2 + z^2 - 1) - 1, (x, -3, 3), (y,
```

A sphere-ish object with twelve holes, four on each XYZ plane:

```
sage: implicit_plot3d(3*(cos(x) + cos(y) + cos(z)) + 4* cos(x) * cos(y) * cos(z), (x, -3, 3), (y
```

A gyroid:

```
sage: implicit_plot3d(cos(x) * sin(y) + cos(y) * sin(z) + cos(z) * sin(x), (x, -4, 4), (y, -4, 4
```

Tetrahedra:

```
sage: implicit_plot3d((x^2 + y^2 + z^2)^2 + 8*x*y*z - 10*(x^2 + y^2 + z^2) + 25, (x, -4, 4), (y,
```

TESTS:

Test a separate resolution in the X direction; this should look like a regular sphere:

```
sage: implicit_plot3d(x^2 + y^2 + z^2, (x, -2, 2), (y, -2, 2), (z, -2, 2), plot_points=(10, 40,
```

Test using different plot ranges in the different directions; this should generate half of a sphere:

```
sage: implicit_plot3d(x^2 + y^2 + z^2, (x, 0, 2), (y, -2, 2), (z, -2, 2), contour=4)
```

Extra keyword arguments will be passed to show():

```
sage: implicit_plot3d(x^2 + y^2 + z^2, (x, -2, 2), (y, -2, 2), (z, -2, 2), contour=4, viewer='ta
```

5.4 List Plots

list_plot3d(*v, interpolation_type='default', texture='automatic', point_list=None, **kwds*)

A 3-dimensional plot of a surface defined by the list v of points in 3-dimensional space.
INPUT:

- `v` - something that defines a set of points in 3 space, for example:
  - a matrix
  - a list of 3-tuples
  - a list of lists (all of the same length) - this is treated the same as a matrix.
- `texture` - (default: "automatic"), solid light blue

OPTIONAL KEYWORDS:

- `interpolation_type` - 'linear', 'nn' (nearest neighbor), 'spline'
  'linear' will perform linear interpolation
  The option 'nn' will interpolate by averaging the value of the nearest neighbors, this produces an interpolating function that is smoother than a linear interpolation, it has one derivative everywhere except at the sample points.
  The option 'spline' interpolates using a bivariate B-spline.
  When v is a matrix the default is to use linear interpolation, when v is a list of points the default is nearest neighbor.
- `degree` - an integer between 1 and 5, controls the degree of spline used for spline interpolation. For data that is highly oscillatory use higher values
- `point_list` - If point_list=True is passed, then if the array is a list of lists of length three, it will be treated as an array of points rather than a 3xn array.
- `num_points` - Number of points to sample interpolating function in each direction. By default for an nxn array this is n.
- `**kwds` - all other arguments are passed to the surface function

OUTPUT: a 3d plot
EXAMPLES:
We plot a matrix that illustrates summation modulo n.

```
sage: n = 5; list_plot3d(matrix(RDF,n,[(i+j)%n for i in [1..n] for j in [1..n]]))
```

We plot a matrix of values of sin.

```
sage: pi = float(pi)
sage: m = matrix(RDF, 6, [sin(i^2 + j^2) for i in [0,pi/5,..,pi] for j in [0,pi/5,..,pi]])
sage: list_plot3d(m, texture='yellow', frame_aspect_ratio=[1,1,1/3])
```

Though it doesn't change the shape of the graph, increasing num_points can increase the clarity of the graph.

```
sage: list_plot3d(m, texture='yellow', frame_aspect_ratio=[1,1,1/3],num_points=40)
```

We can change the interpolation type.

```
sage: list_plot3d(m, texture='yellow', interpolation_type='nn',frame_aspect_ratio=[1,1,1/3])
```

We can make this look better by increasing the number of samples.

```
sage: list_plot3d(m, texture='yellow', interpolation_type='nn',frame_aspect_ratio=[1,1,1/3],
```

Let's try a spline.

```
sage: list_plot3d(m, texture='yellow', interpolation_type='spline',frame_aspect_ratio=[1,1,1
```

That spline doesn't capture the oscillation very well; let's try a higher degree spline.

```
sage: list_plot3d(m, texture='yellow', interpolation_type='spline', degree=5, frame_aspect_r
```

We plot a list of lists:

```
sage: show(list_plot3d([[1, 1, 1, 1], [1, 2, 1, 2], [1, 1, 3, 1], [1, 2, 1, 4]]))
```

We plot a list of points. As a first example we can extract the (x,y,z) coordinates from the above example and make a list plot out of it. By default we do linear interpolation.

```
sage: l=[]
sage: for i in range(6):
...       for j in range(6):
...          l.append((float(i*pi/5),float(j*pi/5),m[i,j]))
sage: list_plot3d(l,texture='yellow')
```

Note that the points do not have to be regularly sampled. For example:

```
sage: l=[]
sage: for i in range(-5,5):
...       for j in range(-5,5):
...          l.append((normalvariate(0,1),normalvariate(0,1),normalvariate(0,1)))
sage: list_plot3d(l,interpolation_type='nn',texture='yellow',num_points=100)
```

TESTS: We plot 0, 1, and 2 points:

```
sage: list_plot3d([])
sage: list_plot3d([(2,3,4)])
sage: list_plot3d([(0,0,1), (2,3,4)])
```

However, if two points are given with the same x,y coordinates but different z coordinates, an exception will be raised:

```
sage: pts =[(-4/5, -2/5, -2/5), (-4/5, -2/5, 2/5), (-4/5, 2/5, -2/5), (-4/5, 2/5, 2/5), (-2/
sage: show(list_plot3d(pts, interpolation_type='nn'))
...
ValueError: Two points with same x,y coordinates and different z coordinates were given. Int
```

Additionally we need at least 3 points to do the interpolation:

```
sage: mat = matrix(RDF, 1, 2, [3.2, 1.550])
sage: show(list_plot3d(mat,interpolation_type='nn'))
...
ValueError: We need at least 3 points to perform the interpolation
```

list_plot3d_array_of_arrays(*v, interpolation_type, texture, **kwds*)

list_plot3d_matrix(*m, texture, **kwds*)

list_plot3d_tuples(*v, interpolation_type, texture, **kwds*)

5.5 Plotting Functions.

EXAMPLES:

```
sage: def f(x,y):
...       return math.sin(y*y+x*x)/math.sqrt(x*x+y*y+.0001)
...
sage: P = plot3d(f,(-3,3),(-3,3), adaptive=True, color=rainbow(60, 'rgbtuple'), max_bend=.1, max_dept
sage: P.show()
```

```
sage: def f(x,y):
...       return math.exp(x/5)*math.sin(y)
...
sage: P = plot3d(f,(-5,5),(-5,5), adaptive=True, color=['red','yellow'])
sage: from sage.plot.plot3d.plot3d import axes
sage: S = P + axes(6, color='black')
sage: S.show()
```

We plot "cape man":

```
sage: S = sphere(size=.5, color='yellow')


sage: from sage.plot.plot3d.shapes import Cone
sage: S += Cone(.5, .5, color='red').translate(0,0,.3)


sage: S += sphere((.45,-.1,.15), size=.1, color='white') + sphere((.51,-.1,.17), size=.05, color='bla
sage: S += sphere((.45, .1,.15),size=.1, color='white') + sphere((.51, .1,.17), size=.05, color='blac
sage: S += sphere((.5,0,-.2),size=.1, color='yellow')
sage: def f(x,y): return math.exp(x/5)*math.cos(y)
sage: P = plot3d(f,(-5,5),(-5,5), adaptive=True, color=['red','yellow'], max_depth=10)
sage: cape_man = P.scale(.2) + S.translate(1,0,0)
sage: cape_man.show(aspect_ratio=[1,1,1])
```

AUTHORS:

- Tom Boothby: adaptive refinement triangles
- Josh Kantor: adaptive refinement triangles
- Robert Bradshaw (2007-08): initial version of this file
- William Stein (2007-12, 2008-01): improving 3d plotting

class **TrivialTriangleFactory**()

smooth_triangle(*a, b, c, da, db, dc, color=None*)

triangle(*a, b, c, color=None*)

axes(*scale=1, radius=None, **kwds*)

plot3d(*f, urange, vrange, adaptive=False, **kwds*)

INPUT:

- `f` - a symbolic expression or function of 2 variables
- `urange` - a 2-tuple (u_min, u_max) or a 3-tuple (u, u_min, u_max)
- `vrange` - a 2-tuple (v_min, v_max) or a 3-tuple (v, v_min, v_max)
- `adaptive` - (default: False) whether to use adaptive refinement to draw the plot (slower, but may look better)
- `mesh` - bool (default: False) whether to display mesh grid lines
- `dots` - bool (default: False) whether to display dots at mesh grid points

Note: `mesh` and `dots` are not supported when using the Tachyon raytracer renderer.

EXAMPLES: We plot a 3d function defined as a Python function:

```
sage: plot3d(lambda x, y: x^2 + y^2, (-2,2), (-2,2))
```

We plot the same 3d function but using adaptive refinement:

```
sage: plot3d(lambda x, y: x^2 + y^2, (-2,2), (-2,2), adaptive=True)
```

Adaptive refinement but with more points:

```
sage: plot3d(lambda x, y: x^2 + y^2, (-2,2), (-2,2), adaptive=True, initial_depth=5)
```

We plot some 3d symbolic functions:

```
sage: x, y = var('x,y')
sage: plot3d(x^2 + y^2, (x,-2,2), (y,-2,2))
sage: plot3d(sin(x*y), (x, -pi, pi), (y, -pi, pi))
```

A 3d plot with a mesh:

```
sage: var('x,y')
(x, y)
sage: plot3d(sin(x-y)*y*cos(x),(x,-3,3),(y,-3,3), mesh=True)
```

Two wobby translucent planes:

```
sage: x,y = var('x,y')
sage: P = plot3d(x+y+sin(x*y), (x,-10,10),(y,-10,10), opacity=0.87, color='blue')
sage: Q = plot3d(x-2*y-cos(x*y),(x,-10,10),(y,-10,10),opacity=0.3,color='red')
sage: P + Q
```

We draw two parametric surfaces and a transparent plane:

```
sage: L = plot3d(lambda x,y: 0, (-5,5), (-5,5), color="lightblue", opacity=0.8)
sage: P = plot3d(lambda x,y: 4 - x^3 - y^2, (-2,2), (-2,2), color='green')
sage: Q = plot3d(lambda x,y: x^3 + y^2 - 4, (-2,2), (-2,2), color='orange')
sage: L + P + Q
```

We draw the "Sinus" function (water ripple-like surface):

```
sage: x, y = var('x y')
sage: plot3d(sin(pi*(x^2+y^2))/2,(x,-1,1),(y,-1,1))
```

Hill and valley (flat surface with a bump and a dent):

```
sage: x, y = var('x y')
sage: plot3d( 4*x*exp(-x^2-y^2), (x,-2,2), (y,-2,2))
```

TESTS: Listing the same plot variable twice gives an error.

```
sage: x, y = var('x y')
sage: plot3d( 4*x*exp(-x^2-y^2), (x,-2,2), (x,-2,2))
...
ValueError: plot variables should be distinct, but both are x.
```

plot3d_adaptive(*f, x_range, y_range, color='automatic', grad_f=None, max_bend=0.5, max_depth=5, initial_depth=4, num_colors=128, **kwds*)

Adaptive 3d plotting of a function of two variables.

This is used internally by the plot3d command when the option `adaptive=True` is given.

INPUT:

- f - a symbolic function or a Python function of 3 variables.
- x_range - x range of values: 2-tuple (xmin, xmax) or 3-tuple (x,xmin,xmax)
- y_range - y range of values: 2-tuple (ymin, ymax) or 3-tuple (y,ymin,ymax)
- grad_f - gradient of f as a Python function
- color - "automatic" - a rainbow of num_colors colors
- num_colors - (default: 128) number of colors to use with default color
- max_bend - (default: 0.5)
- max_depth - (default: 5)
- initial_depth - (default: 4)
- **kwds - standard graphics parameters

EXAMPLES: We plot $\sin(xy)$:

```
sage: from sage.plot.plot3d.plot3d import plot3d_adaptive
sage: x,y=var('x,y'); plot3d_adaptive(sin(x*y), (x,-pi,pi), (y,-pi,pi), initial_depth=5)
```

5.6 Platonic Solids.

EXAMPLES: The five platonic solids in a row;

```
sage: G = tetrahedron((0,-3.5,0), color='blue') + cube((0,-2,0),color=(.25,0,.5)) +\
      octahedron(color='red') + dodecahedron((0,2,0), color='orange') +\
      icosahedron(center=(0,4,0), color='yellow')
sage: G.show(aspect_ratio=[1,1,1])
```

All the platonic solids in the same place:

```
sage: G = tetrahedron(color='blue',opacity=0.7) + \
      cube(color=(.25,0,.5), opacity=0.7) +\
      octahedron(color='red', opacity=0.7) + \
      dodecahedron(color='orange', opacity=0.7) + icosahedron(opacity=0.7)
sage: G.show(aspect_ratio=[1,1,1])
```

Display nice faces only:

```
sage: icosahedron().stickers(['red','blue'], .075, .1)
```

AUTHORS:

- Robert Bradshaw (2007, 2008): initial version
- William Stein

cube (*center=(0, 0, 0), size=1, color=None, frame_thickness=0, frame_color=None, **kwds*)
A 3D cube centered at the origin with default side lengths 1.

INPUT:

- center - (default: (0,0,0))
- size - (default: 1) the side lengths of the cube

- color - a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.
- frame_thickness - (default: 0) if positive, then thickness of the frame
- frame_color - (default: None) if given, gives the color of the frame
- opacity - (default: 1) if less than 1 then it's transparent

EXAMPLES:

A simple cube:

```
sage: cube()
```

A red cube:

```
sage: cube(color="red")
```

A transparent grey cube that contains a red cube:

```
sage: cube(opacity=0.8, color='grey') + cube(size=3/4)
```

A transparent colored cube:

```
sage: cube(color=['red', 'green', 'blue'], opacity=0.5)
```

A bunch of random cubes:

```
sage: v = [(random(), random(), random()) for _ in [1..30]]
sage: sum([cube((10*a,10*b,10*c), size=random()/3, color=(a,b,c)) for a,b,c in v])
```

Non-square cubes (boxes):

```
sage: cube(aspect_ratio=[1,1,1]).scale([1,2,3])
sage: cube(color=['red', 'blue', 'green'],aspect_ratio=[1,1,1]).scale([1,2,3])
```

And one that is colored:

```
sage: cube(color=['red', 'blue', 'green', 'black', 'white', 'orange'],                    aspect_
```

A nice translucent color cube with a frame:

```
sage: c = cube(color=['red', 'blue', 'green'], frame=False, frame_thickness=2,
sage: c
```

A raytraced color cube with frame and transparency:

```
sage: c.show(viewer='tachyon')
```

AUTHORS:

- William Stein

dodecahedron (*center=(0, 0, 0), size=1, **kwds*)

A dodecahedron.

INPUT:

- center - (default: (0,0,0))
- size - (default: 1)

- `color` - a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.
- `opacity` - (default: 1) if less than 1 then is transparent

EXAMPLES: A plain Dodecahedron:

```
sage: dodecahedron()
```

A translucent dodecahedron that contains a black sphere:

```
sage: dodecahedron(color='orange', opacity=0.8) + \
      sphere(size=0.5, color='black')
```

CONSTRUCTION: This is how we construct a dodecahedron. We let one point be $Q = (0, 1, 0)$.

Now there are three points spaced equally on a circle around the north pole. The other requirement is that the angle between them be the angle of a pentagon, namely $3\pi/5$. This is enough to determine them. Placing one on the xz-plane we have.

$P_1 = \left(t, 0, \sqrt{1-t^2}\right)$

$P_2 = \left(-\frac{1}{2}t, \frac{\sqrt{3}}{2}t, \sqrt{1-t^2}\right)$

$P_3 = \left(-\frac{1}{2}t, \frac{\sqrt{3}}{2}t, \sqrt{1-t^2}\right)$

Solving $\frac{(P_1-Q)\cdot(P_2-Q)}{|P_1-Q||P_2-Q|} = \cos(3\pi/5)$ we get $t = 2/3$.

Now we have 6 points $R_1, ..., R_6$ to close the three top pentagons. These can be found by mirroring P_2 and P_3 by the yz-plane and rotating around the y-axis by the angle θ from Q to P_1. Note that $\cos(\theta) = t = 2/3$ and so $\sin(\theta) = \sqrt{5}/3$. Rotation gives us the other four.

Now we reflect through the origin for the bottom half.

AUTHORS:

- Robert Bradshaw, William Stein

icosahedron (*center=(0, 0, 0), size=1, **kwds*)

An icosahedron.

INPUT:

- `center` - (default: (0,0,0))
- `size` - (default: 1)
- `color` - a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.
- `opacity` - (default: 1) if less than 1 then is transparent

EXAMPLES:

```
sage: icosahedron()
```

Two icosahedrons at different positions of different sizes.

```
sage: icosahedron((-1/2,0,1), color='orange') + \
      icosahedron((2,0,1), size=1/2, aspect_ratio=[1,1,1])
```

index_face_set (*face_list, point_list, enclosed, **kwds*)

octahedron(*center=(0, 0, 0), size=1, **kwds*)

Return an octahedron.

INPUT:

- `center` - (default: (0,0,0))
- `size` - (default: 1)
- `color` - a string that describes a color; this can also be a list of 3-tuples or strings length 6 or 3, in which case the faces (and oppositive faces) are colored.
- `opacity` - (default: 1) if less than 1 then is transparent

EXAMPLES:

```
sage: octahedron((1,4,3), color='orange') + \
            octahedron((0,2,1), size=2, opacity=0.6)
```

prep(*G, center, size, kwds*)

tetrahedron(*center=(0, 0, 0), size=1, **kwds*)

A 3d tetrahedron.

INPUT:

- `center` - (default: (0,0,0))
- `size` - (default: 1)
- `color` - a word that describes a color
- `rgbcolor` - (r,g,b) with r, g, b between 0 and 1 that describes a color
- `opacity` - (default: 1) if less than 1 then is transparent

EXAMPLES: A default colored tetrahedron at the origin:

```
sage: tetrahedron()
```

A transparent green tetrahedron in front of a solid red one:

```
sage: tetrahedron(opacity=0.8, color='green') + tetrahedron((-2,1,0),color='red')
```

A translucent tetrahedron sharing space with a sphere:

```
sage: tetrahedron(color='yellow',opacity=0.7) + sphere(r=.5, color='red')
```

A big tetrahedron:

```
sage: tetrahedron(size=10)
```

A wide tetrahedron:

```
sage: tetrahedron(aspect_ratio=[1,1,1]).scale((4,4,1))
```

A red and blue tetrahedron touching noses:

```
sage: tetrahedron(color='red') + tetrahedron((0,0,-2)).scale([1,1,-1])
```

A Dodecahedral complex of 5 tetrahedrons (a more elaborate examples from Peter Jipsen):

```
sage: v=(sqrt(5.)/2-5/6, 5/6*sqrt(3.)-sqrt(15.)/2, sqrt(5.)/3)
sage: t=acos(sqrt(5.)/3)/2
sage: t1=tetrahedron(aspect_ratio=(1,1,1), opacity=0.5).rotateZ(t)
sage: t2=tetrahedron(color='red', opacity=0.5).rotateZ(t).rotate(v,2*pi/5)
sage: t3=tetrahedron(color='green', opacity=0.5).rotateZ(t).rotate(v,4*pi/5)
sage: t4=tetrahedron(color='yellow', opacity=0.5).rotateZ(t).rotate(v,6*pi/5)
sage: t5=tetrahedron(color='orange', opacity=0.5).rotateZ(t).rotate(v,8*pi/5)
sage: show(t1+t2+t3+t4+t5, frame=False, zoom=1.3)
```

AUTHORS:

- Robert Bradshaw and William Stein

5.7 Lines, Frames, Spheres, Points, Dots, and Text

class Line (*points, thickness=5, corner_cutoff=0.5, arrow_head=False, **kwds*)

Draw a 3d line joining a sequence of points.

This line has a fixed diameter unaffected by transformations and zooming. It may be smoothed if `corner_cutoff < 1`.

INPUT:

- `points` - list of points to pass through
- `thickness` - diameter of the line
- `corner_cutoff` - threshold for smoothing (see the corners() method) this is the minimum cosine between adjacent segments to smooth
- `arrow_head` - if True make this curve into an arrow

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes2 import Line
sage: Line([(i*math.sin(i), i*math.cos(i), i/3) for i in range(30)], arrow_head=True)
```

Smooth angles less than 90 degrees:

```
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=0)
```

bounding_box()

corners (*corner_cutoff=None, max_len=None*)

Figures out where the curve turns too sharply to pretend it's smooth.

INPUT: Maximum cosine of angle between adjacent line segments before adding a corner

OUTPUT: List of points at which to start a new line. This always includes the first point, and never the last.

EXAMPLES:

Every point:

```
sage: from sage.plot.plot3d.shapes2 import Line
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=1).corners()
[(0, 0, 0), (1, 0, 0), (2, 1, 0)]
```

Greater than 90 degrees:

```
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=0).corners()
[(0, 0, 0), (2, 1, 0)]
```

No corners:

```
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=-1).corners()
(0, 0, 0)
```

An intermediate value:

```
sage: Line([(0,0,0),(1,0,0),(2,1,0),(0,1,0)], corner_cutoff=.5).corners()
[(0, 0, 0), (2, 1, 0)]
```

jmol_repr(*render_params*)

obj_repr(*render_params*)

tachyon_repr(*render_params*)

class Point(*center, size=1, **kwds*)

Create a position in 3-space, represented by a sphere of fixed size.

INPUT:

- `center` - point (3-tuple)
- `size` - (default: 1)

bounding_box()

jmol_repr(*render_params*)

obj_repr(*render_params*)

tachyon_repr(*render_params*)

avg(*a, b*)

dot(*(x0, y0, z0), (x1, y1, z1)*)

frame3d(*lower_left, upper_right, **kwds*)

Draw a frame in 3d.

frame_labels(*lower_left, upper_right, label_lower_left, label_upper_right, eps=1, **kwds*)

line3d(*points, thickness=1, radius=None, arrow_head=False, **kwds*)

Draw a 3d line joining a sequence of points.

One may specify either a thickness or radius. If a thickness is specified, this line will have a constant diameter regardless of scaling and zooming. If a radius is specified, it will behave as a series of cylinders.

INPUT:

- `points` - a list of at least 2 points
- `thickness` - (default: 1)
- `radius` - (default: None)
- `arrow_head` - (default: False)
- `color` - a word that describes a color
- `rgbcolor` - (r,g,b) with r, g, b between 0 and 1 that describes a color
- `opacity` - (default: 1) if less than 1 then is transparent

EXAMPLES:

A line in 3-space:

```
sage: line3d([(1,2,3), (1,0,-2), (3,1,4), (2,1,-2)])
```

The same line but red:

```
sage: line3d([(1,2,3), (1,0,-2), (3,1,4), (2,1,-2)], color='red')
```

A transparent thick green line and a little blue line:

```
sage: line3d([(0,0,0), (1,1,1), (1,0,2)], opacity=0.5, radius=0.1, \
             color='green') + line3d([(0,1,0), (1,0,2)])
```

A Dodecahedral complex of 5 tetrahedrons (a more elaborate examples from Peter Jipsen):

```
sage: def tetra(col):
...       return line3d([(0,0,1), (2*sqrt(2.)/3,0,-1./3), (-sqrt(2.)/3, sqrt(6.)/3,-1./3),\
...              (-sqrt(2.)/3,-sqrt(6.)/3,-1./3), (0,0,1), (-sqrt(2.)/3, sqrt(6.)/3,-1./3),\
...              (-sqrt(2.)/3,-sqrt(6.)/3,-1./3), (2*sqrt(2.)/3,0,-1./3)],\
...              color=col, thickness=10, aspect_ratio=[1,1,1])
...
sage: v  = (sqrt(5.)/2-5/6, 5/6*sqrt(3.)-sqrt(15.)/2, sqrt(5.)/3)
sage: t  = acos(sqrt(5.)/3)/2
sage: t1 = tetra('blue').rotateZ(t)
sage: t2 = tetra('red').rotateZ(t).rotate(v,2*pi/5)
sage: t3 = tetra('green').rotateZ(t).rotate(v,4*pi/5)
sage: t4 = tetra('yellow').rotateZ(t).rotate(v,6*pi/5)
sage: t5 = tetra('orange').rotateZ(t).rotate(v,8*pi/5)
sage: show(t1+t2+t3+t4+t5, frame=False)
```

point3d(*v, size=5, **kwds*)

Plot a point or list of points in 3d space.

INPUT:

- `v` - a point or list of points
- `size` - (default: 5) size of the point (or points)
- `color` - a word that describes a color
- `rgbcolor` - (r,g,b) with r, g, b between 0 and 1 that describes a color
- `opacity` - (default: 1) if less than 1 then is transparent

EXAMPLES:

```
sage: sum([point3d((i,i^2,i^3), size=5) for i in range(10)])
```

We check to make sure this works with vectors.

sage: pl = point3d([vector(ZZ,(1, 0, 0)), vector(ZZ,(0, 1, 0)), (-1, -1, 0)])

polygon3d(**args, **kwds*)

Draw a polygon in 3d.

INPUT:

- `points` - the vertices of the polygon

Type `polygon3d.options` for a dictionary of the default options for polygons. You can change this to change the defaults for all future polygons. Use `polygon3d.reset()` to reset to the default options.

EXAMPLES:

A simple trangle:

```
sage: polygon3d([[0,0,0], [1,2,3], [3,0,0]])
```

Some modern art – a random polygon:

```
sage: v = [(randrange(-5,5), randrange(-5,5), randrange(-5, 5)) for _ in range(10)]
sage: polygon3d(v)
```

A bent transparent green triangle:

```
sage: polygon3d([[1, 2, 3], [0,1,0], [1,0,1], [3,0,0]], color=(0,1,0), alpha=0.7)
```

ruler (*start, end, ticks=4, sub_ticks=4, absolute=False, snap=False, **kwds*)

ruler_frame (*lower_left, upper_right, ticks=4, sub_ticks=4, **kwds*)

sphere (*center=(0, 0, 0), size=1, **kwds*)

Return a plot of a sphere of radius size centered at (x, y, z).

INPUT:

- `(x,y,z)` - center (default: (0,0,0)
- `size` - the radius (default: 1)

EXAMPLES: A simple sphere:

```
sage: sphere()
```

Two spheres touching:

```
sage: sphere(center=(-1,0,0)) + sphere(center=(1,0,0), aspect_ratio=[1,1,1])
```

Spheres of radii 1 and 2 one stuck into the other:

```
sage: sphere(color='orange') + sphere(color=(0,0,0.3), \
            center=(0,0,-2),size=2,opacity=0.9)
```

We draw a transparent sphere on a saddle.

```
sage: u,v = var('u v')
sage: saddle = plot3d(u^2 - v^2, (u,-2,2), (v,-2,2))
sage: sphere((0,0,1), color='red', opacity=0.5, aspect_ratio=[1,1,1]) + saddle
```

TESTS:

```
sage: T = sage.plot.plot3d.texture.Texture('red')
sage: S = sphere(texture=T)
sage: T in S.texture_set()
True
```

text3d (*txt, (x, y, z), **kwds*)

Display 3d text.

INPUT:

- `txt` - some text
- `(x,y,z)` - position
- `**kwds` - standard 3d graphics options

Note: There is no way to change the font size or opacity yet.

EXAMPLES: We write the word Sage in red at position (1,2,3):

```
sage: text3d("Sage", (1,2,3), color=(0.5,0,0))
```

We draw a multicolor spiral of numbers:

```
sage: sum([text3d('%.1f'%n, (cos(n),sin(n),n), color=(n/2,1-n/2,0)) \
           for n in [0,0.2,..,8]])
```

Another example

```
sage: text3d("Sage is really neat!!",(2,12,1))
```

And in 3d in two places:

```
sage: text3d("Sage is...",(2,12,1), rgbcolor=(1,0,0)) + text3d("quite powerful!!",(4,10,0), rgbc
```

validate_frame_size(*size*)

5.8 Base classes for 3D Graphics objects and plotting.

AUTHORS:

- Robert Bradshaw (2007-02): inital version
- Robert Bradshaw (2007-08): Cythonization, much optimization
- William Stein (2008)

TODO: - finish integrating tachyon - good default lights, camera

class BoundingSphere()

A bounding sphere is like a bounding box, but is simpler to deal with and behaves better under rotations.

transform()

Returns the bounding sphere of this sphere acted on by T. This always returns a new sphere, even if the resulting object is an ellipsoid.

EXAMPLES:

```
sage: from sage.plot.plot3d.transform import Transformation
sage: from sage.plot.plot3d.base import BoundingSphere
sage: BoundingSphere((0,0,0), 10).transform(Transformation(trans=(1,2,3)))
Center (1.0, 2.0, 3.0) radius 10.0
sage: BoundingSphere((0,0,0), 10).transform(Transformation(scale=(1/2, 1, 2)))
Center (0.0, 0.0, 0.0) radius 20.0
sage: BoundingSphere((0,0,3), 10).transform(Transformation(scale=(2, 2, 2)))
Center (0.0, 0.0, 6.0) radius 20.0
```

class Graphics3d()

This is the baseclass for all 3d graphics objects.

aspect_ratio()

Sets or gets the preferred aspect ratio of self.

EXAMPLES:

```
sage: D = dodecahedron()
sage: D.aspect_ratio()
[1.0, 1.0, 1.0]
sage: D.aspect_ratio([1,2,3])
sage: D.aspect_ratio()
[1.0, 2.0, 3.0]
sage: D.aspect_ratio(1)
sage: D.aspect_ratio()
[1.0, 1.0, 1.0]
```

bounding_box()

Returns the lower and upper corners of a 3d bounding box for self. This is used for rendering and self should fit entirely within this box.

Specifically, the first point returned should have x, y, and z coordinates should be the respective infimum over all points in self, and the second point is the supremum.

The default return value is simply the box containing the origin.

EXAMPLES:

```
sage: sphere((1,1,1), 2).bounding_box()
((-1.0, -1.0, -1.0), (3.0, 3.0, 3.0))
sage: G = line3d([(1, 2, 3), (-1,-2,-3)])
sage: G.bounding_box()
((-1.0, -2.0, -3.0), (1.0, 2.0, 3.0))
```

default_render_params()

Returns an instance of RenderParams suitable for plotting this object.

EXAMPLES:

```
sage: type(dodecahedron().default_render_params())
<class 'sage.plot.plot3d.base.RenderParams'>
```

export_jmol()

A jmol scene consists of a script which refers to external files. Fortunately, we are able to put all of them in a single zip archive, which is the output of this call.

EXAMPLES:

```
sage: out_file = sage.misc.misc.tmp_filename() + ".jmol"
sage: G = sphere((1, 2, 3), 5) + cube() + sage.plot.plot3d.shapes.Text("hi")
sage: G.export_jmol(out_file)
sage: import zipfile
sage: z = zipfile.ZipFile(out_file)
sage: z.namelist()
['obj_...pmesh', 'SCRIPT']

sage: print z.read('SCRIPT')
data "model list"
2
empty
Xx 0 0 0
Xx 5.5 5.5 5.5
end "model list"; show data
select *
wireframe off; spacefill off
set labelOffset 0 0
background [255,255,255]
spin OFF
moveto 0 -764 -346 -545 76.39
centerAt absolute {0 0 0}
```

```
zoom 100
frank OFF
set perspectivedepth ON
isosurface sphere_1  center {1.0 2.0 3.0} sphere 5.0
color isosurface  [102,102,255]
pmesh obj_... "obj_...pmesh"
color pmesh  [102,102,255]
select atomno = 1
color atom  [102,102,255]
label "hi"

sage: print z.read(z.namelist()[0])
24
0.5 0.5 0.5
-0.5 0.5 0.5
...
-0.5 -0.5 -0.5
6
5
0
1
...
```

flatten()

Try to reduce the depth of the scene tree by consolidating groups and transformations.

The generic Graphics3d object can't be made flatter.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.flatten() is G
True
```

frame_aspect_ratio()

Sets or gets the preferred frame aspect ratio of self.

EXAMPLES:

```
sage: D = dodecahedron()
sage: D.frame_aspect_ratio()
[1.0, 1.0, 1.0]
sage: D.frame_aspect_ratio([2,2,1])
sage: D.frame_aspect_ratio()
[2.0, 2.0, 1.0]
sage: D.frame_aspect_ratio(1)
sage: D.frame_aspect_ratio()
[1.0, 1.0, 1.0]
```

jmol_repr()

A (possibly nested) list of strings which will be concatenated and used by jmol to render self. (Nested lists of strings are used because otherwise all the intermediate concatenations can kill performance). This may refer to several remove files, which are stored in render_parames.output_archive.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.jmol_repr(G.default_render_params())
[]
sage: G = sphere((1, 2, 3))
sage: G.jmol_repr(G.default_render_params())
[['isosurface sphere_1  center {1.0 2.0 3.0} sphere 1.0\ncolor isosurface  [102,102,255]']]
```

mtl_str()

Returns the contents of a .mtl file, to be used to provide coloring information for an .obj file.

EXAMPLES:: sage: G = tetrahedron(color='red') + tetrahedron(color='yellow', opacity=0.5) sage: print G.mtl_str() newmtl ... Ka 0.5 0.0 0.0 Kd 1.0 0.0 0.0 Ks 0.0 0.0 0.0 illum 1 Ns 1 d 1 newmtl ... Ka 0.5 0.5 0.0 Kd 1.0 1.0 0.0 Ks 0.0 0.0 0.0 illum 1 Ns 1 d 0.500000000000000

obj()

An .obj scene file (as a string) containing the this object. A .mtl file of the same name must also be produced for coloring.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import ColorCube
sage: print ColorCube(1, ['red', 'yellow', 'blue']).obj()
g obj_1
usemtl ...
v 1 1 1
v -1 1 1
v -1 -1 1
v 1 -1 1
f 1 2 3 4
...
g obj_6
usemtl ...
v -1 -1 1
v -1 1 1
v -1 1 -1
v -1 -1 -1
f 21 22 23 24
```

obj_repr()

A (possibly nested) list of strings which will be concatenated and used to construct an .obj file of self. (Nested lists of strings are used because otherwise all the intermediate concatenations can kill performance). This may include a reference to color information which is stored elsewhere.

EXAMPLES:

```
sage: G = sage.plot.plot3d.base.Graphics3d()
sage: G.obj_repr(G.default_render_params())
[]
sage: G = cube()
sage: G.obj_repr(G.default_render_params())
['g obj_1',
 'usemtl ...',
 ['v 0.5 0.5 0.5',
  'v -0.5 0.5 0.5',
  'v -0.5 -0.5 0.5',
  'v 0.5 -0.5 0.5',
  'v 0.5 0.5 -0.5',
  'v -0.5 0.5 -0.5',
  'v 0.5 -0.5 -0.5',
  'v -0.5 -0.5 -0.5'],
 ['f 1 2 3 4',
  'f 1 5 6 2',
  'f 1 4 7 5',
  'f 6 5 7 8',
  'f 7 4 3 8',
  'f 3 2 6 8'],
 []]
```

rotate()

Returns self rotated about the vector v by $theta$ radians.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cone
sage: v = (1,2,3)
sage: G = arrow3d((0, 0, 0), v)
sage: G += Cone(1/5, 1).translate((0, 0, 2))
sage: C = Cone(1/5, 1, opacity=.25).translate((0, 0, 2))
sage: G += sum(C.rotate(v, pi*t/4) for t in [1..7])
sage: G.show(aspect_ratio=1)

sage: from sage.plot.plot3d.shapes import Box
sage: Box(1/3, 1/5, 1/7).rotate((1, 1, 1), pi/3).show(aspect_ratio=1)
```

rotateX()

Returns self rotated about the x-axis by the given angle.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cone
sage: G = Cone(1/5, 1) + Cone(1/5, 1, opacity=.25).rotateX(pi/2)
sage: G.show(aspect_ratio=1)
```

rotateY()

Returns self rotated about the y-axis by the given angle.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Cone
sage: G = Cone(1/5, 1) + Cone(1/5, 1, opacity=.25).rotateY(pi/3)
sage: G.show(aspect_ratio=1)
```

rotateZ()

Returns self rotated about the z-axis by the given angle.

EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Box
sage: G = Box(1/2, 1/3, 1/5) + Box(1/2, 1/3, 1/5, opacity=.25).rotateZ(pi/5)
sage: G.show(aspect_ratio=1)
```

scale()

Returns self scaled in the x, y, and z directions.

EXAMPLES:

```
sage: G = dodecahedron() + dodecahedron(opacity=.5).scale(2)
sage: G.show(aspect_ratio=1)
sage: G = icosahedron() + icosahedron(opacity=.5).scale([1, 1/2, 2])
sage: G.show(aspect_ratio=1)
```

TESTS:

```
sage: G = sphere((0, 0, 0), 1)
sage: G.scale(2)
sage: G.scale(1, 2, 1/2).show(aspect_ratio=1)
sage: G.scale(2).bounding_box()
((-2.0, -2.0, -2.0), (2.0, 2.0, 2.0))
```

show()

INPUT:

- `viewer` - string (default: 'jmol'), how to view the plot 'jmol': interactive 3d (java) 'tachyon': a static png image (ray traced) 'java3d': interactive opengl based 3d
- `filename` - string (default: a temp file); file to save the image to

- verbosity - display information about rendering the figure
- figsize - (default: 5); x or pair [x,y] for numbers, e.g., [5,5]; controls the size of the output figure. E.g., with Tachyon the number of pixels in each direction is 100 times figsize[0]. This is ignored for the jmol embedded renderer.
- aspect_ratio - (default: "automatic") - aspect ratio of the coordinate system itself. Give [1,1,1] to make spheres look round.
- frame_aspect_ratio - (default: "automatic") aspect ratio of frame that contains the 3d scene.
- zoom - (default: 1) how zoomed in
- frame - (default: True) if True, draw a bounding frame with labels
- axes - (deault: False) if True, draw coordinate axes
- **kwds - other options, which make sense for particular rendering engines

CHANGING DEFAULTS: Defaults can be uniformly changed by importing a dictionary and changing it. For example, here we change the default so images display without a frame instead of with one:

```
sage: from sage.plot.plot3d.base import SHOW_DEFAULTS
sage: SHOW_DEFAULTS['frame'] = False
```

This sphere will not have a frame around it:

```
sage: sphere((0,0,0))
```

We change the default back:

```
sage: SHOW_DEFAULTS['frame'] = True
```

Now this sphere is enclosed in a frame:

```
sage: sphere((0,0,0))
```

EXAMPLES: We illustrate use of the aspect_ratio option:

```
sage: x, y = var('x,y')
sage: p = plot3d(2*sin(x*y), (x, -pi, pi), (y, -pi, pi))
sage: p.show(aspect_ratio=[1,1,1])
```

This looks flattened, but filled with the plot:

```
sage: p.show(frame_aspect_ratio=[1,1,1/16])
```

This looks flattened, but the plot is square and smaller:

```
sage: p.show(aspect_ratio=[1,1,1], frame_aspect_ratio=[1,1,1/8])
```

tachyon()

An tachyon input file (as a string) containing the this object.

EXAMPLES:

```
sage: print sphere((1, 2, 3), 5, color='yellow').tachyon()
begin_scene
resolution 400 400
         camera
        ...
      plane
        center -2000 -1000 -500
        normal 2.3 2.4 2.0
        TEXTURE
            AMBIENT 1.0 DIFFUSE 1.0 SPECULAR 1.0 OPACITY 1.0
            COLOR 1.0 1.0 1.0
            TEXFUNC 0
    Texdef texture...
```

```
  Ambient 0.333333333333 Diffuse 0.666666666667 Specular 0.0 Opacity 1
   Color 1.0 1.0 0.0
   TexFunc 0
    Sphere center 1.0 -2.0 3.0 Rad 5.0 texture...
end_scene

sage: G = icosahedron(color='red') + sphere((1,2,3), 0.5, color='yellow')
sage: G.show(viewer='tachyon', frame=false)
sage: print G.tachyon()
begin_scene
...
Texdef texture...
  Ambient 0.333333333333 Diffuse 0.666666666667 Specular 0.0 Opacity 1
   Color 1.0 0.0 0.0
   TexFunc 0
TRI V0 ...
Sphere center 1.0 -2.0 3.0 Rad 0.5 texture...
end_scene
```

tachyon_repr()

A (possibly nested) list of strings which will be concatenated and used by tachyon to render self. (Nested lists of strings are used because otherwise all the intermediate concatenations can kill performance). This may include a reference to color information which is stored elsewhere.

EXAMPLES:: sage: G = sage.plot.plot3d.base.Graphics3d() sage: G.tachyon_repr(G.default_render_params()) [] sage: G = sphere((1, 2, 3)) sage: G.tachyon_repr(G.default_render_params()) ['Sphere center 1.0 2.0 3.0 Rad 1.0 texture...']

testing_render_params()

Returns an instance of RenderParams suitable for testing this object. In particular, it opens up '/dev/null' as an auxiliary zip file for jmol.

EXAMPLES:

```
sage: type(dodecahedron().testing_render_params())
<class 'sage.plot.plot3d.base.RenderParams'>
```

texture

texture_set()

Often the textures of a 3d file format are kept separate from the objects themselves. This function returns the set of textures used, so they can be defined in a preamble or separate file.

EXAMPLES:

```
sage: sage.plot.plot3d.base.Graphics3d().texture_set()
set([])

sage: G = tetrahedron(color='red') + tetrahedron(color='yellow') + tetrahedron(color='red',
sage: G.texture_set()
set([Texture(texture..., red, ff0000), Texture(texture..., yellow, ffff00), Texture(texture.
```

transform()

Apply a transformation to self, where the inputs are passed onto a TransformGroup object. Mostly for internal use; see the translate, scale, and rotate methods for more details.

EXAMPLES:

```
sage: sphere((0,0,0), 1).transform(trans=(1, 0, 0), scale=(2,3,4)).bounding_box()
((-1.0, -3.0, -4.0), (3.0, 3.0, 4.0))
```

translate()

Return self translated by the given vector (which can be given either as a 3-iterable or via positional

arguments).

EXAMPLES:

```
sage: icosahedron() + sum(icosahedron(opacity=0.25).translate(2*n, 0, 0) for n in [1..4])
sage: icosahedron() + sum(icosahedron(opacity=0.25).translate([-2*n, n, n^2]) for n in [1..4
```

TESTS:

```
sage: G = sphere((0, 0, 0), 1)
sage: G.bounding_box()
((-1.0, -1.0, -1.0), (1.0, 1.0, 1.0))
sage: G.translate(0, 0, 1).bounding_box()
((-1.0, -1.0, 0.0), (1.0, 1.0, 2.0))
sage: G.translate(-1, 5, 0).bounding_box()
((-2.0, 4.0, -1.0), (0.0, 6.0, 1.0))
```

viewpoint()

Returns the viewpoint of this plot. Currently only a stub for x3d.

EXAMPLES:

```
sage: type(dodecahedron().viewpoint())
<class 'sage.plot.plot3d.base.Viewpoint'>
```

x3d()

An x3d scene file (as a string) containing the this object.

EXAMPLES:

```
sage: print sphere((1, 2, 3), 5).x3d()
<X3D version='3.0' profile='Immersive' xmlns:xsd='http://www.w3.org/2001/XMLSchema-instance'
<head>
<meta name='title' content='sage3d'/>
</head>
<Scene>
<Viewpoint position='0 0 6'/>
<Transform translation='1 2 3'>
<Shape><Sphere radius='5.0'/><Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1'
</Transform>
</Scene>
</X3D>

sage: G = icosahedron() + sphere((0,0,0), 0.5, color='red')
sage: print G.x3d()
<X3D version='3.0' profile='Immersive' xmlns:xsd='http://www.w3.org/2001/XMLSchema-instance'
<head>
<meta name='title' content='sage3d'/>
</head>
<Scene>
<Viewpoint position='0 0 6'/>
<Shape>
<IndexedFaceSet coordIndex='...'>
  <Coordinate point='...'/>
</IndexedFaceSet>
<Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1' specularColor='0.0 0.0 0.0'/>
<Transform translation='0 0 0'>
<Shape><Sphere radius='0.5'/><Appearance><Material diffuseColor='1.0 0.0 0.0' shininess='1'
</Transform>
</Scene>
</X3D>
```

class **Graphics3dGroup**()

This class represents a collection of 3d objects. Usually they are formed implicitly by summing.

bounding_box()

Box that contains the bounding boxes of all the objects that make up self.

EXAMPLES:

```
sage: A = sphere((0,0,0), 5)
sage: B = sphere((1, 5, 10), 1)
sage: A.bounding_box()
((-5.0, -5.0, -5.0), (5.0, 5.0, 5.0))
sage: B.bounding_box()
((0.0, 4.0, 9.0), (2.0, 6.0, 11.0))
sage: (A+B).bounding_box()
((-5.0, -5.0, -5.0), (5.0, 6.0, 11.0))
sage: (A+B).show(aspect_ratio=1, frame=True)

sage: sage.plot.plot3d.base.Graphics3dGroup([]).bounding_box()
((0.0, 0.0, 0.0), (0.0, 0.0, 0.0))
```

flatten()

Try to reduce the depth of the scene tree by consolidating groups and transformations.

EXAMPLES:

```
sage: G = sum([circle((0, 0), t) for t in [1..10]], sphere()); G
sage: G.flatten()
sage: len(G.all)
2
sage: len(G.flatten().all)
11
```

jmol_repr()

The jmol representation of a group is simply the concatenation of the representation of its objects.

EXAMPLES:

```
sage: G = sphere() + sphere((1,2,3))
sage: G.jmol_repr(G.default_render_params())
[[['isosurface sphere_1  center {0.0 0.0 0.0} sphere 1.0\ncolor isosurface  [102,102,255]']]
 [['isosurface sphere_2  center {1.0 2.0 3.0} sphere 1.0\ncolor isosurface  [102,102,255]']]
```

obj_repr()

The obj representation of a group is simply the concatenation of the representation of its objects.

EXAMPLES:

```
sage: G = tetrahedron() + tetrahedron().translate(10, 10, 10)
sage: G.obj_repr(G.default_render_params())
[['g obj_1',
  'usemtl ...',
  ['v 0 0 1',
   'v 0.942809 0 -0.333333',
   'v -0.471405 0.816497 -0.333333',
   'v -0.471405 -0.816497 -0.333333'],
  ['f 1 2 3', 'f 2 4 3', 'f 1 3 4', 'f 1 4 2'],
  []],
 [['g obj_2',
   'usemtl ...',
   ['v 10 10 11',
    'v 10.9428 10 9.66667',
    'v 9.5286 10.8165 9.66667',
    'v 9.5286 9.1835 9.66667'],
```

```
        ['f 5 6 7', 'f 6 8 7', 'f 5 7 8', 'f 5 8 6'],
        []]]]
```

set_texture()

EXAMPLES:

```
sage: G = dodecahedron(color='red', opacity=.5) + icosahedron((3, 0, 0), color='blue')
sage: G
sage: G.set_texture(color='yellow')
sage: G
```

tachyon_repr()

The tachyon representation of a group is simply the concatenation of the representations of its objects.

EXAMPLES:

```
sage: G = sphere() + sphere((1,2,3))
sage: G.tachyon_repr(G.default_render_params())
[['Sphere center 0.0 0.0 0.0 Rad 1.0 texture...'],
 ['Sphere center 1.0 2.0 3.0 Rad 1.0 texture...']]
```

texture_set()

The texture set of a group is simply the union of the textures of all its objects.

EXAMPLES:

```
sage: G = sphere(color='red') + sphere(color='yellow')
sage: G.texture_set()
set([Texture(texture..., yellow, ffff00), Texture(texture... red, ff0000)])

sage: T = sage.plot.plot3d.texture.Texture('blue'); T
Texture(texture..., blue, 0000ff)
sage: G = sphere(texture=T) + sphere((1, 1, 1), texture=T)
sage: len(G.texture_set())
1
```

transform()

Transforming this entire group simply makes a transform group with the same contents.

EXAMPLES:

```
sage: G = dodecahedron(color='red', opacity=.5) + icosahedron(color='blue')
sage: G
sage: G.transform(scale=(2,1/2,1))
sage: G.transform(trans=(1,1,3))
```

x3d_str()

The x3d representation of a group is simply the concatenation of the representation of its objects.

EXAMPLES:

```
sage: G = sphere() + sphere((1,2,3))
sage: print G.x3d_str()
<Transform translation='0 0 0'>
<Shape><Sphere radius='1.0'/><Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1'
</Transform>
<Transform translation='1 2 3'>
<Shape><Sphere radius='1.0'/><Appearance><Material diffuseColor='0.4 0.4 1.0' shininess='1'
</Transform>
```

class **PrimitiveObject**()

This is the base class for the non-container 3d objects.

get_texture()
EXAMPLES:

```
sage: G = dodecahedron(color='red')
sage: G.get_texture()
Texture(texture..., red, ff0000)
```

jmol_repr()
Default behavior is to render the triangulation. The actual polygon data is stored in a separate file.
EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: G = Torus(1, .5)
sage: G.jmol_repr(G.testing_render_params())
['pmesh obj_1 "obj_1.pmesh"\ncolor pmesh  [102,102,255]']
```

obj_repr()
Default behavior is to render the triangulation.
EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: G = Torus(1, .5)
sage: G.obj_repr(G.default_render_params())
['g obj_1',
 'usemtl ...',
 ['v 0 1 0.5',
 ...
  'f ...'],
 []]
```

set_texture()
EXAMPLES:

```
sage: G = dodecahedron(color='red'); G
sage: G.set_texture(color='yellow'); G
```

tachyon_repr()
Default behavior is to render the triangulation.
EXAMPLES:

```
sage: from sage.plot.plot3d.shapes import Torus
sage: G = Torus(1, .5)
sage: G.tachyon_repr(G.default_render_params())
['TRI V0 0 1 0.5
...
'texture...']
```

texture_set()
EXAMPLES:

```
sage: G = dodecahedron(color='red')
sage: G.texture_set()
set([Texture(texture..., red, ff0000)])
```

x3d_str()
EXAMPLES:

```
sage: sphere().flatten().x3d_str()
"<Transform>\n<Shape><Sphere radius='1.0'/><Appearance><Material diffuseColor='0.4 0.4 1.0'
```

class RenderParams()

This class is a container for all parameters that may be needed to render triangulate/render an object to a certain format. It can contain both cumulative and global parameters.

Of particular note is the transformation object, which holds the cumulative transformation from the root of the scene graph to this node in the tree.

pop_transform()

Remove the last transformation off the stack, resetting self.transform to the previous value.

EXAMPLES:

```
sage: from sage.plot.plot3d.transform import Transformation
sage: params = sage.plot.plot3d.base.RenderParams()
sage: T = Transformation(trans=(100, 500, 0))
sage: params.push_transform(T)
sage: params.transform.get_matrix()
[  1.0   0.0   0.0 100.0]
[  0.0   1.0   0.0 500.0]
[  0.0   0.0   1.0   0.0]
[  0.0   0.0   0.0   1.0]
sage: params.push_transform(Transformation(trans=(-100, 500, 200)))
sage: params.transform.get_matrix()
[   1.0    0.0    0.0    0.0]
[   0.0    1.0    0.0 1000.0]
[   0.0    0.0    1.0  200.0]
[   0.0    0.0    0.0    1.0]
sage: params.pop_transform()
sage: params.transform.get_matrix()
[  1.0   0.0   0.0 100.0]
[  0.0   1.0   0.0 500.0]
[  0.0   0.0   1.0   0.0]
[  0.0   0.0   0.0   1.0]
```

push_transform()

Push a transformation onto the stack, updating self.transform.

EXAMPLES:

```
sage: from sage.plot.plot3d.transform import Transformation
sage: params = sage.plot.plot3d.base.RenderParams()
sage: params.transform is None
True
sage: T = Transformation(scale=(10,20,30))
sage: params.push_transform(T)
sage: params.transform.get_matrix()
[10.0  0.0  0.0  0.0]
[ 0.0 20.0  0.0  0.0]
[ 0.0  0.0 30.0  0.0]
[ 0.0  0.0  0.0  1.0]
sage: params.push_transform(T)  # scale again
sage: params.transform.get_matrix()
[100.0   0.0   0.0   0.0]
[  0.0 400.0   0.0   0.0]
[  0.0   0.0 900.0   0.0]
[  0.0   0.0   0.0   1.0]
```

unique_name()

Returns a unique identifier starting with desc.

EXAMPLES:

```
sage: params = sage.plot.plot3d.base.RenderParams()
sage: params.unique_name()
'name_1'
sage: params.unique_name()
'name_2'
sage: params.unique_name('texture')
'texture_3'
```

class TransformGroup()

This class is a container for a group of objects with a common transformation.

bounding_box()

Returns the bounding box of self, i.e. the box containing the contents of self after applying the transformation.

EXAMPLES:

```
sage: G = cube()
sage: G.bounding_box()
((-0.5, -0.5, -0.5), (0.5, 0.5, 0.5))
sage: G.scale(4).bounding_box()
((-2.0, -2.0, -2.0), (2.0, 2.0, 2.0))
sage: G.rotateZ(pi/4).bounding_box()
((-0.70710678118654746, -0.70710678118654746, -0.5),
 (0.70710678118654746, 0.70710678118654746, 0.5))
```

flatten()

Try to reduce the depth of the scene tree by consolidating groups and transformations.

EXAMPLES:

```
sage: G = sphere((1,2,3)).scale(100)
sage: T = G.get_transformation()
sage: T.get_matrix()
[100.0   0.0   0.0   0.0]
[  0.0 100.0   0.0   0.0]
[  0.0   0.0 100.0   0.0]
[  0.0   0.0   0.0   1.0]

sage: G.flatten().get_transformation().get_matrix()
[100.0   0.0   0.0 100.0]
[  0.0 100.0   0.0 200.0]
[  0.0   0.0 100.0 300.0]
[  0.0   0.0   0.0   1.0]
```

get_transformation()

Returns the actual transformation object associated with self.

EXAMPLES:

```
sage: G = sphere().scale(100)
sage: T = G.get_transformation()
sage: T.get_matrix()
[100.0   0.0   0.0   0.0]
[  0.0 100.0   0.0   0.0]
[  0.0   0.0 100.0   0.0]
[  0.0   0.0   0.0   1.0]
```

jmol_repr()

Transformations for jmol are applied at the leaf nodes.

EXAMPLES:

```
sage: G = sphere((1,2,3)).scale(2)
sage: G.jmol_repr(G.default_render_params())
[[['isosurface sphere_1  center {2.0 4.0 6.0} sphere 2.0\ncolor isosurface  [102,102,255]']]
```

obj_repr()

Transformations for .obj files are applied at the leaf nodes.

EXAMPLES:

```
sage: G = cube().scale(4).translate(1, 2, 3)
sage: G.obj_repr(G.default_render_params())
[['g obj_1',
  'usemtl ...',
  ['v 3 4 5',
   'v -1 4 5',
   'v -1 0 5',
   'v 3 0 5',
   'v 3 4 1',
   'v -1 4 1',
   'v 3 0 1',
   'v -1 0 1'],
  ['f 1 2 3 4',
   'f 1 5 6 2',
   'f 1 4 7 5',
   'f 6 5 7 8',
   'f 7 4 3 8',
   'f 3 2 6 8'],
  []]]
```

tachyon_repr()

Transformations for Tachyon are applied at the leaf nodes.

EXAMPLES:

```
sage: G = sphere((1,2,3)).scale(2)
sage: G.tachyon_repr(G.default_render_params())
[['Sphere center 2.0 4.0 6.0 Rad 2.0 texture...']]
```

transform()

Transforming this entire group can be done by composing transformations.

EXAMPLES:

```
sage: G = dodecahedron(color='red', opacity=.5) + icosahedron(color='blue')
sage: G
sage: G.transform(scale=(2,1/2,1))
sage: G.transform(trans=(1,1,3))
```

x3d_str()

To apply a transformation to a set of objects in x3d, simply make them all children of an x3d Transform node.

EXAMPLES:

```
sage: sphere((1,2,3)).x3d_str()
"<Transform translation='1 2 3'>\n<Shape><Sphere radius='1.0'/><Appearance><Material diffuse
```

class **Viewpoint**()

This class represents a viewpoint, necessary for x3d.

In the future, there could be multiple viewpoints, and they could have more properties. (Currently they only hold a position).

x3d_str()
EXAMPLES:

```
sage: sphere((0,0,0), 100).viewpoint().x3d_str()
"<Viewpoint position='0 0 6'/>"
```

flatten_list()
This is an optimized routine to turn a list of lists (of lists ...) into a single list. We generate data in a non-flat format to avoid multiple data copying, and then concatenate it all at the end.

This is NOT recursive, otherwise there would be a lot of redundant copying (which we are trying to avoid in the first place, though at least it would be just the pointers).

EXAMPLES:

```
sage: from sage.plot.plot3d.base import flatten_list
sage: flatten_list([])
[]
sage: flatten_list([[[[]]]])
[]
sage: flatten_list([['a', 'b'], 'c'])
['a', 'b', 'c']
sage: flatten_list([['a'], [['b'], 'c'], ['d'], [['e', 'f', 'g']]])
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

max3()
Return the componentwise maximum of a list of 3-tuples.

EXAMPLES:

```
sage: from sage.plot.plot3d.base import min3, max3
sage: max3([(-1,2,5), (-3, 4, 2)])
(-1, 4, 5)
```

min3()
Return the componentwise minimum of a list of 3-tuples.

EXAMPLES:

```
sage: from sage.plot.plot3d.base import min3, max3
sage: min3([(-1,2,5), (-3, 4, 2)])
(-3, 2, 2)
```

optimal_aspect_ratios()

optimal_extra_kwds()
Given a list v of dictionaries, this function merges them such that later dictionaries have precedence.

point_list_bounding_box()
EXAMPLES:

```
sage: from sage.plot.plot3d.base import point_list_bounding_box
sage: point_list_bounding_box([(1,2,3),(4,5,6),(-10,0,10)])
((-10.0, 0.0, 3.0), (4.0, 5.0, 10.0))
```

5.9 The Tachyon 3D Ray Tracer

Given any 3D graphics object one can compute a raytraced representation by typing `show(viewer='tachyon')`. For example, we draw two translucent spheres that contain a red tube, and render the result using Tachyon.

```
sage: S = sphere(opacity=0.8, aspect_ratio=[1,1,1])
sage: L = line3d([(0,0,0),(2,0,0)], thickness=10, color='red')
sage: M = S + S.translate((2,0,0)) + L
sage: M.show(viewer='tachyon')
```

One can also directly control Tachyon, which gives a huge amount of flexibility. For example, here we directly use Tachyon to draw 3 spheres on the coordinate axes. Notice that the result is gorgeous:

```
sage: t = Tachyon(xres=500,yres=500, camera_center=(2,0,0))
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t2', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1,0,0))
sage: t.texture('t3', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,1,0))
sage: t.texture('t4', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,0,1))
sage: t.sphere((0,0.5,0), 0.2, 't2')
sage: t.sphere((0.5,0,0), 0.2, 't3')
sage: t.sphere((0,0,0.5), 0.2, 't4')
sage: t.show()
```

AUTHOR:

- John E. Stone (johns@megapixel.com): wrote tachyon ray tracer
- William Stein: sage-tachyon interface
- Joshua Kantor: 3d function plotting
- Tom Boothby: 3d function plotting n'stuff
- Leif Hille: key idea for bugfix for texfunc issue (trac #799)

TODO:

- clean up trianglefactory stuff

class **Cylinder**(*center, axis, radius, texture*)

str()

class **FCylinder**(*base, apex, radius, texture*)

str()

class **Light**(*center, radius, color*)

str()

class **ParametricPlot**(*f, t_0, t_f, tex, r=0.10000000000000001, cylinders=True, min_depth=4, max_depth=8, e_rel=0.01, e_abs=0.01*)

str()

tol(*est, val*)

class **Plane**(*center, normal, texture*)

str()

class PlotBlock(*left, left_c, top, top_c, right, right_c, bottom, bottom_c*)

class Sphere(*center, radius, texture*)

str()

class Tachyon(*xres=350, yres=350, zoom=1.0, antialiasing=False, aspectratio=1.0, raydepth=8, camera_center=(-3, 0, 0), updir=(0, 0, 1), look_at=(0, 0, 0), viewdir=None, projection='PERSPECTIVE'*)

Create a scene the can be rendered using the Tachyon ray tracer.

INPUT:

- `xres` - (default 350)
- `yres` - (default 350)
- `zoom` - (default 1.0)
- `antialiasing` - (default False)
- `aspectratio` - (default 1.0)
- `raydepth` - (default 5)
- `camera_center` - (default (-3, 0, 0))
- `updir` - (default (0, 0, 1))
- `look_at` - (default (0,0,0))
- `viewdir` - (default None)
- `projection` - (default 'PERSPECTIVE')

OUTPUT: A Tachyon 3d scene.

Note that the coordinates are by default such that z is up, positive y is to the {left} and x is toward you. This is not oriented according to the right hand rule.

EXAMPLES: Spheres along the twisted cubic.

```
sage: t = Tachyon(xres=512,yres=512, camera_center=(3,0.3,0))
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1.0,0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.3, opacity=1.0, color=(0,1.0,0))
sage: t.texture('t2', ambient=0.2,diffuse=0.7, specular=0.5, opacity=0.7, color=(0,0,1.0))
sage: k=0
sage: for i in srange(-1,1,0.05):
...     k += 1
...     t.sphere((i,i^2-0.5,i^3), 0.1, 't%s'%(k%3))
...
sage: t.show()
```

Another twisted cubic, but with a white background, got by putting infinite planes around the scene.

```
sage: t = Tachyon(xres=512,yres=512, camera_center=(3,0.3,0), raydepth=8)
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1.0,0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.3, opacity=1.0, color=(0,1.0,0))
sage: t.texture('t2', ambient=0.2,diffuse=0.7, specular=0.5, opacity=0.7, color=(0,0,1.0))
sage: t.texture('white', color=(1,1,1))
sage: t.plane((0,0,-1), (0,0,1), 'white')
sage: t.plane((0,-20,0), (0,1,0), 'white')
sage: t.plane((-20,0,0), (1,0,0), 'white')
```

```
sage: k=0
sage: for i in srange(-1,1,0.05):
...     k += 1
...     t.sphere((i,i^2 - 0.5,i^3), 0.1, 't%s'%(k%3))
...     t.cylinder((0,0,0), (0,0,1), 0.05,'t1')
...
sage: t.show()
```

Many random spheres:

```
sage: t = Tachyon(xres=512,yres=512, camera_center=(2,0.5,0.5), look_at=(0.5,0.5,0.5), raydepth=
sage: t.light((4,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1.0,0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.3, opacity=1.0, color=(0,1.0,0))
sage: t.texture('t2', ambient=0.2, diffuse=0.7, specular=0.5, opacity=0.7, color=(0,0,1.0))
sage: k=0
sage: for i in range(100):
...     k += 1
...     t.sphere((random(),random(), random()), random()/10, 't%s'%(k%3))
...
sage: t.show()
```

Points on an elliptic curve, their height indicated by their height above the axis:

```
sage: t = Tachyon(camera_center=(5,2,2), look_at=(0,1,0))
sage: t.light((10,3,2), 0.2, (1,1,1))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1,0,0))
sage: t.texture('t1', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,1,0))
sage: t.texture('t2', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(0,0,1))
sage: E = EllipticCurve('37a')
sage: P = E([0,0])
sage: Q = P
sage: n = 100
sage: for i in range(n):   # increase 20 for a better plot
...     Q = Q + P
...     t.sphere((Q[1], Q[0], ZZ(i)/n), 0.1, 't%s'%(i%3))
...
sage: t.show()
```

A beautiful picture of rational points on a rank 1 elliptic curve.

```
sage: t = Tachyon(xres=1000, yres=800, camera_center=(2,7,4), look_at=(2,0,0), raydepth=4)
sage: t.light((10,3,2), 1, (1,1,1))
sage: t.light((10,-3,2), 1, (1,1,1))
sage: t.texture('black', color=(0,0,0))
sage: t.texture('red', color=(1,0,0))
sage: t.texture('grey', color=(.9,.9,.9))
sage: t.plane((0,0,0),(0,0,1),'grey')
sage: t.cylinder((0,0,0),(1,0,0),.01,'black')
sage: t.cylinder((0,0,0),(0,1,0),.01,'black')
sage: E = EllipticCurve('37a')
sage: P = E([0,0])
sage: Q = P
sage: n = 100
sage: for i in range(n):
...     Q = Q + P
...     c = i/n + .1
...     t.texture('r%s'%i,color=(float(i/n),0,0))
```

```
...     t.sphere((Q[0], -Q[1], .01), .04, 'r%s'%i)
...
...
sage: t.show()     # long time, e.g., 10-20 seconds
```

A beautiful spiral.

```
sage: t = Tachyon(xres=800,yres=800, camera_center=(2,5,2), look_at=(2.5,0,0))
sage: t.light((0,0,100), 1, (1,1,1))
sage: t.texture('r', ambient=0.1, diffuse=0.9, specular=0.5, opacity=1.0, color=(1,0,0))
sage: for i in srange(0,50,0.1):
...     t.sphere((i/10,sin(i),cos(i)), 0.05, 'r')
...
sage: t.texture('white', color=(1,1,1), opacity=1, specular=1, diffuse=1)
sage: t.plane((0,0,-100), (0,0,-100), 'white')
sage: t.show()
```

collect (*objects*)
Add a set of objects to the scene from a collection

cylinder (*center, axis, radius, texture*)

fcylinder (*base, apex, radius, texture*)

light (*center, radius, color*)

parametric_plot (*f, t_0, t_f, tex, r=0.10000000000000001, cylinders=True, min_depth=4, max_depth=8, e_rel=0.01, e_abs=0.01*)
Plots a space curve as a series of speheres and finite cylinders. Example (twisted cubic)

```
sage: f = lambda t: (t,t^2,t^3)
sage: t = Tachyon(camera_center=(5,0,4))
sage: t.texture('t')
sage: t.light((-20,-20,40), 0.2, (1,1,1))
sage: t.parametric_plot(f,-5,5,'t',min_depth=6)
```

plane (*center, normal, texture*)

plot (*f, (xmin, xmax), (ymin, ymax), texture, grad_f=None, max_bend=0.69999999999999996, max_depth=5, initial_depth=3, num_colors=None*)
INPUT:

- `f` - Function of two variables, which returns a float (or coercable to a float) (xmin,xmax)
- `(ymin,ymax)` - defines the rectangle to plot over texture: Name of texture to be used Optional arguments:
- `grad_f` - gradient function. If specified, smooth triangles will be used.
- `max_bend` - Cosine of the threshold angle between triangles used to determine whether or not to recurse after the minimum depth
- `max_depth` - maximum recursion depth. Maximum triangles plotted = 2^{2*max_depth}
- `initial_depth` - minimum recursion depth. No error-tolerance checking is performed below this depth. Minimum triangles plotted: 2^{2*min_depth}
- `num_colors` - Number of rainbow bands to color the plot with. Texture supplied will be cloned (with different colors) using the texture_recolor method of the Tachyon object.

Plots a function by constructing a mesh with nonstandard sampling density without gaps. At very high resolutions (depths 10) it becomes very slow. Cython may help. Complexity is approx. $O(2^{2*maxdepth})$. This algorithm has been optimized for speed, not memory - values from f(x,y) are recycled rather than calling the function multiple times. At high recursion depth, this may cause problems for some machines.

Flat Triangles:

```
sage: t = Tachyon(xres=512,yres=512, camera_center=(4,-4,3),viewdir=(-4,4,-3), raydepth=4)
sage: t.light((4.4,-4.4,4.4), 0.2, (1,1,1))
sage: def f(x,y): return float(sin(x*y))
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.1,  opacity=1.0, color=(1.0,0,0))
sage: t.plot(f,(-4,4),(-4,4),"t0",max_depth=5,initial_depth=3, num_colors=60)  # increase mi
sage: t.show()
```

Plotting with Smooth Triangles (requires explicit gradient function):

```
sage: t = Tachyon(xres=512,yres=512, camera_center=(4,-4,3),viewdir=(-4,4,-3), raydepth=4)
sage: t.light((4.4,-4.4,4.4), 0.2, (1,1,1))
sage: def f(x,y): return float(sin(x*y))
sage: def g(x,y): return ( float(y*cos(x*y)), float(x*cos(x*y)), 1 )
sage: t.texture('t0', ambient=0.1, diffuse=0.9, specular=0.1,  opacity=1.0, color=(1.0,0,0))
sage: t.plot(f,(-4,4),(-4,4),"t0",max_depth=5,initial_depth=3, grad_f = g)  # increase min_d
sage: t.show()
```

Preconditions: f is a scalar function of two variables, grad_f is None or a triple-valued function of two variables, min_x != max_x, min_y != max_y

```
sage: f = lambda x,y: x*y
sage: t = Tachyon()
sage: t.plot(f,(2.,2.),(-2.,2.),'')
...
ValueError: Plot rectangle is really a line.  Make sure min_x != max_x and min_y != max_y.
```

save (*filename='sage.png', verbose=0, block=True, extra_opts=''*)

INPUT:

- `filename` - (default: 'sage.png') output filename; the extension of the filename determines the type. Supported types include:
- `tga` - 24-bit (uncompressed)
- `bmp` - 24-bit Windows BMP (uncompressed)
- `ppm` - 24-bit PPM (uncompressed)
- `rgb` - 24-bit SGI RGB (uncompressed)
- `png` - 24-bit PNG (compressed, lossless)
- `verbose` - integer; (default: 0)
- `0` - silent
- `1` - some output
- `2` - very verbose output
- `block` - bool (default: True); if False, run the rendering command in the background.
- `extra_opts` - passed directly to tachyon command line. Use tachyon_rt.usage() to see some of the possibilities.

show (*verbose=0, extra_opts=''*)

smooth_triangle (*vertex_1, vertex_2, vertex_3, normal_1, normal_2, normal_3, texture*)

sphere (*center, radius, texture*)

str ()

texfunc (*type=0, center=(0, 0, 0), rotate=(0, 0, 0), scale=(1, 1, 1)*)

INPUT:

- `type` - (default: 0)
  1. No special texture, plain shading
  2. 3D checkerboard function, like a rubik's cube
  3. Grit Texture, randomized surface color

4.3D marble texture, uses object's base color
5.3D wood texture, light and dark brown, not very good yet
6.3D gradient noise function (can't remember what it looks like)
7.Don't remember
8.Cylindrical Image Map, requires ppm filename (don't know how to specify name in sage?!)
9.Spherical Image Map, requires ppm filename (don't know how to specify name in sage?!)
10.Planar Image Map, requires ppm filename (don't know how to specify name in sage?!)

- `center` - (default: (0,0,0))
- `rotate` - (default: (0,0,0))
- `scale` - (default: (1,1,1))

EXAMPLES: We draw an infinite checkboard:

```
sage: t = Tachyon(camera_center=(2,7,4), look_at=(2,0,0))
sage: t.texture('black', color=(0,0,0), texfunc=1)
sage: t.plane((0,0,0),(0,0,1),'black')
sage: t.show()
```

texture(*name, ambient=0.20000000000000001, diffuse=0.80000000000000004, specular=0.0, opacity=1.0, color=(1.0, 0.0, 0.5), texfunc=0, phong=0, phongsize=0.5, phongtype='PLASTIC'*)

INPUT:

- `name` - string; the name of the texture (to be used later)
- `ambient` - (default: 0.2)
- `diffuse` - (default: 0.8)
- `specular` - (default: 0.0)
- `opacity` - (default: 1.0)
- `color` - (default: (1.0,0.0,0.5))
- `texfunc` - (default: 0); a texture function; this is either the output of self.texfunc, or a number between 0 and 9, inclusive. See the docs for self.texfunc.
- `phong` - (default: 0)
- `phongsize` - (default: 0.5)
- `phongtype` - (default: "PLASTIC")

EXAMPLES: We draw a scene with 4 sphere that illustrates various uses of the texture command:

```
sage: t = Tachyon(camera_center=(2,5,4), look_at=(2,0,0), raydepth=6)
sage: t.light((10,3,4), 1, (1,1,1))
sage: t.texture('mirror', ambient=0.05, diffuse=0.05, specular=.9, opacity=0.9, color=(.8,.8
sage: t.texture('grey', color=(.8,.8,.8), texfunc=3)
sage: t.plane((0,0,0),(0,0,1),'grey')
sage: t.sphere((4,-1,1), 1, 'mirror')
sage: t.sphere((0,-1,1), 1, 'mirror')
sage: t.sphere((2,-1,1), 0.5, 'mirror')
sage: t.sphere((2,1,1), 0.5, 'mirror')
sage: show(t)
```

texture_recolor(*name, colors*)

triangle(*vertex_1, vertex_2, vertex_3, texture*)

class **TachyonPlot**(*tachyon, f, (min_x, max_x), (min_y, max_y), tex, g=None, min_depth=4, max_depth=8, e_rel=0.01, e_abs=0.01, num_colors=None*)

extrema(*list*)

interface(*n, p, p_c, q, q_c*)

plot_block(*min_x, mid_x, max_x, min_y, mid_y, max_y, sw_z, nw_z, se_z, ne_z, mid_z, depth*)

str()

tol(*(est, val)*)

tol_list(*l*)

triangulate(*p, c*)

class TachyonSmoothTriangle(*a, b, c, da, db, dc, color=0*)

str()

class TachyonTriangle(*a, b, c, color=0*)

str()

class TachyonTriangleFactory(*tach, tex*)

get_colors(*list*)

smooth_triangle(*a, b, c, da, db, dc, color=None*)

triangle(*a, b, c, color=None*)

class Texfunc(*type=0, center=(0, 0, 0), rotate=(0, 0, 0), scale=(1, 1, 1)*)

str()

class Texture(*name, ambient=0.20000000000000001, diffuse=0.80000000000000004, specular=0.0, opacity=1.0, color=(1.0, 0.0, 0.5), texfunc=0, phong=0, phongsize=0, phongtype='PLASTIC'*)

recolor(*name, color*)

str()

hue(*h, s=1, v=1*)

hue(h,s=1,v=1) where 'h' stands for hue, 's' stands for saturation, 'v' stands for value. hue returns a list of rgb intensities (r, g, b) All values are in the range 0 to 1.

INPUT:

- `h, s, v` - real numbers between 0 and 1. Note that if any are not in this range they are automatically normalized to be in this range by reducing them modulo 1.

OUTPUT: A valid RGB tuple.

EXAMPLES:

```
sage: hue(0.6)
(0.0, 0.40000000000000036, 1.0)
```

hue is an easy way of getting a broader range of colors for graphics

```
sage: p = plot(sin, -2, 2, rgbcolor=hue(0.6))
```

tostr(*s*)

CHAPTER

SIX

GAMES

Sage includes a simple Sudoku solver. It also has a Rubik's cube solver (see *Rubik's Cube Group*).

6.1 Sudoku Solver

Given a 9x9 Sudoku puzzle as an integer matrix, this routine finds a single solution.

grid_has_k (*A, i, j, k*)

Checks for the presence of k in the 3x3 subgrid containing the location in row i and column j.

INPUT:

- A - a 9x9 matrix with entries from 0, 1..9
- i - integer specifying row i
- j - integer specifying column j
- k - an integer from 1..9

OUTPUT:

boolean - True exactly when k is present in the 3x3 sub-grid that also has the entry in row i and column j.

EXAMPLES:

```
sage: from sage.games.sudoku import grid_has_k
sage: B = matrix(ZZ, 9, 9, [ [0,0,0,0,1,0,9,0,0], [8,0,0,4,0,0,0,0,0], [2,0,0,0,0,0,0,0,0], [0,7
sage: B
[0 0 0 0 1 0 9 0 0]
[8 0 0 4 0 0 0 0 0]
[2 0 0 0 0 0 0 0 0]
[0 7 0 0 3 0 0 0 0]
[0 0 0 0 0 0 2 0 4]
[0 0 0 0 0 0 0 5 8]
[0 6 0 0 0 0 1 3 0]
[7 0 0 2 0 0 0 0 0]
[0 0 0 8 0 0 0 0 0]
sage: grid_has_k(B, 3, 2, 7)
True
sage: grid_has_k(B, 3, 2, 1)
False
```

solve_recursive (*A, i, j*)

Completes a Sudoku puzzle starting at (an empty) square in row i and column j

INPUT:

- A - a 9x9 matrix with integer entries from 0, 1..9
- i - integer specifying row i
- j - integer specifying column j

OUTPUT:

matrix - a 9x9 matrix over ZZ

If square (i, j) is non-zero (not empty) then A is returned immediately
If there is no way to complete the puzzle then None is returned
Otherwise a recursive call is made, which will eventually return a (partial) solution or None

NOTES:

As a practical matter this should be called with (i, j) being the first empty square (in row-major order), though nothing prohibits starting at another square (empty or not), and possibly getting a partially complete square back as output.

EXAMPLES:

This puzzle has only 17 non-empty squares (hints) and has a unique solution. At this writing (2009/04/11) there are no known 16-hint Sudoku puzzles with a unique solution. It is number 3000 in Gordon Royle's database of 17-hint uniquely-solvable puzzles [1].

Here we just test finding a possible completion if we started midway through the bottom row.

```
sage: from sage.games.sudoku import solve_recursive
sage: B = matrix(ZZ, 9, 9, [ [0,0,0,0,1,0,9,0,0], [8,0,0,4,0,0,0,0,0], [2,0,0,0,0,0,0,0,0], [0,7
sage: B
[0 0 0 0 1 0 9 0 0]
[8 0 0 4 0 0 0 0 0]
[2 0 0 0 0 0 0 0 0]
[0 7 0 0 3 0 0 0 0]
[0 0 0 0 0 0 2 0 4]
[0 0 0 0 0 0 0 5 8]
[0 6 0 0 0 0 1 3 0]
[7 0 0 2 0 0 0 0 0]
[0 0 0 8 0 0 0 0 0]
sage: solve_recursive(B, 8, 5)
[0 0 0 0 1 0 9 0 0]
[8 0 0 4 0 0 0 0 0]
[2 0 0 0 0 0 0 0 0]
[0 7 0 0 3 0 0 0 0]
[0 0 0 0 0 0 2 0 4]
[0 0 0 0 0 0 0 5 8]
[0 6 0 0 0 0 1 3 0]
[7 0 0 2 0 0 0 0 0]
[0 0 0 8 0 1 4 2 5]
```

Now we perturb B to make the (8,7) square impossible to complete. And we start at an empty square a little further back up the matrix.

```
sage: B[0,7], B[1,7], B[2,7] = 2, 6, 7
sage: B[8,0], B[8, 1] = 4, 9
sage: B
[0 0 0 0 1 0 9 2 0]
[8 0 0 4 0 0 0 6 0]
[2 0 0 0 0 0 0 7 0]
[0 7 0 0 3 0 0 0 0]
[0 0 0 0 0 0 2 0 4]
```

```
[0 0 0 0 0 0 0 5 8]
[0 6 0 0 0 0 1 3 0]
[7 0 0 2 0 0 0 0 0]
[4 9 0 8 0 0 0 0 0]
sage: print solve_recursive(B, 7, 1)
None
```

REFERENCES:

- [1] Gordon Royle, Minimum Sudoku, http://people.csse.uwa.edu.au/gordon/sudokumin.php, (2009/04/11)

sudoku(A)

Solve the 9x9 Sudoku puzzle contained in the matrix A.

INPUT:

- A - a 9x9 matrix with integer entries from 0, 1..9. A 0 indicates an empty square

OUTPUT:

matrix - a 9x9 matrix over ZZ containing the first solution found

ALGORITHM:

A solution is found by examining blank cells in turn, determining which symbols are in use in the corresponding row, column and 3x3 sub-grid, and then making recursive calls exhausting all possibilities at that blank cell. Basically this is a depth-first search to the first solution, with pruning accomplished according to only the basic requirements of a legitimate completed Sudoku.

EXAMPLES:

```
sage: A = matrix(ZZ,9,[5,0,0, 0,8,0, 0,4,9, 0,0,0, 5,0,0, 0,3,0, 0,6,7, 3,0,0, 0,0,1,  1,5,0, 0,
sage: A
[5 0 0 0 8 0 0 4 9]
[0 0 0 5 0 0 0 3 0]
[0 6 7 3 0 0 0 0 1]
[1 5 0 0 0 0 0 0 0]
[0 0 0 2 0 8 0 0 0]
[0 0 0 0 0 0 0 1 8]
[7 0 0 0 0 4 1 5 0]
[0 3 0 0 0 2 0 0 0]
[4 9 0 0 5 0 0 0 3]
sage: sudoku(A)
[5 1 3 6 8 7 2 4 9]
[8 4 9 5 2 1 6 3 7]
[2 6 7 3 4 9 5 8 1]
[1 5 8 4 6 3 9 7 2]
[9 7 4 2 1 8 3 6 5]
[3 2 6 7 9 5 4 1 8]
[7 8 2 9 3 4 1 5 6]
[6 3 5 1 7 2 8 9 4]
[4 9 1 8 5 6 7 2 3]
```

Now we perturb A slightly to make the $(3, 5)$ entry impossible to complete.

```
sage: A[1,4], A[2,4] = 7, 9
sage: A[3,6], A[3,7], A[3,8] = 3, 4, 6
sage: A
[5 0 0 0 8 0 0 4 9]
[0 0 0 5 7 0 0 3 0]
[0 6 7 3 9 0 0 0 1]
```

```
[1 5 0 0 0 0 3 4 6]
[0 0 0 2 0 8 0 0 0]
[0 0 0 0 0 0 0 1 8]
[7 0 0 0 0 4 1 5 0]
[0 3 0 0 0 2 0 0 0]
[4 9 0 0 5 0 0 0 3]
sage: print sudoku(A)
None
```

CHAPTER

SEVEN

GRAPH THEORY

7.1 Graph Theory

This module implements many graph theoretic operations and concepts.

AUTHORS:

- Robert L. Miller (2006-10-22): initial version
- William Stein (2006-12-05): Editing
- Robert L. Miller (2007-01-13): refactoring, adjusting for NetworkX-0.33, fixed plotting bugs (2007-01-23): basic tutorial, edge labels, loops, multiple edges and arcs (2007-02-07): graph6 and sparse6 formats, matrix input
- Emily Kirkmann (2007-02-11): added graph_border option to plot and show
- Robert L. Miller (2007-02-12): vertex color-maps, graph boundaries, graph6 helper functions in Cython
- Robert L. Miller Sage Days 3 (2007-02-17-21): 3d plotting in Tachyon
- Robert L. Miller (2007-02-25): display a partition
- Robert L. Miller (2007-02-28): associate arbitrary objects to vertices, edge and arc label display (in 2d), edge coloring
- Robert L. Miller (2007-03-21): Automorphism group, isomorphism check, canonical label
- Robert L. Miller (2007-06-07-09): NetworkX function wrapping
- Michael W. Hansen (2007-06-09): Topological sort generation
- Emily Kirkman, Robert L. Miller Sage Days 4: Finished wrapping NetworkX
- Emily Kirkman (2007-07-21): Genus (including circular planar, all embeddings and all planar embeddings), all paths, interior paths
- Bobby Moretti (2007-08-12): fixed up plotting of graphs with edge colors differentiated by label
- Jason Grout (2007-09-25): Added functions, bug fixes, and general enhancements
- Robert L. Miller (Sage Days 7): Edge labeled graph isomorphism
- Tom Boothby (Sage Days 7): Miscellaneous awesomeness
- Tom Boothby (2008-01-09): Added graphviz output
- David Joyner (2009-2): Fixed docstring bug related to GAP.

7.1.1 Graph Format

The Sage Graph Class: NetworkX plus

Sage graphs are actually NetworkX graphs, wrapped in a Sage class. In fact, any graph can produce its underlying NetworkX graph. For example,

```
sage: import networkx
sage: G = graphs.PetersenGraph()
sage: N = G.networkx_graph()
sage: isinstance(N, networkx.graph.Graph)
True
```

The NetworkX graph is essentially a dictionary of dictionaries:

```
sage: N.adj
{0: {1: None, 4: None, 5: None}, 1: {0: None, 2: None, 6: None}, 2: {1: None, 3: None, 7: None}, 3:
```

Each dictionary key is a vertex label, and each key in the following dictionary is a neighbor of that vertex. In undirected graphs, there is redundancy: for example, the dictionary containing the entry `1: {2: None}` implies it must contain `{2: {1: None}`. The innermost entry of `None` is related to edge labeling (see section *Labels*).

Supported formats

Sage Graphs can be created from a wide range of inputs. A few examples are covered here.

- NetworkX dictionary format:

```
sage: d = {0: [1,4,5], 1: [2,6], 2: [3,7], 3: [4,8], 4: [9], \
      5: [7, 8], 6: [8,9], 7: [9]}
sage: G = Graph(d); G
Graph on 10 vertices
sage: G.plot().show()    # or G.show()
```

- A NetworkX graph:

```
sage: K = networkx.complete_bipartite_graph(12,7)
sage: G = Graph(K)
sage: G.degree()
[7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 12, 12, 12, 12, 12, 12, 12]
```

- graph6 or sparse6 format:

```
sage: s = ':I`AKGsaOs`cI]Gb~'
sage: G = Graph(s, sparse=True); G
Looped multi-graph on 10 vertices
sage: G.plot().show()    # or G.show()
```

Note that the `\` character is an escape character in Python, and also a character used by graph6 strings:

```
sage: G = Graph('Ihe\n@GUA')
...
RuntimeError: The string (Ihe) seems corrupt: for n = 10, the string is too short.
```

In Python, the escaped character `\` is represented by `\\`:

```
sage: G = Graph('Ihe\\n@GUA')
sage: G.plot().show()     # or G.show()
```

- adjacency matrix: In an adjacency matrix, each column and each row represent a vertex. If a 1 shows up in row i, column j, there is an edge (i, j).

```
sage: M = Matrix([(0,1,0,0,1,1,0,0,0,0),(1,0,1,0,0,0,1,0,0,0), \
(0,1,0,1,0,0,0,1,0,0), (0,0,1,0,1,0,0,0,1,0),(1,0,0,1,0,0,0,0,0,1), \
(1,0,0,0,0,0,0,1,1,0), (0,1,0,0,0,0,0,0,1,1),(0,0,1,0,0,1,0,0,0,1), \
(0,0,0,1,0,1,1,0,0,0), (0,0,0,0,1,0,1,1,0,0)])
sage: M
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
sage: G = Graph(M); G
Graph on 10 vertices
sage: G.plot().show()     # or G.show()
```

- incidence matrix: In an incidence matrix, each row represents a vertex and each column reprensents an edge.

```
sage: M = Matrix([(-1,0,0,0,1,0,0,0,0,0,-1,0,0,0,0), \
(1,-1,0,0,0,0,0,0,0,0,0,-1,0,0,0), (0,1,-1,0,0,0,0,0,0,0,0,0,-1,0,0), \
(0,0,1,-1,0,0,0,0,0,0,0,0,0,-1,0), (0,0,0,1,-1,0,0,0,0,0,0,0,0,0,-1), \
(0,0,0,0,0,-1,0,0,0,1,1,0,0,0,0), (0,0,0,0,0,0,0,1,-1,0,0,1,0,0,0), \
(0,0,0,0,0,1,-1,0,0,0,0,0,1,0,0), (0,0,0,0,0,0,0,0,1,-1,0,0,0,1,0), \
(0,0,0,0,0,0,1,-1,0,0,0,0,0,0,1)])
sage: M
[-1  0  0  0  1  0  0  0  0  0 -1  0  0  0  0]
[ 1 -1  0  0  0  0  0  0  0  0  0 -1  0  0  0]
[ 0  1 -1  0  0  0  0  0  0  0  0  0 -1  0  0]
[ 0  0  1 -1  0  0  0  0  0  0  0  0  0 -1  0]
[ 0  0  0  1 -1  0  0  0  0  0  0  0  0  0 -1]
[ 0  0  0  0  0 -1  0  0  0  1  1  0  0  0  0]
[ 0  0  0  0  0  0  0  1 -1  0  0  1  0  0  0]
[ 0  0  0  0  0  1 -1  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  0  0  1 -1  0  0  0  1  0]
[ 0  0  0  0  0  0  1 -1  0  0  0  0  0  0  1]
sage: G = Graph(M); G
Graph on 10 vertices
sage: G.plot().show()     # or G.show()
```

7.1.2 Generators

If you wish to iterate through all the isomorphism types of graphs, type, for example:

```
sage: for g in graphs(4):
...       print g.spectrum()
[0.0, 0.0, 0.0, 0.0]
```

```
...
[-1.0, -1.0, -1.0, 3.0]
```

For some commonly used graphs to play with, type

```
sage: graphs.[tab]          # not tested
```

and hit {tab}. Most of these graphs come with their own custom plot, so you can see how people usually visualize these graphs.

```
sage: G = graphs.PetersenGraph()
sage: G.plot().show()     # or G.show()
sage: G.degree_histogram()
[0, 0, 0, 10]
sage: G.adjacency_matrix()
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
```

```
sage: S = G.subgraph([0,1,2,3])
sage: S.plot().show()     # or S.show()
sage: S.density()
1/2
```

```
sage: G = GraphQuery(display_cols=['graph6'], num_vertices=7, diameter=5)
sage: L = G.get_graphs_list()
sage: graphs_list.show_graphs(L)
```

7.1.3 Labels

Each vertex can have any hashable object as a label. These are things like strings, numbers, and tuples. Each edge is given a default label of `None`, but if specified, edges can have any label at all. Edges between vertices u and v are represented typically as `(u, v, l)`, where `l` is the label for the edge.

Note that vertex labels themselves cannot be mutable items:

```
sage: M = Matrix( [[0,0],[0,0]] )
sage: G = Graph({ 0 : { M : None } })
...
TypeError: mutable matrices are unhashable
```

However, if one wants to define a dictionary, with the same keys and arbitrary objects for entries, one can make that association:

```
sage: d = {0 : graphs.DodecahedralGraph(), 1 : graphs.FlowerSnark(), \
      2 : graphs.MoebiusKantorGraph(), 3 : graphs.PetersenGraph() }
sage: d[2]
Moebius-Kantor Graph: Graph on 16 vertices
sage: T = graphs.TetrahedralGraph()
sage: T.vertices()
[0, 1, 2, 3]
sage: T.set_vertices(d)
sage: T.get_vertex(1)
Flower Snark: Graph on 20 vertices
```

7.1.4 Database

There is a database available for searching for graphs that satisfy a certain set of parameters, including number of vertices and edges, density, maximum and minimum degree, diameter, radius, and connectivity. To see a list of all search parameter keywords broken down by their designated table names, type

```
sage: graph_db_info()
{...}
```

For more details on datatypes or keyword input, enter

```
sage:  GraphQuery?    # not tested
```

The results of a query can be viewed with the show method, or can be viewed individually by iterating through the results:

```
sage: Q = GraphQuery(display_cols=['graph6'],num_vertices=7, diameter=5)
sage: Q.show()
Graph6
--------------------
F@?]O
F@OKg
F?`po
F?gqg
FIAHo
F@R@o
FA_pW
FGC{o
FEOhW
```

Show each graph as you iterate through the results:

```
sage: for g in Q:
...       show(g)
```

7.1.5 Visualization

To see a graph G you are working with, right now there are two main options. You can view the graph in two dimensions via matplotlib with `show()`.

```
sage: G = graphs.RandomGNP(15,.3)
sage: G.show()
```

Or you can view it in three dimensions via jmol with `show3d()`.

```
sage: G.show3d()
```

7.1.6 Graph classes and methods

class **DiGraph**(*data=None, pos=None, loops=None, format=None, boundary=, [], weighted=None, implementation='networkx', sparse=True, vertex_labels=True, **kwds*)

Directed graph.

INPUT:

- `data` - can be any of the following:
  1. A NetworkX digraph
  2. A dictionary of dictionaries
  3. A dictionary of lists
  4. A numpy matrix or ndarray
  5. A Sage adjacency matrix or incidence matrix
  6. A pygraphviz agraph
  7. A SciPy sparse matrix
- `pos` - a positioning dictionary: for example, the spring layout from NetworkX for the 5-cycle is:

```
{0: [-0.91679746, 0.88169588],
 1: [ 0.47294849, 1.125     ],
 2: [ 1.125     ,-0.12867615],
 3: [ 0.12743933,-1.125     ],
 4: [-1.125     ,-0.50118505]}
```

- `name` - (must be an explicitly named parameter, i.e., name="complete") gives the graph a name
- `loops` - boolean, whether to allow loops (ignored if data is an instance of the DiGraph class)
- `multiedges` - boolean, whether to allow multiple edges (ignored if data is an instance of the DiGraph class)
- `weighted` - whether digraph thinks of itself as weighted or not. See self.weighted()
- `format` - if None, DiGraph tries to guess- can be several values, including:
  - `'adjacency_matrix'` - a square Sage matrix M, with M[i,j] equal to the number of edges {i,j}
  - `'incidence_matrix'` - a Sage matrix, with one column C for each edge, where if C represents {i, j}, C[i] is -1 and C[j] is 1
  - `'weighted_adjacency_matrix'` - a square Sage matrix M, with M[i,j] equal to the weight of the single edge {i,j}. Given this format, weighted is ignored (assumed True).
- `boundary` - a list of boundary vertices, if none, digraph is considered as a 'digraph without boundary'
- `implementation` - what to use as a backend for the graph. Currently, the options are either 'networkx' or 'c_graph'
- `sparse` - only for implementation == 'c_graph'. Whether to use sparse or dense graphs as backend. Note that currently dense graphs do not have edge labels, nor can they be multigraphs
- `vertex_labels` - only for implementation == 'c_graph'. Whether to allow any object as a vertex (slower), or only the integers 0, ..., n-1, where n is the number of vertices.

EXAMPLES:

1.A NetworkX XDiGraph:

```
sage: import networkx
sage: g = networkx.XDiGraph({0:[1,2,3], 2:[4]})
sage: DiGraph(g)
Digraph on 5 vertices
```

2.A NetworkX digraph:

```
sage: import networkx
sage: g = networkx.DiGraph({0:[1,2,3], 2:[4]})
sage: DiGraph(g)
Digraph on 5 vertices
```

Note that in all cases, we copy the networkX structure.

```
sage: import networkx
sage: g = networkx.DiGraph({0:[1,2,3], 2:[4]})
sage: G = DiGraph(g, implementation='networkx')
sage: H = DiGraph(g, implementation='networkx')
sage: G._backend._nxg is H._backend._nxg
False
```

3.A dictionary of dictionaries:

```
sage: g = DiGraph({0:{1:'x',2:'z',3:'a'}, 2:{5:'out'}}, implementation='networkx'); g
Digraph on 5 vertices
```

The labels ('x', 'z', 'a', 'out') are labels for edges. For example, 'out' is the label for the edge from 2 to 5. Labels can be used as weights, if all the labels share some common parent.

4.A dictionary of lists:

```
sage: g = DiGraph({0:[1,2,3], 2:[4]}); g
Digraph on 5 vertices
```

5.A list of vertices and a function describing adjacencies. Note that the list of vertices and the function must be enclosed in a list (i.e., [list of vertices, function]).

We construct a graph on the integers 1 through 12 such that there is a directed edge from i to j if and only if i divides j.

```
sage: g=DiGraph([[1..12],lambda i,j: i!=j and i.divides(j)], implementation='networkx')
sage: g.vertices()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: g.adjacency_matrix()
[0 1 1 1 1 1 1 1 1 1 1 1]
[0 0 0 1 0 1 0 1 0 1 0 1]
[0 0 0 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 1]
[0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0]
```

6.A numpy matrix or ndarray:

```
sage: import numpy
sage: A = numpy.array([[0,1,0],[1,0,0],[1,1,0]])
sage: DiGraph(A, implementation='networkx')
Digraph on 3 vertices
```

7. A Sage matrix: Note: If format is not specified, then Sage assumes a square matrix is an adjacency matrix, and a nonsquare matrix is an incidence matrix.

   •an adjacency matrix:

```
sage: M = Matrix([[0, 1, 1, 1, 0],[0, 0, 0, 0, 0],[0, 0, 0, 0, 1],[0, 0, 0, 0, 0],[0, 0,
[0 1 1 1 0]
[0 0 0 0 0]
[0 0 0 0 1]
[0 0 0 0 0]
[0 0 0 0 0]
sage: DiGraph(M)
Digraph on 5 vertices
```

   •an incidence matrix:

```
sage: M = Matrix(6, [-1,0,0,0,1, 1,-1,0,0,0, 0,1,-1,0,0, 0,0,1,-1,0, 0,0,0,1,-1, 0,0,0,0
[-1  0  0  0  1]
[ 1 -1  0  0  0]
[ 0  1 -1  0  0]
[ 0  0  1 -1  0]
[ 0  0  0  1 -1]
[ 0  0  0  0  0]
sage: DiGraph(M)
Digraph on 6 vertices
```

8. A c_graph implemented DiGraph can be constructed from a networkx implemented DiGraph if its vertex set is equal to range(n):

```
sage: D = DiGraph({0:[1],1:[2],2:[0]}, implementation="networkx")
sage: E = DiGraph(D,implementation="c_graph")
sage: D == E
True
```

9. A dig6 string: Sage automatically recognizes whether a string is in dig6 format, which is a directed version of graph6:

```
sage: D = DiGraph('IRAaDCIIOWEOKcPWAo')
sage: D
Digraph on 10 vertices

sage: D = DiGraph('IRAaDCIIOEOKcPWAo')
...
RuntimeError: The string (IRAaDCIIOEOKcPWAo) seems corrupt: for n = 10, the string is too sh

sage: D = DiGraph("IRAaDCI'OWEOKcPWAo")
...
RuntimeError: The string seems corrupt: valid characters are
?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

dig6_string()

Returns the dig6 representation of the digraph as an ASCII string. Valid for single (no multiple edges) digraphs on 0 to 262143 vertices.

EXAMPLE:

```
sage: D = DiGraph()
sage: D.dig6_string()
'?'
sage: D.add_edge(0,1)
sage: D.dig6_string()
'AO'
```

graphviz_string()

Returns a representation in the DOT language, ready to render in graphviz.

REFERENCES:

- http://www.graphviz.org/doc/info/lang.html

EXAMPLE:

```
sage: G = DiGraph({0:{1:None,2:None}, 1:{2:None}, 2:{3:'foo'}, 3:{}}, implementation='networ
sage: s = G.graphviz_string(); s
'digraph {\n"0";"1";"2";"3";\n"0"->"1";"0"->"2";"1"->"2";"2"->"3"[label="foo"];\n}'
```

in_degree(*vertices=None, labels=False*)

Same as degree, but for in degree.

EXAMPLES:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: D.in_degree(vertices = [0,1,2], labels=True)
{0: 2, 1: 2, 2: 2}
sage: D.in_degree()
[2, 2, 2, 2, 1, 1]
sage: G = graphs.PetersenGraph().to_directed()
sage: G.in_degree(0)
3
```

in_degree_iterator(*vertices=None, labels=False*)

Same as degree_iterator, but for in degree.

EXAMPLES:

```
sage: D = graphs.Grid2dGraph(2,4).to_directed()
sage: for i in D.in_degree_iterator():
...     print i
3
3
2
3
2
2
2
3
sage: for i in D.in_degree_iterator(labels=True):
...     print i
((0, 1), 3)
((1, 2), 3)
((0, 0), 2)
((0, 2), 3)
((1, 3), 2)
((1, 0), 2)
((0, 3), 2)
((1, 1), 3)
```

incoming_edge_iterator(*vertices=None, labels=True*)

Return an iterator over all arriving edges from vertices, or over all edges if vertices is None.

EXAMPLE:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: for a in D.incoming_edge_iterator([0]):
...     print a
(1, 0, None)
(4, 0, None)
```

incoming_edges(*vertices=None, labels=True*)

Returns a list of edges arriving at vertices.

INPUT:

- `labels` - if False, each edge is a tuple (u,v) of vertices.

EXAMPLE:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: D.incoming_edges([0])
[(1, 0, None), (4, 0, None)]
```

is_directed()

Since digraph is directed, returns True.

EXAMPLE:

```
sage: DiGraph().is_directed()
True
```

is_directed_acyclic()

Returns whether the digraph is acyclic or not.

A directed graph is acyclic if for any vertex v, there is no directed path that starts and ends at v. Every directed acyclic graph (dag) corresponds to a partial ordering of its vertices, however multiple dags may lead to the same partial ordering.

EXAMPLES:

```
sage: D = DiGraph({ 0:[1,2,3], 4:[2,5], 1:[8], 2:[7], 3:[7], 5:[6,7], 7:[8], 6:[9], 8:[10],
sage: D.plot(layout='circular').show()
sage: D.is_directed_acyclic()
True

sage: D.add_edge(9,7)
sage: D.is_directed_acyclic()
True

sage: D.add_edge(7,4)
sage: D.is_directed_acyclic()
False
```

out_degree(*vertices=None, labels=False*)

Same as degree, but for out degree.

EXAMPLES:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: D.out_degree(vertices = [0,1,2], labels=True)
{0: 3, 1: 2, 2: 1}
sage: D.out_degree()
[3, 2, 1, 1, 2, 1]
```

out_degree_iterator(*vertices=None, labels=False*)

Same as degree_iterator, but for out degree.

EXAMPLES:

```
sage: D = graphs.Grid2dGraph(2,4).to_directed()
sage: for i in D.out_degree_iterator():
...    print i
3
3
2
3
2
2
2
3
sage: for i in D.out_degree_iterator(labels=True):
...    print i
((0, 1), 3)
((1, 2), 3)
((0, 0), 2)
((0, 2), 3)
((1, 3), 2)
((1, 0), 2)
((0, 3), 2)
((1, 1), 3)
```

outgoing_edge_iterator(*vertices=None, labels=True*)

Return an iterator over all departing edges from vertices, or over all edges if vertices is None.

EXAMPLE:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: for a in D.outgoing_edge_iterator([0]):
...    print a
(0, 1, None)
(0, 2, None)
(0, 3, None)
```

outgoing_edges(*vertices=None, labels=True*)

Returns a list of edges departing from vertices.

INPUT:

- `labels` - if False, each edge is a tuple (u,v) of vertices.

EXAMPLE:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: D.outgoing_edges([0])
[(0, 1, None), (0, 2, None), (0, 3, None)]
```

predecessor_iterator(*vertex*)

Returns an iterator over predecessor vertices of vertex.

EXAMPLE:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: for a in D.predecessor_iterator(0):
...    print a
1
4
```

predecessors(*vertex*)

Returns a list of predecessor vertices of vertex.

EXAMPLE:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: D.predecessors(0)
[1, 4]
```

reverse()

Returns a copy of digraph with edges reversed in direction.

EXAMPLES:

```
sage: D = DiGraph({ 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] })
sage: D.reverse()
Reverse of (): Digraph on 6 vertices
```

strongly_connected_components()

Returns a list of lists of vertices, each list representing a strongly connected component.

EXAMPLES:

```
sage: D = DiGraph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: D.connected_components()
[[0, 1, 2, 3], [4, 5, 6]]
sage: D = DiGraph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: D.strongly_connected_components()
[[3], [2], [1], [0], [6], [5], [4]]
sage: D.add_edge([2,0])
sage: D.strongly_connected_components()
[[0, 1, 2], [3], [6], [5], [4]]
```

successor_iterator(*vertex*)

Returns an iterator over successor vertices of vertex.

EXAMPLE:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: for a in D.successor_iterator(0):
...    print a
1
2
3
```

successors(*vertex*)

Returns a list of successor vertices of vertex.

EXAMPLE:

```
sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: D.successors(0)
[1, 2, 3]
```

to_directed()

Since the graph is already directed, simply returns a copy of itself.

EXAMPLE:

```
sage: DiGraph({0:[1,2,3],4:[5,1]}).to_directed()
Digraph on 6 vertices
```

to_undirected(*implementation='networkx'*)

Returns an undirected version of the graph. Every directed edge becomes an edge.

EXAMPLE:

```
sage: D = DiGraph({0:[1,2],1:[0]})
sage: G = D.to_undirected()
sage: D.edges(labels=False)
```

```
[(0, 1), (0, 2), (1, 0)]
sage: G.edges(labels=False)
[(0, 1), (0, 2)]
```

topological_sort()

Returns a topological sort of the digraph if it is acyclic, and raises a TypeError if the digraph contains a directed cycle.

A topological sort is an ordering of the vertices of the digraph such that each vertex comes before all of its successors. That is, if u comes before v in the sort, then there may be a directed path from u to v, but there will be no directed path from v to u.

EXAMPLES:

```
sage: D = DiGraph({ 0:[1,2,3], 4:[2,5], 1:[8], 2:[7], 3:[7], 5:[6,7], 7:[8], 6:[9], 8:[10],
sage: D.plot(layout='circular').show()
sage: D.topological_sort()
[4, 5, 6, 9, 0, 1, 2, 3, 7, 8, 10]

sage: D.add_edge(9,7)
sage: D.topological_sort()
[4, 5, 6, 9, 0, 1, 2, 3, 7, 8, 10]

sage: D.add_edge(7,4)
sage: D.topological_sort()
...
TypeError: Digraph is not acyclic-- there is no topological sort.
```

Note: There is a recursive version of this in NetworkX, but it has problems:

```
sage: import networkx
sage: D = DiGraph({ 0:[1,2,3], 4:[2,5], 1:[8], 2:[7], 3:[7], 5:[6,7], 7:[8], 6:[9], 8:[10],
sage: N = D.networkx_graph()
sage: networkx.topological_sort(N)
[4, 5, 6, 9, 0, 1, 2, 3, 7, 8, 10]
sage: networkx.topological_sort_recursive(N) is None
True
```

topological_sort_generator()

Returns a list of all topological sorts of the digraph if it is acyclic, and raises a TypeError if the digraph contains a directed cycle.

A topological sort is an ordering of the vertices of the digraph such that each vertex comes before all of its successors. That is, if u comes before v in the sort, then there may be a directed path from u to v, but there will be no directed path from v to u. See also Graph.topological_sort().

AUTHORS:

- Mike Hansen - original implementation
- Robert L. Miller: wrapping, documentation

REFERENCE:

- [1] Pruesse, Gara and Ruskey, Frank. Generating Linear Extensions Fast. SIAM J. Comput., Vol. 23 (1994), no. 2, pp. 373-386.

EXAMPLES:

```
sage: D = DiGraph({ 0:[1,2], 1:[3], 2:[3,4] })
sage: D.plot(layout='circular').show()
sage: D.topological_sort_generator()
[[0, 1, 2, 3, 4], [0, 1, 2, 4, 3], [0, 2, 1, 3, 4], [0, 2, 1, 4, 3], [0, 2, 4, 1, 3]]
```

```
sage: for sort in D.topological_sort_generator():
...       for edge in D.edge_iterator():
...           u,v,l = edge
...           if sort.index(u) > sort.index(v):
...               print "This should never happen."
```

class GenericGraph()

Base class for graphs and digraphs.

add_cycle(*vertices*)

Adds a cycle to the graph with the given vertices. If the vertices are already present, only the edges are added.

For digraphs, adds the directed cycle, whose orientation is determined by the list. Adds edges (vertices[u], vertices[u+1]) and (vertices[-1], vertices[0]).

INPUT:

- `vertices` - a list of indices for the vertices of the cycle to be added.

EXAMPLES:

```
sage: G = Graph(implementation='networkx')
sage: G.add_vertices(range(10)); G
Graph on 10 vertices
sage: show(G)
sage: G.add_cycle(range(20)[10:20])
sage: show(G)
sage: G.add_cycle(range(10))
sage: show(G)

sage: D = DiGraph(implementation='networkx')
sage: D.add_cycle(range(4))
sage: D.edges()
[(0, 1, None), (1, 2, None), (2, 3, None), (3, 0, None)]
```

add_edge(*u, v=None, label=None*)

Adds an edge from u and v.

INPUT: The following forms are all accepted:

- G.add_edge(1, 2)
- G.add_edge((1, 2))
- G.add_edges([(1, 2)])
- G.add_edge(1, 2, 'label')
- G.add_edge((1, 2, 'label'))
- G.add_edges([(1, 2, 'label')])

WARNING: The following intuitive input results in nonintuitive output:

```
sage: G = Graph(implementation='networkx')
sage: G.add_edge((1,2), 'label')
sage: G.networkx_graph().adj           # random output order
{'label': {(1, 2): None}, (1, 2): {'label': None}}
```

Use one of these instead:

```
sage: G = Graph(implementation='networkx')
sage: G.add_edge((1,2), label="label")
sage: G.networkx_graph().adj           # random output order
{1: {2: 'label'}, 2: {1: 'label'}}
```

```
sage: G = Graph(implementation='networkx')
sage: G.add_edge(1,2,'label')
sage: G.networkx_graph().adj            # random output order
{1: {2: 'label'}, 2: {1: 'label'}}
```

add_edges(*edges*)

Add edges from an iterable container.

EXAMPLE:

```
sage: G = graphs.DodecahedralGraph()
sage: H = Graph(implementation='networkx')
sage: H.add_edges( G.edge_iterator() ); H
Graph on 20 vertices
sage: G = graphs.DodecahedralGraph().to_directed()
sage: H = DiGraph(implementation='networkx')
sage: H.add_edges( G.edge_iterator() ); H
Digraph on 20 vertices
```

add_path(*vertices*)

Adds a cycle to the graph with the given vertices. If the vertices are already present, only the edges are added.

For digraphs, adds the directed path vertices[0], ..., vertices[-1].

INPUT:

- `vertices` - a list of indices for the vertices of the cycle to be added.

EXAMPLES:

```
sage: G = Graph(implementation='networkx')
sage: G.add_vertices(range(10)); G
Graph on 10 vertices
sage: show(G)
sage: G.add_path(range(20)[10:20])
sage: show(G)
sage: G.add_path(range(10))
sage: show(G)

sage: D = DiGraph(sparse=True)
sage: D.add_path(range(4))
sage: D.edges()
[(0, 1, None), (1, 2, None), (2, 3, None)]
```

add_vertex(*name=None*)

Creates an isolated vertex. If the vertex already exists, then nothing is done.

INPUT:

- `n` - Name of the new vertex. If no name is specified, then the vertex will be represented by the least integer not already representing a vertex. Name must be an immutable object.

As it is implemented now, if a graph G has a large number of vertices with numeric labels, then G.add_vertex() could potentially be slow.

EXAMPLES:

```
sage: G = Graph(sparse=True); G.add_vertex(); G
Graph on 1 vertex

sage: D = DiGraph(sparse=True); D.add_vertex(); D
Digraph on 1 vertex
```

add_vertices(*vertices*)

Add vertices to the (di)graph from an iterable container of vertices. Vertices that already exist in the graph will not be added again.

EXAMPLES:

```
sage: d = {0: [1,4,5], 1: [2,6], 2: [3,7], 3: [4,8], 4: [9], 5: [7,8], 6: [8,9], 7: [9]}
sage: G = Graph(d, sparse=True)
sage: G.add_vertices([10,11,12])
sage: G.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: G.add_vertices(graphs.CycleGraph(25).vertices())
sage: G.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24]
```

adjacency_matrix(*sparse=None, boundary_first=False*)

Returns the adjacency matrix of the (di)graph. Each vertex is represented by its position in the list returned by the vertices() function.

The matrix returned is over the integers. If a different ring is desired, use either the change_ring function or the matrix function.

INPUT:

- `sparse` - whether to represent with a sparse matrix
- `boundary_first` - whether to represent the boundary vertices in the upper left block

EXAMPLES:

```
sage: G = graphs.CubeGraph(4)
sage: G.adjacency_matrix()
[0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0]
[1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0]
[1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0]
[0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0]
[1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0]
[0 1 0 0 1 0 0 1 0 0 0 0 0 1 0 0]
[0 0 1 0 1 0 0 1 0 0 0 0 0 0 1 0]
[0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0]
[0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0]
[0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0]
[0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1]
[0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 0]
[0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0]

sage: matrix(GF(2),G) # matrix over GF(2)
[0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0]
[1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0]
[1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0]
[0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0]
[1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0]
[0 1 0 0 1 0 0 1 0 0 0 0 0 1 0 0]
[0 0 1 0 1 0 0 1 0 0 0 0 0 0 1 0]
[0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0]
[0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0]
[0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0]
[0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1]
[0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 0]
```

```
[0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0]

sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: D.adjacency_matrix()
[0 1 1 1 0 0]
[1 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[1 0 0 0 0 1]
[0 1 0 0 0 0]
```

TESTS:

```
sage: graphs.CubeGraph(8).adjacency_matrix().parent()
Full MatrixSpace of 256 by 256 dense matrices over Integer Ring
sage: graphs.CubeGraph(9).adjacency_matrix().parent()
Full MatrixSpace of 512 by 512 sparse matrices over Integer Ring
```

all_paths (*start, end*)

Returns a list of all paths (also lists) between a pair of vertices (start, end) in the (di)graph.

EXAMPLES:

```
sage: eg1 = Graph({0:[1,2], 1:[4], 2:[3,4], 4:[5], 5:[6]})
sage: eg1.all_paths(0,6)
[[0, 1, 4, 5, 6], [0, 2, 4, 5, 6]]
sage: eg2 = graphs.PetersenGraph()
sage: sorted(eg2.all_paths(1,4))
[[1, 0, 4],
 [1, 0, 5, 7, 2, 3, 4],
 [1, 0, 5, 7, 2, 3, 8, 6, 9, 4],
 [1, 0, 5, 7, 9, 4],
 [1, 0, 5, 7, 9, 6, 8, 3, 4],
 [1, 0, 5, 8, 3, 2, 7, 9, 4],
 [1, 0, 5, 8, 3, 4],
 [1, 0, 5, 8, 6, 9, 4],
 [1, 0, 5, 8, 6, 9, 7, 2, 3, 4],
 [1, 2, 3, 4],
 [1, 2, 3, 8, 5, 0, 4],
 [1, 2, 3, 8, 5, 7, 9, 4],
 [1, 2, 3, 8, 6, 9, 4],
 [1, 2, 3, 8, 6, 9, 7, 5, 0, 4],
 [1, 2, 7, 5, 0, 4],
 [1, 2, 7, 5, 8, 3, 4],
 [1, 2, 7, 5, 8, 6, 9, 4],
 [1, 2, 7, 9, 4],
 [1, 2, 7, 9, 6, 8, 3, 4],
 [1, 2, 7, 9, 6, 8, 5, 0, 4],
 [1, 6, 8, 3, 2, 7, 5, 0, 4],
 [1, 6, 8, 3, 2, 7, 9, 4],
 [1, 6, 8, 3, 4],
 [1, 6, 8, 5, 0, 4],
 [1, 6, 8, 5, 7, 2, 3, 4],
 [1, 6, 8, 5, 7, 9, 4],
 [1, 6, 9, 4],
 [1, 6, 9, 7, 2, 3, 4],
 [1, 6, 9, 7, 2, 3, 8, 5, 0, 4],
 [1, 6, 9, 7, 5, 0, 4],
```

```
 [1, 6, 9, 7, 5, 8, 3, 4]]
sage: dg = DiGraph({0:[1,3], 1:[3], 2:[0,3]})
sage: sorted(dg.all_paths(0,3))
[[0, 1, 3], [0, 3]]
sage: ug = dg.to_undirected()
sage: sorted(ug.all_paths(0,3))
[[0, 1, 3], [0, 2, 3], [0, 3]]
```

allow_loops (*new*)

Changes whether loops are permitted in the (di)graph.

INPUT:

- `new` - boolean.

EXAMPLES:

```
sage: G = Graph(loops=True); G
Looped graph on 0 vertices
sage: G.has_loops()
False
sage: G.allows_loops()
True
sage: G.add_edge((0,0))
sage: G.has_loops()
True
sage: G.loops()
[(0, 0, None)]
sage: G.allow_loops(False); G
Graph on 1 vertex
sage: G.has_loops()
False
sage: G.edges()
[]

sage: D = DiGraph(loops=True); D
Looped digraph on 0 vertices
sage: D.has_loops()
False
sage: D.allows_loops()
True
sage: D.add_edge((0,0))
sage: D.has_loops()
True
sage: D.loops()
[(0, 0, None)]
sage: D.allow_loops(False); D
Digraph on 1 vertex
sage: D.has_loops()
False
sage: D.edges()
[]
```

allow_multiple_edges (*new*)

Changes whether multiple edges are permitted in the (di)graph.

INPUT:

- `new` - boolean.

EXAMPLES:

```
sage: G = Graph(multiedges=True); G
Multi-graph on 0 vertices
sage: G.has_multiple_edges()
False
sage: G.allows_multiple_edges()
True
sage: G.add_edges([(0,1)]*3)
sage: G.has_multiple_edges()
True
sage: G.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: G.allow_multiple_edges(False); G
Graph on 2 vertices
sage: G.has_multiple_edges()
False
sage: G.edges()
[(0, 1, None)]

sage: D = DiGraph(multiedges=True); D
Multi-digraph on 0 vertices
sage: D.has_multiple_edges()
False
sage: D.allows_multiple_edges()
True
sage: D.add_edges([(0,1)]*3)
sage: D.has_multiple_edges()
True
sage: D.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: D.allow_multiple_edges(False); D
Digraph on 2 vertices
sage: D.has_multiple_edges()
False
sage: D.edges()
[(0, 1, None)]
```

allows_loops()

Returns whether loops are permitted in the (di)graph.

EXAMPLES:

```
sage: G = Graph(loops=True); G
Looped graph on 0 vertices
sage: G.has_loops()
False
sage: G.allows_loops()
True
sage: G.add_edge((0,0))
sage: G.has_loops()
True
sage: G.loops()
[(0, 0, None)]
sage: G.allow_loops(False); G
Graph on 1 vertex
sage: G.has_loops()
False
sage: G.edges()
[]
```

```
sage: D = DiGraph(loops=True); D
Looped digraph on 0 vertices
sage: D.has_loops()
False
sage: D.allows_loops()
True
sage: D.add_edge((0,0))
sage: D.has_loops()
True
sage: D.loops()
[(0, 0, None)]
sage: D.allow_loops(False); D
Digraph on 1 vertex
sage: D.has_loops()
False
sage: D.edges()
[]
```

allows_multiple_edges()

Returns whether multiple edges are permitted in the (di)graph.

EXAMPLES:

```
sage: G = Graph(multiedges=True); G
Multi-graph on 0 vertices
sage: G.has_multiple_edges()
False
sage: G.allows_multiple_edges()
True
sage: G.add_edges([(0,1)]*3)
sage: G.has_multiple_edges()
True
sage: G.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: G.allow_multiple_edges(False); G
Graph on 2 vertices
sage: G.has_multiple_edges()
False
sage: G.edges()
[(0, 1, None)]

sage: D = DiGraph(multiedges=True); D
Multi-digraph on 0 vertices
sage: D.has_multiple_edges()
False
sage: D.allows_multiple_edges()
True
sage: D.add_edges([(0,1)]*3)
sage: D.has_multiple_edges()
True
sage: D.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: D.allow_multiple_edges(False); D
Digraph on 2 vertices
sage: D.has_multiple_edges()
False
sage: D.edges()
[(0, 1, None)]
```

am(*sparse=None, boundary_first=False*)

Returns the adjacency matrix of the (di)graph. Each vertex is represented by its position in the list returned by the vertices() function.

The matrix returned is over the integers. If a different ring is desired, use either the change_ring function or the matrix function.

INPUT:

- `sparse` - whether to represent with a sparse matrix
- `boundary_first` - whether to represent the boundary vertices in the upper left block

EXAMPLES:

```
sage: G = graphs.CubeGraph(4)
sage: G.adjacency_matrix()
[0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0]
[1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0]
[1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0]
[0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0]
[1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0]
[0 1 0 0 1 0 0 1 0 0 0 0 0 1 0 0]
[0 0 1 0 1 0 0 1 0 0 0 0 0 0 1 0]
[0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0]
[0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0]
[0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0]
[0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1]
[0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 0]
[0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0]

sage: matrix(GF(2),G) # matrix over GF(2)
[0 1 1 0 1 0 0 0 1 0 0 0 0 0 0 0]
[1 0 0 1 0 1 0 0 0 1 0 0 0 0 0 0]
[1 0 0 1 0 0 1 0 0 0 1 0 0 0 0 0]
[0 1 1 0 0 0 0 1 0 0 0 1 0 0 0 0]
[1 0 0 0 0 1 1 0 0 0 0 0 1 0 0 0]
[0 1 0 0 1 0 0 1 0 0 0 0 0 1 0 0]
[0 0 1 0 1 0 0 1 0 0 0 0 0 0 1 0]
[0 0 0 1 0 1 1 0 0 0 0 0 0 0 0 1]
[1 0 0 0 0 0 0 0 0 1 1 0 1 0 0 0]
[0 1 0 0 0 0 0 0 1 0 0 1 0 1 0 0]
[0 0 1 0 0 0 0 0 1 0 0 1 0 0 1 0]
[0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1]
[0 0 0 0 1 0 0 0 1 0 0 0 0 1 1 0]
[0 0 0 0 0 1 0 0 0 1 0 0 1 0 0 1]
[0 0 0 0 0 0 1 0 0 0 1 0 1 0 0 1]
[0 0 0 0 0 0 0 1 0 0 0 1 0 1 1 0]

sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: D.adjacency_matrix()
[0 1 1 1 0 0]
[1 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[1 0 0 0 0 1]
[0 1 0 0 0 0]
```

TESTS:

```
sage: graphs.CubeGraph(8).adjacency_matrix().parent()
Full MatrixSpace of 256 by 256 dense matrices over Integer Ring
sage: graphs.CubeGraph(9).adjacency_matrix().parent()
Full MatrixSpace of 512 by 512 sparse matrices over Integer Ring
```

antisymmetric()

Returns True if the relation given by the graph is antisymmetric and False otherwise.

A graph represents an antisymmetric relation if there being a path from a vertex x to a vertex y implies that there is not a path from y to x unless x=y.

A directed acyclic graph is antisymmetric. An undirected graph is never antisymmetric unless it is just a union of isolated vertices.

```
sage: graphs.RandomGNP(20,0.5).antisymmetric()
False
sage: digraphs.RandomDirectedGNR(20,0.5).antisymmetric()
True
```

automorphism_group(*partition=None, translation=False, verbosity=0, edge_labels=False, order=False, return_group=True, orbits=False*)

Returns the largest subgroup of the automorphism group of the (di)graph whose orbit partition is finer than the partition given. If no partition is given, the unit partition is used and the entire automorphism group is given.

INPUT:

- `translation` - if True, then output includes a a dictionary translating from keys == vertices to entries == elements of 1,2,...,n (since permutation groups can currently only act on positive integers).
- `partition` - default is the unit partition, otherwise computes the subgroup of the full automorphism group respecting the partition.
- `edge_labels` - default False, otherwise allows only permutations respecting edge labels.
- `order` - (default False) if True, compute the order of the automorphism group
- `return_group` - default True
- `orbits` - returns the orbits of the group acting on the vertices of the graph

OUTPUT: The order of the output is group, translation, order, orbits. However, there are options to turn each of these on or off.

EXAMPLES: Graphs:

```
sage: graphs_query = GraphQuery(display_cols=['graph6'],num_vertices=4)
sage: L = graphs_query.get_graphs_list()
sage: graphs_list.show_graphs(L)
sage: for g in L:
...     G = g.automorphism_group()
...     G.order(), G.gens()
(24, [(2,3), (1,2), (1,4)])
(4, [(2,3), (1,4)])
(2, [(1,2)])
(8, [(1,2), (1,4)(2,3)])
(6, [(1,2), (1,4)])
(6, [(2,3), (1,2)])
(2, [(1,4)(2,3)])
(2, [(1,2)])
(8, [(2,3), (1,3)(2,4), (1,4)])
(4, [(2,3), (1,4)])
(24, [(2,3), (1,2), (1,4)])
sage: C = graphs.CubeGraph(4)
sage: G = C.automorphism_group()
sage: M = G.character_table()
```

```
sage: M.determinant()
-712483534798848
sage: G.order()
384

sage: D = graphs.DodecahedralGraph()
sage: G = D.automorphism_group()
sage: A5 = AlternatingGroup(5)
sage: Z2 = CyclicPermutationGroup(2)
sage: H = A5.direct_product(Z2)[0] #see documentation for direct_product to explain the [0]
sage: G.is_isomorphic(H)
True
```

Multigraphs:

```
sage: G = Graph(multiedges=True, implementation='networkx')
sage: G.add_edge(('a', 'b'))
sage: G.add_edge(('a', 'b'))
sage: G.add_edge(('a', 'b'))
sage: G.automorphism_group()
Permutation Group with generators [(1,2)]
```

Digraphs:

```
sage: D = DiGraph( { 0:[1], 1:[2], 2:[3], 3:[4], 4:[0] } )
sage: D.automorphism_group()
Permutation Group with generators [(1,2,3,4,5)]
```

Edge labeled graphs:

```
sage: G = Graph(implementation='networkx')
sage: G.add_edges( [(0,1,'a'),(1,2,'b'),(2,3,'c'),(3,4,'b'),(4,0,'a')] )
sage: G.automorphism_group(edge_labels=True)
Permutation Group with generators [(1,4)(2,3)]

sage: G = Graph({0 : {1 : 7}})
sage: G.automorphism_group(translation=True, edge_labels=True)
(Permutation Group with generators [(1,2)], {0: 2, 1: 1})

sage: foo = Graph()
sage: bar = Graph(implementation='c_graph')
sage: foo.add_edges([(0,1,1),(1,2,2), (2,3,3)])
sage: bar.add_edges([(0,1,1),(1,2,2), (2,3,3)])
sage: foo.automorphism_group(translation=True, edge_labels=True)
(Permutation Group with generators [()], {0: 4, 1: 1, 2: 2, 3: 3})
sage: foo.automorphism_group(translation=True)
(Permutation Group with generators [(1,2)(3,4)], {0: 4, 1: 1, 2: 2, 3: 3})
sage: bar.automorphism_group(translation=True, edge_labels=True)
(Permutation Group with generators [()], {0: 4, 1: 1, 2: 2, 3: 3})
sage: bar.automorphism_group(translation=True)
(Permutation Group with generators [(1,2)(3,4)], {0: 4, 1: 1, 2: 2, 3: 3})
```

You can also ask for just the order of the group:

```
sage: G = graphs.PetersenGraph()
sage: G.automorphism_group(return_group=False, order=True)
120
```

Or, just the orbits (recall the Petersen graph is transitive!)

```
sage: G = graphs.PetersenGraph()
sage: G.automorphism_group(return_group=False, orbits=True)
[[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]]
```

blocks_and_cut_vertices()

Computes the blocks and cut vertices of the graph. In the case of a digraph, this computation is done on the underlying graph.

A cut vertex is one whose deletion increases the number of connected components. A block is a maximal induced subgraph which itself has no cut vertices. Two distinct blocks cannot overlap in more than a single cut vertex.

OUTPUT: (B, C), where B is a list of blocks- each is a list of vertices and the blocks are the corresponding induced subgraphs- and C is a list of cut vertices.

EXAMPLES:

```
sage: graphs.PetersenGraph().blocks_and_cut_vertices()
([[0, 1, 2, 3, 8, 5, 7, 9, 4, 6]], [])
sage: graphs.PathGraph(6).blocks_and_cut_vertices()
([[5, 4], [4, 3], [3, 2], [2, 1], [0, 1]], [4, 3, 2, 1])
sage: graphs.CycleGraph(7).blocks_and_cut_vertices()
([[0, 1, 2, 3, 4, 5, 6]], [])
sage: graphs.KrackhardtKiteGraph().blocks_and_cut_vertices()
([[9, 8], [8, 7], [0, 1, 3, 2, 5, 6, 4, 7]], [8, 7])
```

ALGORITHM: 8.3.8 in [1]. Notice the typo - the stack must also be considered as one of the blocks at termination.

REFERENCE:

- [1] D. Jungnickel, Graphs, Networks and Algorithms, Springer, 2005.

breadth_first_search(*u, ignore_direction=False*)

Returns an iterator over vertices in a breadth-first ordering.

INPUT:

- `u` - vertex at which to start search
- `ignore_direction` - (default False) only applies to directed graphs. If True, searches across edges in either direction.

EXAMPLES:

```
sage: G = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} } )
sage: list(G.breadth_first_search(0))
[0, 1, 4, 2, 3]
sage: list(G.depth_first_search(0))
[0, 4, 3, 2, 1]

sage: D = DiGraph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} } )
sage: list(D.breadth_first_search(0))
[0, 1, 2, 3, 4]
sage: list(D.depth_first_search(0))
[0, 1, 2, 3, 4]

sage: D = DiGraph({1:[0], 2:[0]})
sage: list(D.breadth_first_search(0))
[0]
sage: list(D.breadth_first_search(0, ignore_direction=True))
[0, 1, 2]
```

canonical_label(*partition=None, certify=False, verbosity=0, edge_labels=False*)

Returns the canonical label with respect to the partition. If no partition is given, uses the unit partition.

INPUT:

- `partition` - if given, the canonical label with respect to this partition will be computed. The default is the unit partition.
- `certify` - if True, a dictionary mapping from the (di)graph to its canonical label will be given.
- `verbosity` - gets passed to nice: prints helpful output.
- `edge_labels` - default False, otherwise allows only permutations respecting edge labels.

EXAMPLE:

```
sage: D = graphs.DodecahedralGraph()
sage: E = D.canonical_label(); E
Dodecahedron: Graph on 20 vertices
sage: D.canonical_label(certify=True)
(Dodecahedron: Graph on 20 vertices, {0: 0, 1: 19, 2: 16, 3: 15, 4: 9, 5: 1, 6: 10, 7: 8, 8:
sage: D.is_isomorphic(E)
True
```

Multigraphs:

```
sage: G = Graph(multiedges=True)
sage: G.add_edge((0,1))
sage: G.add_edge((0,1))
sage: G.add_edge((0,1))
sage: G.canonical_label()
Multi-graph on 2 vertices
```

Digraphs:

```
sage: P = graphs.PetersenGraph()
sage: DP = P.to_directed()
sage: DP.canonical_label().adjacency_matrix()
[0 0 0 0 0 0 0 1 1 1]
[0 0 0 0 1 0 1 0 0 1]
[0 0 0 1 0 0 1 0 1 0]
[0 0 1 0 0 1 0 0 0 1]
[0 1 0 0 0 1 0 0 1 0]
[0 0 0 1 1 0 0 1 0 0]
[0 1 1 0 0 0 0 1 0 0]
[1 0 0 0 0 1 1 0 0 0]
[1 0 1 0 1 0 0 0 0 0]
[1 1 0 1 0 0 0 0 0 0]
```

Edge labeled graphs:

```
sage: G = Graph(implementation='networkx')
sage: G.add_edges( [(0,1,'a'),(1,2,'b'),(2,3,'c'),(3,4,'b'),(4,0,'a')] )
sage: G.canonical_label(edge_labels=True)
Graph on 5 vertices
```

cartesian_product(*other*)

Returns the Cartesian product of self and other.

The Cartesian product of G and H is the graph L with vertex set V(L) equal to the Cartesian product of the vertices V(G) and V(H), and ((u,v), (w,x)) is an edge iff either - (u, w) is an edge of self and v = x, or - (v, x) is an edge of other and u = w.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: C = graphs.CycleGraph(5)
sage: P = C.cartesian_product(Z); P
Graph on 10 vertices
sage: P.plot().show()
```

```
sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: C = D.cartesian_product(P); C
Graph on 200 vertices
sage: C.plot().show()
```

categorical_product(*other*)

Returns the tensor product, also called the categorical product, of self and other.

The tensor product of G and H is the graph L with vertex set V(L) equal to the Cartesian product of the vertices V(G) and V(H), and ((u,v), (w,x)) is an edge iff - (u, w) is an edge of self, and - (v, x) is an edge of other.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: C = graphs.CycleGraph(5)
sage: T = C.tensor_product(Z); T
Graph on 10 vertices
sage: T.plot().show()

sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: T = D.tensor_product(P); T
Graph on 200 vertices
sage: T.plot().show()
```

center()

Returns the set of vertices in the center, i.e. whose eccentricity is equal to the radius of the (di)graph.

In other words, the center is the set of vertices achieving the minimum eccentricity.

EXAMPLES:

```
sage: G = graphs.DiamondGraph()
sage: G.center()
[1, 2]
sage: P = graphs.PetersenGraph()
sage: P.subgraph(P.center()) == P
True
sage: S = graphs.StarGraph(19)
sage: S.center()
[0]
sage: G = Graph(sparse=True)
sage: G.center()
[]
sage: G.add_vertex()
sage: G.center()
[0]
```

characteristic_polynomial(*var='x', laplacian=False*)

Returns the characteristic polynomial of the adjacency matrix of the (di)graph.

INPUT:

- `laplacian` - if True, use the Laplacian matrix instead (see self.kirchhoff_matrix())

EXAMPLE:

```
sage: P = graphs.PetersenGraph()
sage: P.characteristic_polynomial()
x^10 - 15*x^8 + 75*x^6 - 24*x^5 - 165*x^4 + 120*x^3 + 120*x^2 - 160*x + 48
sage: P.characteristic_polynomial(laplacian=True)
x^10 - 30*x^9 + 390*x^8 - 2880*x^7 + 13305*x^6 - 39882*x^5 + 77640*x^4 - 94800*x^3 + 66000*x
```

check_embedding_validity(*embedding=None*)

Checks whether an _embedding attribute is defined on self and if so, checks for accuracy. Returns True if everything is okay, False otherwise.

If embedding=None will test the attribute _embedding.

EXAMPLES:

```
sage: d = {0: [1, 5, 4], 1: [0, 2, 6], 2: [1, 3, 7], 3: [8, 2, 4], 4: [0, 9, 3], 5: [0, 8, 7
sage: G = graphs.PetersenGraph()
sage: G.check_embedding_validity(d)
True
```

clear()

Empties the graph of vertices and edges and removes name, boundary, associated objects, and position information.

EXAMPLE:

```
sage: G=graphs.CycleGraph(4); G.set_vertices({0:'vertex0'})
sage: G.order(); G.size()
4
4
sage: len(G._pos)
4
sage: G.name()
'Cycle graph'
sage: G.get_vertex(0)
'vertex0'
sage: H = G.copy(implementation='c_graph', sparse=True)
sage: H.clear()
sage: H.order(); H.size()
0
0
sage: len(H._pos)
0
sage: H.name()
''
sage: H.get_vertex(0)
sage: H = G.copy(implementation='networkx')
sage: H.clear()
sage: H.order(); H.size()
0
0
sage: len(H._pos)
0
sage: H.name()
''
sage: H.get_vertex(0)
```

cluster_transitivity()

Returns the transitivity (fraction of transitive triangles) of the graph.

The clustering coefficient of a graph is the fraction of possible triangles that are triangles, c_i = triangles_i / (k_i*(k_i-1)/2) where k_i is the degree of vertex i, [1]. A coefficient for the whole graph is the average of the c_i. Transitivity is the fraction of all possible triangles which are triangles, T = 3*triangles/triads, [1].

REFERENCE:

- [1] Aric Hagberg, Dan Schult and Pieter Swart. NetworkX documentation. [Online] Available: https://networkx.lanl.gov/reference/networkx/

EXAMPLES:

```
sage: (graphs.FruchtGraph()).cluster_transitivity()
0.25
```

cluster_triangles(*nbunch=None, with_labels=False*)

Returns the number of triangles for nbunch of vertices as an ordered list.

The clustering coefficient of a graph is the fraction of possible triangles that are triangles, c_i = triangles_i / (k_i*(k_i-1)/2) where k_i is the degree of vertex i, [1]. A coefficient for the whole graph is the average of the c_i. Transitivity is the fraction of all possible triangles which are triangles, T = 3*triangles/triads, [1].

INPUT:

- `nbunch` - The vertices to inspect. If nbunch=None, returns data for all vertices in the graph
- `with_labels` - (boolean) default False returns list as above True returns dict keyed by vertex labels.

REFERENCE:

- [1] Aric Hagberg, Dan Schult and Pieter Swart. NetworkX documentation. [Online] Available: https://networkx.lanl.gov/reference/networkx/

EXAMPLES:

```
sage: (graphs.FruchtGraph()).cluster_triangles()
[1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 0]
sage: (graphs.FruchtGraph()).cluster_triangles(with_labels=True)
{0: 1, 1: 1, 2: 0, 3: 1, 4: 1, 5: 1, 6: 1, 7: 1, 8: 0, 9: 1, 10: 1, 11: 0}
sage: (graphs.FruchtGraph()).cluster_triangles(nbunch=[0,1,2])
[1, 1, 0]
```

clustering_average()

Returns the average clustering coefficient.

The clustering coefficient of a graph is the fraction of possible triangles that are triangles, c_i = triangles_i / (k_i*(k_i-1)/2) where k_i is the degree of vertex i, [1]. A coefficient for the whole graph is the average of the c_i. Transitivity is the fraction of all possible triangles which are triangles, T = 3*triangles/triads, [1].

REFERENCE:

- [1] Aric Hagberg, Dan Schult and Pieter Swart. NetworkX documentation. [Online] Available: https://networkx.lanl.gov/reference/networkx/

EXAMPLES:

```
sage: (graphs.FruchtGraph()).clustering_average()
0.25
```

clustering_coeff(*nbunch=None, with_labels=False, weights=False*)

Returns the clustering coefficient for each vertex in nbunch as an ordered list.

The clustering coefficient of a graph is the fraction of possible triangles that are triangles, c_i = triangles_i / (k_i*(k_i-1)/2) where k_i is the degree of vertex i, [1]. A coefficient for the whole graph is the average of the c_i. Transitivity is the fraction of all possible triangles which are triangles, T = 3*triangles/triads, [1].

INPUT:

- `nbunch` - the vertices to inspect (default None returns data on all vertices in graph)
- `with_labels` - (boolean) default False returns list as above True returns dict keyed by vertex labels.
- `weights` - default is False. If both with_labels and weights are True, then returns a clustering coefficient dict and a dict of weights based on degree. Weights are the fraction of connected triples in the graph that include the keyed vertex.

REFERENCE:

- [1] Aric Hagberg, Dan Schult and Pieter Swart. NetworkX documentation. [Online] Available: https://networkx.lanl.gov/reference/networkx/

EXAMPLES:

```
sage: (graphs.FruchtGraph()).clustering_coeff()
[0.3333333333333331, 0.3333333333333331, 0.0, 0.3333333333333331, 0.3333333333333331, 0.
sage: (graphs.FruchtGraph()).clustering_coeff(with_labels=True)
{0: 0.3333333333333331, 1: 0.3333333333333331, 2: 0.0, 3: 0.3333333333333331, 4: 0.333333
sage: (graphs.FruchtGraph()).clustering_coeff(with_labels=True,weights=True)
({0: 0.3333333333333331, 1: 0.3333333333333331, 2: 0.0, 3: 0.3333333333333331, 4: 0.33333
sage: (graphs.FruchtGraph()).clustering_coeff(nbunch=[0,1,2])
[0.3333333333333331, 0.3333333333333331, 0.0]
sage: (graphs.FruchtGraph()).clustering_coeff(nbunch=[0,1,2],with_labels=True,weights=True)
({0: 0.3333333333333331, 1: 0.3333333333333331, 2: 0.0}, {0: 0.083333333333333329, 1: 0.08
```

coarsest_equitable_refinement(*partition, sparse=False*)

Returns the coarsest partition which is finer than the input partition, and equitable with respect to self.

A partition is equitable with respect to a graph if for every pair of cells C1, C2 of the partition, the number of edges from a vertex of C1 to C2 is the same, over all vertices in C1.

A partition P1 is finer than P2 (P2 is coarser than P1) if every cell of P1 is a subset of a cell of P2.

INPUT:

- `partition` - a list of lists
- `sparse` - (default False) whether to use sparse or dense representation- for small graphs, use dense for speed

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.coarsest_equitable_refinement([[0],range(1,10)])
[[0], [2, 3, 6, 7, 8, 9], [1, 4, 5]]
sage: G = graphs.CubeGraph(3)
sage: verts = G.vertices()
sage: Pi = [verts[:1], verts[1:]]
sage: Pi
[['000'], ['001', '010', '011', '100', '101', '110', '111']]
sage: G.coarsest_equitable_refinement(Pi)
[['000'], ['011', '101', '110'], ['111'], ['001', '010', '100']]
```

Note that given an equitable partition, this function returns that partition:

```
sage: P = graphs.PetersenGraph()
sage: prt = [[0], [1, 4, 5], [2, 3, 6, 7, 8, 9]]
sage: P.coarsest_equitable_refinement(prt)
[[0], [1, 4, 5], [2, 3, 6, 7, 8, 9]]

sage: ss = (graphs.WheelGraph(6)).line_graph(labels=False)
sage: prt = [[(0, 1)], [(0, 2), (0, 3), (0, 4), (1, 2), (1, 4)], [(2, 3), (3, 4)]]
sage: ss.coarsest_equitable_refinement(prt)
...
TypeError: Partition ([[(0, 1)], [(0, 2), (0, 3), (0, 4), (1, 2), (1, 4)], [(2, 3), (3, 4)]]

sage: ss = (graphs.WheelGraph(5)).line_graph(labels=False)
sage: ss.coarsest_equitable_refinement(prt)
[[(0, 1)], [(1, 2), (1, 4)], [(0, 3)], [(0, 2), (0, 4)], [(2, 3), (3, 4)]]
```

ALGORITHM: Brendan D. McKay's Master's Thesis, University of Melbourne, 1976.

complement()

Returns the complement of the (di)graph.

The complement of a graph has the same vertices, but exactly those edges that are not in the original graph. This is not well defined for graphs with multiple edges.

EXAMPLE:

```
sage: P = graphs.PetersenGraph()
sage: P.plot().show()
sage: PC = P.complement()
sage: PC.plot().show()

sage: graphs.TetrahedralGraph().complement().size()
0
sage: graphs.CycleGraph(4).complement().edges()
[(0, 2, None), (1, 3, None)]
sage: graphs.CycleGraph(4).complement()
complement(Cycle graph): Graph on 4 vertices
sage: G = Graph(multiedges=True)
sage: G.add_edges([(0,1)]*3)
sage: G.complement()
...
TypeError: Complement not well defined for (di)graphs with multiple edges.
```

connected_component_containing_vertex(*vertex*)

Returns a list of the vertices connected to vertex.

EXAMPLE:

```
sage: G = Graph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: G.connected_component_containing_vertex(0)
[0, 1, 2, 3]
sage: D = DiGraph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: D.connected_component_containing_vertex(0)
[0, 1, 2, 3]
```

connected_components()

Returns a list of lists of vertices, each list representing a connected component. The list is ordered from largest to smallest component.

EXAMPLE:

```
sage: G = Graph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: G.connected_components()
[[0, 1, 2, 3], [4, 5, 6]]
sage: D = DiGraph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: D.connected_components()
[[0, 1, 2, 3], [4, 5, 6]]
```

connected_components_number()

Returns the number of connected components.

EXAMPLE:

```
sage: G = Graph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: G.connected_components_number()
2
sage: D = DiGraph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: D.connected_components_number()
2
```

connected_components_subgraphs()

Returns a list of connected components as graph objects.

EXAMPLE:

```
sage: G = Graph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
sage: L = G.connected_components_subgraphs()
sage: graphs_list.show_graphs(L)
sage: D = DiGraph( { 0 : [1, 3], 1 : [2], 2 : [3], 4 : [5, 6], 5 : [6] } )
```

```
sage: L = D.connected_components_subgraphs()
sage: graphs_list.show_graphs(L)
```

copy(*implementation='networkx', sparse=True*)

Creates a copy of the graph.

EXAMPLES:

```
sage: g=Graph({0:[0,1,1,2]},loops=True,multiedges=True,implementation='networkx')
sage: g==g.copy()
True
sage: g=DiGraph({0:[0,1,1,2],1:[0,1]},loops=True,multiedges=True,implementation='networkx')
sage: g==g.copy()
True
```

Note that vertex associations are also kept:

```
sage: d = {0 : graphs.DodecahedralGraph(), 1 : graphs.FlowerSnark(), 2 : graphs.MoebiusKanto
sage: T = graphs.TetrahedralGraph()
sage: T.set_vertices(d)
sage: T2 = T.copy()
sage: T2.get_vertex(0)
Dodecahedron: Graph on 20 vertices
```

Notice that the copy is at least as deep as the objects:

```
sage: T2.get_vertex(0) is T.get_vertex(0)
False
```

TESTS: We make copies of the _pos and _boundary attributes.

```
sage: g = graphs.PathGraph(3)
sage: h = g.copy()
sage: h._pos is g._pos
False
sage: h._boundary is g._boundary
False
```

cores(*with_labels=False*)

Returns the core number for each vertex in an ordered list.

'K-cores in graph theory were introduced by Seidman in 1983 and by Bollobas in 1984 as a method of (destructively) simplifying graph topology to aid in analysis and visualization. They have been more recently defined as the following by Batagelj et al: given a graph G with vertices set V and edges set E, the k-core is computed by pruning all the vertices (with their respective edges) with degree less than k. That means that if a vertex u has degree d_u, and it has n neighbors with degree less than k, then the degree of u becomes d_u - n, and it will be also pruned if k d_u - n. This operation can be useful to filter or to study some properties of the graphs. For instance, when you compute the 2-core of graph G, you are cutting all the vertices which are in a tree part of graph. (A tree is a graph with no loops),' [1].

INPUT:

- `with_labels` - default False returns list as described above. True returns dict keyed by vertex labels.

REFERENCE:

- [1] K-core. Wikipedia. (2007). [Online] Available: http://en.wikipedia.org/wiki/K-core
- [2] Boris Pittel, Joel Spencer and Nicholas Wormald. Sudden Emergence of a Giant k-Core in a Random Graph. (1996). J. Combinatorial Theory. Ser B 67. pages 111-151. [Online] Available: http://cs.nyu.edu/cs/faculty/spencer/papers/k-core.pdf

EXAMPLES:

```
sage: (graphs.FruchtGraph()).cores()
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
sage: (graphs.FruchtGraph()).cores(with_labels=True)
{0: 3, 1: 3, 2: 3, 3: 3, 4: 3, 5: 3, 6: 3, 7: 3, 8: 3, 9: 3, 10: 3, 11: 3}
```

degree(*vertices=None, labels=False*)

Gives the degree (in + out for digraphs) of a vertex or of vertices.

INPUT:

- `vertices` - If vertices is a single vertex, returns the number of neighbors of vertex. If vertices is an iterable container of vertices, returns a list of degrees. If vertices is None, same as listing all vertices.
- `labels` - see OUTPUT

OUTPUT: Single vertex- an integer. Multiple vertices- a list of integers. If labels is True, then returns a dictionary mapping each vertex to its degree.

EXAMPLES:

```
sage: P = graphs.PetersenGraph()
sage: P.degree(5)
3

sage: K = graphs.CompleteGraph(9)
sage: K.degree()
[8, 8, 8, 8, 8, 8, 8, 8, 8]

sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: D.degree(vertices = [0,1,2], labels=True)
{0: 5, 1: 4, 2: 3}
sage: D.degree()
[5, 4, 3, 3, 3, 2]
```

degree_histogram()

Returns a list, whose ith entry is the frequency of degree i.

EXAMPLE:

```
sage: G = graphs.Grid2dGraph(9,12)
sage: G.degree_histogram()
[0, 0, 4, 34, 70]

sage: G = graphs.Grid2dGraph(9,12).to_directed()
sage: G.degree_histogram()
[0, 0, 0, 0, 4, 0, 34, 0, 70]
```

degree_iterator(*vertices=None, labels=False*)

Returns an iterator over the degrees of the (di)graph. In the case of a digraph, the degree is defined as the sum of the in-degree and the out-degree, i.e. the total number of edges incident to a given vertex.

INPUT: labels=False: returns an iterator over degrees. labels=True: returns an iterator over tuples (vertex, degree).

- `vertices` - if specified, restrict to this subset.

EXAMPLES:

```
sage: G = graphs.Grid2dGraph(3,4)
sage: for i in G.degree_iterator():
...     print i
3
4
2
...
```

```
2
3
sage: for i in G.degree_iterator(labels=True):
...     print i
((0, 1), 3)
((1, 2), 4)
((0, 0), 2)
...
((0, 3), 2)
((0, 2), 3)

sage: D = graphs.Grid2dGraph(2,4).to_directed()
sage: for i in D.degree_iterator():
...     print i
6
6
...
4
6
sage: for i in D.degree_iterator(labels=True):
...     print i
((0, 1), 6)
((1, 2), 6)
...
((0, 3), 4)
((1, 1), 6)
```

degree_to_cell(*vertex, cell*)

Returns the number of edges from vertex to an edge in cell. In the case of a digraph, returns a tuple (in_degree, out_degree).

EXAMPLES:

```
sage: G = graphs.CubeGraph(3)
sage: cell = G.vertices()[:3]
sage: G.degree_to_cell('011', cell)
2
sage: G.degree_to_cell('111', cell)
0

sage: D = DiGraph({ 0:[1,2,3], 1:[3,4], 3:[4,5]})
sage: cell = [0,1,2]
sage: D.degree_to_cell(5, cell)
(0, 0)
sage: D.degree_to_cell(3, cell)
(2, 0)
sage: D.degree_to_cell(0, cell)
(0, 2)
```

delete_edge(*u, v=None, label=None*)

Delete the edge from u to v, returning silently if vertices or edge does not exist.

INPUT: The following forms are all accepted:

- G.delete_edge(1, 2)
- G.delete_edge((1, 2))
- G.delete_edges([(1, 2)])
- G.delete_edge(1, 2, 'label')
- G.delete_edge((1, 2, 'label'))
- G.delete_edges([(1, 2, 'label')])

EXAMPLES:

```
sage: G = graphs.CompleteGraph(19)
sage: G.size()
171
sage: G.delete_edge( 1, 2 )
sage: G.delete_edge( (3, 4) )
sage: G.delete_edges( [ (5, 6), (7, 8) ] )
sage: G.delete_edge( 9, 10, 'label' )
sage: G.delete_edge( (11, 12, 'label') )
sage: G.delete_edges( [ (13, 14, 'label') ] )
sage: G.size()
164
sage: G.has_edge( (11, 12) )
False
```

Note that even though the edge (11, 12) has no label, it still gets deleted: NetworkX does not pay attention to labels here.

```
sage: D = graphs.CompleteGraph(19).to_directed()
sage: D.size()
342
sage: D.delete_edge( 1, 2 )
sage: D.delete_edge( (3, 4) )
sage: D.delete_edges( [ (5, 6), (7, 8) ] )
sage: D.delete_edge( 9, 10, 'label' )
sage: D.delete_edge( (11, 12, 'label') )
sage: D.delete_edges( [ (13, 14, 'label') ] )
sage: D.size()
335
sage: D.has_edge( (11, 12) )
False
```

delete_edges(*edges*)

Delete edges from an iterable container.

EXAMPLE:

```
sage: K12 = graphs.CompleteGraph(12)
sage: K4 = graphs.CompleteGraph(4)
sage: K12.size()
66
sage: K12.delete_edges(K4.edge_iterator())
sage: K12.size()
60

sage: K12 = graphs.CompleteGraph(12).to_directed()
sage: K4 = graphs.CompleteGraph(4).to_directed()
sage: K12.size()
132
sage: K12.delete_edges(K4.edge_iterator())
sage: K12.size()
120
```

delete_multiedge(*u*, *v*)

Deletes all edges from u and v.

EXAMPLE:

```
sage: G = Graph(multiedges=True, implementation='networkx')
sage: G.add_edges([(0,1), (0,1), (0,1), (1,2), (2,3)])
sage: G.edges()
```

```
[(0, 1, None), (0, 1, None), (0, 1, None), (1, 2, None), (2, 3, None)]
sage: G.delete_multiedge( 0, 1 )
sage: G.edges()
[(1, 2, None), (2, 3, None)]

sage: D = DiGraph(multiedges=True)
sage: D.add_edges([(0,1,1), (0,1,2), (0,1,3), (1,0), (1,2), (2,3)])
sage: D.edges()
[(0, 1, 1), (0, 1, 2), (0, 1, 3), (1, 0, None), (1, 2, None), (2, 3, None)]
sage: D.delete_multiedge( 0, 1 )
sage: D.edges()
[(1, 0, None), (1, 2, None), (2, 3, None)]
```

delete_vertex(*vertex, in_order=False*)

Deletes vertex, removing all incident edges. Deleting a non-existant vertex will raise an exception.

INPUT:

- `in_order` - (default False) If True, this deletes the ith vertex in the sorted list of vertices, i.e. G.vertices()[i]

EXAMPLES:

```
sage: G = Graph(graphs.WheelGraph(9), sparse=True)
sage: G.delete_vertex(0); G.show()

sage: D = DiGraph({0:[1,2,3,4,5],1:[2],2:[3],3:[4],4:[5],5:[1]}, implementation='networkx')
sage: D.delete_vertex(0); D
Digraph on 5 vertices
sage: D.vertices()
[1, 2, 3, 4, 5]
sage: D.delete_vertex(0)
...
NetworkXError: node 0 not in graph

sage: G = graphs.CompleteGraph(4).line_graph(labels=False)
sage: G.vertices()
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: G.delete_vertex(0, in_order=True)
sage: G.vertices()
[(0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: G = graphs.PathGraph(5)
sage: G.set_vertices({0: 'no delete', 1: 'delete'})
sage: G.set_boundary([1,2])
sage: G.delete_vertex(1)
sage: G.get_vertices()
{0: 'no delete', 2: None, 3: None, 4: None}
sage: G.get_boundary()
[2]
sage: G.get_pos()
{0: [0, 0], 2: [2, 0], 3: [3, 0], 4: [4, 0]}
```

delete_vertices(*vertices*)

Remove vertices from the (di)graph taken from an iterable container of vertices. Deleting a non-existant vertex will raise an exception.

EXAMPLE:

```
sage: D = DiGraph({0:[1,2,3,4,5],1:[2],2:[3],3:[4],4:[5],5:[1]}, implementation='networkx')
sage: D.delete_vertices([1,2,3,4,5]); D
Digraph on 1 vertex
```

```
sage: D.vertices()
[0]
sage: D.delete_vertices([1])
...
NetworkXError: node 1 not in graph
```

density()

Returns the density (number of edges divided by number of possible edges).

In the case of a multigraph, raises an error, since there is an infinite number of possible edges.

EXAMPLE:

```
sage: d = {0: [1,4,5], 1: [2,6], 2: [3,7], 3: [4,8], 4: [9], 5: [7, 8], 6: [8,9], 7: [9]}
sage: G = Graph(d); G.density()
1/3
sage: G = Graph({0:[1,2], 1:[0] }); G.density()
2/3
sage: G = DiGraph({0:[1,2], 1:[0] }); G.density()
1/2
```

Note that there are more possible edges on a looped graph:

```
sage: G.allow_loops(True)
sage: G.density()
1/3
```

depth_first_search(*u, ignore_direction=False*)

Returns an iterator over vertices in a depth-first ordering.

INPUT:

- u - vertex at which to start search
- ignore_direction - (default False) only applies to directed graphs. If True, searches across edges in either direction.

EXAMPLES:

```
sage: G = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} } )
sage: list(G.breadth_first_search(0))
[0, 1, 4, 2, 3]
sage: list(G.depth_first_search(0))
[0, 4, 3, 2, 1]

sage: D = DiGraph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} } )
sage: list(D.breadth_first_search(0))
[0, 1, 2, 3, 4]
sage: list(D.depth_first_search(0))
[0, 1, 2, 3, 4]

sage: D = DiGraph({1:[0], 2:[0]})
sage: list(D.depth_first_search(0))
[0]
sage: list(D.depth_first_search(0, ignore_direction=True))
[0, 2, 1]
```

diameter()

Returns the largest distance between any two vertices. Returns Infinity if the (di)graph is not connected.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.diameter()
2
```

```
sage: G = Graph( { 0 : [], 1 : [], 2 : [1] } )
sage: G.diameter()
+Infinity
```

Although max( ) is usually defined as -Infinity, since the diameter will never be negative, we define it to be zero:

```
sage: G = graphs.EmptyGraph()
sage: G.diameter()
0
```

disjoint_union(*other, verbose_relabel=True*)

Returns the disjoint union of self and other.

If the graphs have common vertices, the vertices will be renamed to form disjoint sets.

INPUT:

- `verbose_relabel` - (defaults to True) If True and the graphs have common vertices, then each vertex v in the first graph will be changed to '0,v' and each vertex u in the second graph will be changed to '1,u'. If False, the vertices of the first graph and the second graph will be relabeled with consecutive integers.

EXAMPLE:

```
sage: G = graphs.CycleGraph(3)
sage: H = graphs.CycleGraph(4)
sage: J = G.disjoint_union(H); J
Cycle graph disjoint_union Cycle graph: Graph on 7 vertices
sage: J.vertices()
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (1, 3)]
sage: J = G.disjoint_union(H, verbose_relabel=False); J
Cycle graph disjoint_union Cycle graph: Graph on 7 vertices
sage: J.vertices()
[0, 1, 2, 3, 4, 5, 6]
```

If the vertices are already disjoint and verbose_relabel is True, then the vertices are not relabeled.

```
sage: G=Graph({'a': ['b']}, implementation='networkx')
sage: G.name("Custom path")
sage: G.name()
'Custom path'
sage: H=graphs.CycleGraph(3)
sage: J=G.disjoint_union(H); J
Custom path disjoint_union Cycle graph: Graph on 5 vertices
sage: J.vertices()
[0, 1, 2, 'a', 'b']
```

disjunctive_product(*other*)

Returns the disjunctive product of self and other.

The disjunctive product of G and H is the graph L with vertex set V(L) equal to the Cartesian product of the vertices V(G) and V(H), and ((u,v), (w,x)) is an edge iff either - (u, w) is an edge of self, or - (v, x) is an edge of other.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: D = Z.disjunctive_product(Z); D
Graph on 4 vertices
sage: D.plot().show()

sage: C = graphs.CycleGraph(5)
sage: D = C.disjunctive_product(Z); D
```

```
Graph on 10 vertices
sage: D.plot().show()
```

distance(*u, v*)

Returns the (directed) distance from u to v in the (di)graph, i.e. the length of the shortest path from u to v.

EXAMPLES:

```
sage: G = graphs.CycleGraph(9)
sage: G.distance(0,1)
1
sage: G.distance(0,4)
4
sage: G.distance(0,5)
4
sage: G = Graph( {0:[], 1:[]} )
sage: G.distance(0,1)
+Infinity
```

eccentricity(*v=None, dist_dict=None, with_labels=False*)

Return the eccentricity of vertex (or vertices) v.

The eccentricity of a vertex is the maximum distance to any other vertex.

INPUT:

- `v` - either a single vertex or a list of vertices. If it is not specified, then it is taken to be all vertices.
- `dist_dict` - optional, a dict of dicts of distance.
- `with_labels` - Whether to return a list or a dict.

EXAMPLES:

```
sage: G = graphs.KrackhardtKiteGraph()
sage: G.eccentricity()
[4, 4, 4, 4, 4, 3, 3, 2, 3, 4]
sage: G.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: G.eccentricity(7)
2
sage: G.eccentricity([7,8,9])
[3, 4, 2]
sage: G.eccentricity([7,8,9], with_labels=True) == {8: 3, 9: 4, 7: 2}
True
sage: G = Graph( { 0 : [], 1 : [], 2 : [1] } )
sage: G.eccentricity()
[+Infinity, +Infinity, +Infinity]
sage: G = Graph({0:[]})
sage: G.eccentricity(with_labels=True)
{0: 0}
sage: G = Graph({0:[], 1:[]})
sage: G.eccentricity(with_labels=True)
{0: +Infinity, 1: +Infinity}
```

edge_boundary(*vertices1, vertices2=None, labels=True*)

Returns a list of edges (u,v,l) with u in vertices1 and v in vertices2. If vertices2 is None, then it is set to the complement of vertices1.

In a digraph, the external boundary of a vertex v are those vertices u with an arc (v, u).

INPUT:

- `labels` - if False, each edge is a tuple (u,v) of vertices.

EXAMPLE:

```
sage: K = graphs.CompleteBipartiteGraph(9,3)
sage: len(K.edge_boundary( [0,1,2,3,4,5,6,7,8], [9,10,11] ))
27
sage: K.size()
27
```

Note that the edge boundary preserves direction:

```
sage: K = graphs.CompleteBipartiteGraph(9,3).to_directed()
sage: len(K.edge_boundary( [0,1,2,3,4,5,6,7,8], [9,10,11] ))
27
sage: K.size()
54
```

```
sage: D = DiGraph({0:[1,2], 3:[0]})
sage: D.edge_boundary([0])
[(0, 1, None), (0, 2, None)]
sage: D.edge_boundary([0], labels=False)
[(0, 1), (0, 2)]
```

edge_iterator(*vertices=None, labels=True, ignore_direction=False*)

Returns an iterator over the edges incident with any vertex given. If the graph is directed, iterates over edges going out only. If vertices is None, then returns an iterator over all edges. If self is directed, returns outgoing edges only.

INPUT:

- `labels` - if False, each edge is a tuple (u,v) of vertices.
- `ignore_direction` - (default False) only applies to directed graphs. If True, searches across edges in either direction.

EXAMPLE:

```
sage: for i in graphs.PetersenGraph().edge_iterator([0]):
...     print i
(0, 1, None)
(0, 4, None)
(0, 5, None)
sage: D = DiGraph( { 0 : [1,2], 1: [0] } )
sage: for i in D.edge_iterator([0]):
...     print i
(0, 1, None)
(0, 2, None)
```

```
sage: G = graphs.TetrahedralGraph()
sage: list(G.edge_iterator(labels=False))
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
```

```
sage: D = DiGraph({1:[0], 2:[0]})
sage: list(D.edge_iterator(0))
[]
sage: list(D.edge_iterator(0, ignore_direction=True))
[(1, 0, None), (2, 0, None)]
```

edge_label(*u, v=None*)

Returns the label of an edge. Note that if the graph allows multiple edges, then a list of labels on the edge is returned.

EXAMPLE:

```
sage: G = Graph({0 : {1 : 'edgelabel'}}, implementation='networkx')
sage: G.edges(labels=False)
[(0, 1)]
sage: G.edge_label( 0, 1 )
'edgelabel'
sage: D = DiGraph({0 : {1 : 'edgelabel'}}, implementation='networkx')
sage: D.edges(labels=False)
[(0, 1)]
sage: D.edge_label( 0, 1 )
'edgelabel'

sage: G = Graph(multiedges=True)
sage: [G.add_edge(0,1,i) for i in range(1,6)]
[None, None, None, None, None]
sage: sorted(G.edge_label(0,1))
[1, 2, 3, 4, 5]
```

edge_labels()

Returns a list of edge labels.

EXAMPLE:

```
sage: G = Graph({0:{1:'x',2:'z',3:'a'}, 2:{5:'out'}}, implementation='networkx')
sage: G.edge_labels()
['x', 'z', 'a', 'out']
sage: G = DiGraph({0:{1:'x',2:'z',3:'a'}, 2:{5:'out'}}, implementation='networkx')
sage: G.edge_labels()
['x', 'z', 'a', 'out']
```

edges(*labels=True, sort=True*)

Return a list of edges. Each edge is a triple (u,v,l) where u and v are vertices and l is a label.

INPUT:

- `labels` - (bool; default: True) if False, each edge is a tuple (u,v) of vertices.
- `sort` - (bool; default: True) if True, ensure that the list of edges is sorted.

OUTPUT: A list of tuples. It is safe to change the returned list.

EXAMPLES:

```
sage: graphs.DodecahedralGraph().edges()
[(0, 1, None), (0, 10, None), (0, 19, None), (1, 2, None), (1, 8, None), (2, 3, None), (2, 6

sage: graphs.DodecahedralGraph().edges(labels=False)
[(0, 1), (0, 10), (0, 19), (1, 2), (1, 8), (2, 3), (2, 6), (3, 4), (3, 19), (4, 5), (4, 17),

sage: D = graphs.DodecahedralGraph().to_directed()
sage: D.edges()
[(0, 1, None), (0, 10, None), (0, 19, None), (1, 0, None), (1, 2, None), (1, 8, None), (2, 1
sage: D.edges(labels = False)
[(0, 1), (0, 10), (0, 19), (1, 0), (1, 2), (1, 8), (2, 1), (2, 3), (2, 6), (3, 2), (3, 4), (
```

edges_incident(*vertices=None, labels=True*)

Returns a list of edges incident with any vertex given. If vertices is None, returns a list of all edges in graph. For digraphs, only lists outward edges.

INPUT:

- `label` - if False, each edge is a tuple (u,v) of vertices.

EXAMPLE:

```
sage: graphs.PetersenGraph().edges_incident([0,9], labels=False)
[(0, 1), (0, 4), (0, 5), (9, 4), (9, 6), (9, 7)]
sage: D = DiGraph({0:[1]})
sage: D.edges_incident([0])
[(0, 1, None)]
sage: D.edges_incident([1])
[]
```

eigenspaces(*laplacian=False*)

Returns the eigenspaces of the adjacency matrix of the graph.

INPUT:

- `laplacian` - if True, use the Laplacian matrix instead (see self.kirchhoff_matrix())

EXAMPLE:

```
sage: C = graphs.CycleGraph(5)
sage: E = C.eigenspaces()
sage: E[0][0]
-1.618...
sage: E[1][0]  # eigenspace computation is somewhat random
Vector space of degree 5 and dimension 1 over Real Double Field
User basis matrix:
[ 0.632... -0.632... -0.447... -0.013... 0.073...]

sage: D = C.to_directed()
sage: F = D.eigenspaces()
sage: abs(E[0][0] - F[0][0]) < 0.00001
True
```

genus(*set_embedding=True, on_embedding=None, minimal=True, maximal=False, circular=False, ordered=True*)

Returns the minimal genus of the graph. The genus of a compact surface is the number of handles it has. The genus of a graph is the minimal genus of the surface it can be embedded into.

Note - This function uses Euler's formula and thus it is necessary to consider only connected graphs.

INPUT:

- `set_embedding (boolean)` - whether or not to store an embedding attribute of the computed (minimal) genus of the graph. (Default is True).
- `on_embedding (dict)` - a combinatorial embedding to compute the genus of the graph on. Note that this must be a valid embedding for the graph. The dictionary structure is given by: vertex1: [neighbor1, neighbor2, neighbor3], vertex2: [neighbor] where there is a key for each vertex in the graph and a (clockwise) ordered list of each vertex's neighbors as values. on_embedding takes precedence over a stored _embedding attribute if minimal is set to False. Note that as a shortcut, the user can enter on_embedding=True to compute the genus on the current _embedding attribute. (see eg's.)
- `minimal (boolean)` - whether or not to compute the minimal genus of the graph (i.e., testing all embeddings). If minimal is False, then either maximal must be True or on_embedding must not be None. If on_embedding is not None, it will take priority over minimal. Similarly, if maximal is True, it will take priority over minimal.
- `maximal (boolean)` - whether or not to compute the maximal genus of the graph (i.e., testin all embeddings). If maximal is False, then either minimal must be True or on_embedding must not be None. If on_embedding is not None, it will take priority over maximal. However, maximal takes priority over the default minimal.
- `circular (boolean)` - whether or not to compute the genus preserving a planar embedding of the boundary. (Default is False). If circular is True, on_embedding is not a valid option.
- `ordered (boolean)` - if circular is True, then whether or not the boundary order may be permuted. (Default is True, which means the boundary order is preserved.)

EXAMPLES:

```
sage: g = graphs.PetersenGraph()
sage: g.genus() # tests for minimal genus by default
1
sage: g.genus(on_embedding=True, maximal=True) # on_embedding overrides minimal and maximal
1
sage: g.genus(maximal=True) # setting maximal to True overrides default minimal=True
3
sage: g.genus(on_embedding=g.get_embedding()) # can also send a valid combinatorial embeddir
3
sage: (graphs.CubeGraph(3)).genus()
0
sage: K23 = graphs.CompleteBipartiteGraph(2,3)
sage: K23.genus()
0
sage: K33 = graphs.CompleteBipartiteGraph(3,3)
sage: K33.genus()
1
```

Using the circular argument, we can compute the minimal genus preserving a planar, ordered boundary:

```
sage: cube = graphs.CubeGraph(3)
sage: cube.set_boundary(['001','110'])
sage: cube.genus()
0
sage: cube.is_circular_planar()
False
sage: cube.genus(circular=True) #long time
1
sage: cube.genus(circular=True, maximal=True) #long time
3
sage: cube.genus(circular=True, on_embedding=True) #long time
3
```

get_boundary()

Returns the boundary of the (di)graph.

EXAMPLE:

```
sage: G = graphs.PetersenGraph()
sage: G.set_boundary([0,1,2,3,4])
sage: G.get_boundary()
[0, 1, 2, 3, 4]
```

get_embedding()

Returns the attribute _embedding if it exists. _embedding is a dictionary organized with vertex labels as keys and a list of each vertex's neighbors in clockwise order.

Error-checked to insure valid embedding is returned.

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.genus()
1
sage: G.get_embedding()
{0: [1, 5, 4],
 1: [0, 2, 6],
 2: [1, 3, 7],
 3: [8, 2, 4],
 4: [0, 9, 3],
 5: [0, 8, 7],
```

```
 6: [8, 1, 9],
 7: [9, 2, 5],
 8: [3, 5, 6],
 9: [4, 6, 7]}
```

get_pos()

Returns the position dictionary, a dictionary specifying the coordinates of each vertex.

EXAMPLES: By default, the position of a graph is None:

```
sage: G = Graph()
sage: G.get_pos()
sage: G.get_pos() is None
True
sage: P = G.plot(save_pos=True)
sage: G.get_pos()
{}
```

Some of the named graphs come with a pre-specified positioning:

```
sage: G = graphs.PetersenGraph()
sage: G.get_pos()
{0: [..., ...],
 ...
 9: [..., ...]}
```

get_vertex(*vertex*)

Retrieve the object associated with a given vertex.

INPUT:

- `vertex` - the given vertex

EXAMPLES:

```
sage: d = {0 : graphs.DodecahedralGraph(), 1 : graphs.FlowerSnark(), 2 : graphs.MoebiusKanto
sage: d[2]
Moebius-Kantor Graph: Graph on 16 vertices
sage: T = graphs.TetrahedralGraph()
sage: T.vertices()
[0, 1, 2, 3]
sage: T.set_vertices(d)
sage: T.get_vertex(1)
Flower Snark: Graph on 20 vertices
```

get_vertices(*verts=None*)

Return a dictionary of the objects associated to each vertex.

INPUT:

- `verts` - iterable container of vertices

EXAMPLES:

```
sage: d = {0 : graphs.DodecahedralGraph(), 1 : graphs.FlowerSnark(), 2 : graphs.MoebiusKanto
sage: T = graphs.TetrahedralGraph()
sage: T.set_vertices(d)
sage: T.get_vertices([1,2])
{1: Flower Snark: Graph on 20 vertices,
 2: Moebius-Kantor Graph: Graph on 16 vertices}
```

girth()

Computes the girth of the graph. For directed graphs, computes the girth of the undirected graph.

The girth is the length of the shortest cycle in the graph. Graphs without cycles have infinite girth.

EXAMPLES:

```
sage: graphs.TetrahedralGraph().girth()
3
sage: graphs.CubeGraph(3).girth()
4
sage: graphs.PetersenGraph().girth()
5
sage: graphs.HeawoodGraph().girth()
6
sage: graphs.trees(9).next().girth()
+Infinity
```

graphplot(**args, **kwds*)

Returns a GraphPlot object.

EXAMPLES:

Creating a graphplot object uses the same options as graph.plot():

```
sage: g = Graph({}, loops=True, multiedges=True)
sage: g.add_edges([(0,0,'a'),(0,0,'b'),(0,1,'c'),(0,1,'d'),
...       (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: g.set_boundary([0,1])
sage: GP = g.graphplot(edge_labels=True, color_by_label=True, edge_style='dashed')
sage: GP.plot()
```

We can modify the graphplot object. Notice that the changes are cumulative:

```
sage: GP.set_edges(edge_style='solid')
sage: GP.plot()
sage: GP.set_vertices(talk=True)
sage: GP.plot()
```

graphviz_string()

Returns a representation in the DOT language, ready to render in graphviz.

EXAMPLES:

```
sage: G = Graph({0:{1:None,2:None}, 1:{0:None,2:None}, 2:{0:None,1:None,3:'foo'}, 3:{2:'foo'
sage: s = G.graphviz_string()
sage: s
'graph {\n"0";"1";"2";"3";\n"0"--"1";"0"--"2";"1"--"2";"2"--"3"[label="foo"];\n}'
```

graphviz_to_file_named(*filename*)

Write a representation in the DOT language to the named file, ready to render in graphviz.

EXAMPLES:

```
sage: G = Graph({0:{1:None,2:None}, 1:{0:None,2:None}, 2:{0:None,1:None,3:'foo'}, 3:{2:'foo'
sage: G.graphviz_to_file_named(os.environ['SAGE_TESTDIR']+'/temp_graphviz')
sage: open(os.environ['SAGE_TESTDIR']+'/temp_graphviz').read()
'graph {\n"0";"1";"2";"3";\n"0"--"1";"0"--"2";"1"--"2";"2"--"3"[label="foo"];\n}'
```

has_edge(*u, v=None, label=None*)

Returns True if (u, v) is an edge, False otherwise.

INPUT: The following forms are accepted by NetworkX:

- G.has_edge(1, 2)
- G.has_edge((1, 2))
- G.has_edge(1, 2, 'label')

EXAMPLE:

```
sage: graphs.EmptyGraph().has_edge(9,2)
False
sage: DiGraph().has_edge(9,2)
False
sage: G = Graph(implementation='networkx')
sage: G.add_edge(0,1,"label")
sage: G.has_edge(0,1,"different label")
False
sage: G.has_edge(0,1,"label")
True
```

has_loops()

Returns whether there are loops in the (di)graph.

EXAMPLES:

```
sage: G = Graph(loops=True); G
Looped graph on 0 vertices
sage: G.has_loops()
False
sage: G.allows_loops()
True
sage: G.add_edge((0,0))
sage: G.has_loops()
True
sage: G.loops()
[(0, 0, None)]
sage: G.allow_loops(False); G
Graph on 1 vertex
sage: G.has_loops()
False
sage: G.edges()
[]

sage: D = DiGraph(loops=True); D
Looped digraph on 0 vertices
sage: D.has_loops()
False
sage: D.allows_loops()
True
sage: D.add_edge((0,0))
sage: D.has_loops()
True
sage: D.loops()
[(0, 0, None)]
sage: D.allow_loops(False); D
Digraph on 1 vertex
sage: D.has_loops()
False
sage: D.edges()
[]
```

has_multiple_edges(*to_undirected=False*)

Returns whether there are multiple edges in the (di)graph.

INPUT: to_undirected – (default: False) If True, runs the test on the undirected version of a DiGraph. Otherwise, treats DiGraph edges (u,v) and (v,u) as unique individual edges.

EXAMPLES:

```
sage: G = Graph(multiedges=True); G
Multi-graph on 0 vertices
sage: G.has_multiple_edges()
False
sage: G.allows_multiple_edges()
True
sage: G.add_edges([(0,1)]*3)
sage: G.has_multiple_edges()
True
sage: G.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: G.allow_multiple_edges(False); G
Graph on 2 vertices
sage: G.has_multiple_edges()
False
sage: G.edges()
[(0, 1, None)]

sage: D = DiGraph(multiedges=True); D
Multi-digraph on 0 vertices
sage: D.has_multiple_edges()
False
sage: D.allows_multiple_edges()
True
sage: D.add_edges([(0,1)]*3)
sage: D.has_multiple_edges()
True
sage: D.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: D.allow_multiple_edges(False); D
Digraph on 2 vertices
sage: D.has_multiple_edges()
False
sage: D.edges()
[(0, 1, None)]

sage: G = DiGraph({1:{2: 'h'}, 2:{1:'g'}})
sage: G.has_multiple_edges()
False
sage: G.has_multiple_edges(to_undirected=True)
True
sage: G.multiple_edges()
[]
sage: G.multiple_edges(to_undirected=True)
[(1, 2, 'h'), (2, 1, 'g')]
```

has_vertex(*vertex*)

Return True if vertex is one of the vertices of this graph.

INPUT:

- `vertex` - an integer

OUTPUT:

- `bool` - True or False

EXAMPLES:

```
sage: g = Graph({0:[1,2,3], 2:[4]}); g
Graph on 5 vertices
sage: 2 in g
```

```
True
sage: 10 in g
False
sage: graphs.PetersenGraph().has_vertex(99)
False
```

incidence_matrix(*sparse=True*)

Returns an incidence matrix of the (di)graph. Each row is a vertex, and each column is an edge. Note that in the case of graphs, there is a choice of orientation for each edge.

EXAMPLE:

```
sage: G = graphs.CubeGraph(3)
sage: G.incidence_matrix()
[-1 -1 -1  0  0  0  0  0  0  0  0  0]
[ 0  0  1 -1 -1  0  0  0  0  0  0  0]
[ 0  1  0  0  0 -1 -1  0  0  0  0  0]
[ 0  0  0  0  1  0  1 -1  0  0  0  0]
[ 1  0  0  0  0  0  0  0 -1 -1  0  0]
[ 0  0  0  1  0  0  0  0  0  1 -1  0]
[ 0  0  0  0  0  1  0  0  1  0  0 -1]
[ 0  0  0  0  0  0  0  1  0  0  1  1]

sage: D = DiGraph( { 0: [1,2,3], 1: [0,2], 2: [3], 3: [4], 4: [0,5], 5: [1] } )
sage: D.incidence_matrix()
[-1 -1 -1  0  0  0  0  0  1  1]
[ 0  0  1 -1  0  0  0  1 -1  0]
[ 0  1  0  1 -1  0  0  0  0  0]
[ 1  0  0  0  1 -1  0  0  0  0]
[ 0  0  0  0  0  1 -1  0  0 -1]
[ 0  0  0  0  0  0  1 -1  0  0]
```

interior_paths(*start, end*)

Returns an exhaustive list of paths (also lists) through only interior vertices from vertex start to vertex end in the (di)graph.

Note - start and end do not necessarily have to be boundary vertices.

INPUT:

- `start` - the vertex of the graph to search for paths from
- `end` - the vertex of the graph to search for paths to

EXAMPLES:

```
sage: eg1 = Graph({0:[1,2], 1:[4], 2:[3,4], 4:[5], 5:[6]})
sage: sorted(eg1.all_paths(0,6))
[[0, 1, 4, 5, 6], [0, 2, 4, 5, 6]]
sage: eg2 = eg1.copy()
sage: eg2.set_boundary([0,1,3])
sage: sorted(eg2.interior_paths(0,6))
[[0, 2, 4, 5, 6]]
sage: sorted(eg2.all_paths(0,6))
[[0, 1, 4, 5, 6], [0, 2, 4, 5, 6]]
sage: eg3 = graphs.PetersenGraph()
sage: eg3.set_boundary([0,1,2,3,4])
sage: sorted(eg3.all_paths(1,4))
[[1, 0, 4],
 [1, 0, 5, 7, 2, 3, 4],
 [1, 0, 5, 7, 2, 3, 8, 6, 9, 4],
 [1, 0, 5, 7, 9, 4],
 [1, 0, 5, 7, 9, 6, 8, 3, 4],
```

```
 [1, 0, 5, 8, 3, 2, 7, 9, 4],
 [1, 0, 5, 8, 3, 4],
 [1, 0, 5, 8, 6, 9, 4],
 [1, 0, 5, 8, 6, 9, 7, 2, 3, 4],
 [1, 2, 3, 4],
 [1, 2, 3, 8, 5, 0, 4],
 [1, 2, 3, 8, 5, 7, 9, 4],
 [1, 2, 3, 8, 6, 9, 4],
 [1, 2, 3, 8, 6, 9, 7, 5, 0, 4],
 [1, 2, 7, 5, 0, 4],
 [1, 2, 7, 5, 8, 3, 4],
 [1, 2, 7, 5, 8, 6, 9, 4],
 [1, 2, 7, 9, 4],
 [1, 2, 7, 9, 6, 8, 3, 4],
 [1, 2, 7, 9, 6, 8, 5, 0, 4],
 [1, 6, 8, 3, 2, 7, 5, 0, 4],
 [1, 6, 8, 3, 2, 7, 9, 4],
 [1, 6, 8, 3, 4],
 [1, 6, 8, 5, 0, 4],
 [1, 6, 8, 5, 7, 2, 3, 4],
 [1, 6, 8, 5, 7, 9, 4],
 [1, 6, 9, 4],
 [1, 6, 9, 7, 2, 3, 4],
 [1, 6, 9, 7, 2, 3, 8, 5, 0, 4],
 [1, 6, 9, 7, 5, 0, 4],
 [1, 6, 9, 7, 5, 8, 3, 4]]
sage: sorted(eg3.interior_paths(1,4))
[[1, 6, 8, 5, 7, 9, 4], [1, 6, 9, 4]]
sage: dg = DiGraph({0:[1,3,4], 1:[3], 2:[0,3,4],4:[3]}, boundary=[4])
sage: sorted(dg.all_paths(0,3))
[[0, 1, 3], [0, 3], [0, 4, 3]]
sage: sorted(dg.interior_paths(0,3))
[[0, 1, 3], [0, 3]]
sage: ug = dg.to_undirected()
sage: sorted(ug.all_paths(0,3))
[[0, 1, 3], [0, 2, 3], [0, 2, 4, 3], [0, 3], [0, 4, 2, 3], [0, 4, 3]]
sage: sorted(ug.interior_paths(0,3))
[[0, 1, 3], [0, 2, 3], [0, 3]]
```

is_circular_planar(*ordered=True, on_embedding=None, kuratowski=False, set_embedding=False, set_pos=False*)

A graph (with nonempty boundary) is circular planar if it has a planar embedding in which all boundary vertices can be drawn in order on a disc boundary, with all the interior vertices drawn inside the disc.

Returns True if the graph is circular planar, and False if it is not. If kuratowski is set to True, then this function will return a tuple, with boolean first entry and second entry the Kuratowski subgraph or minor isolated by the Boyer-Myrvold algorithm. Note that this graph might contain a vertex or edges that were not in the initial graph. These would be elements referred to below as parts of the wheel and the star, which were added to the graph to require that the boundary can be drawn on the boundary of a disc, with all other vertices drawn inside (and no edge crossings). For more information, refer to reference [2].

This is a linear time algorithm to test for circular planarity. It relies on the edge-addition planarity algorithm due to Boyer-Myrvold. We accomplish linear time for circular planarity by modifying the graph before running the general planarity algorithm.

REFERENCE:

- [1] John M. Boyer and Wendy J. Myrvold, On the Cutting Edge: Simplified O(n) Planarity by Edge Addition. Journal of Graph Algorithms and Applications, Vol. 8, No. 3, pp. 241-273, 2004.
- [2] Kirkman, Emily A. O(n) Circular Planarity Testing. [Online] Available: soon!

INPUT:

- `ordered` - whether or not to consider the order of the boundary (set ordered=False to see if there is any possible boundary order that will satisfy circular planarity)
- `kuratowski` - if set to True, returns a tuple with boolean first entry and the Kuratowski subgraph or minor as the second entry. See notes above.
- `on_embedding` - the embedding dictionary to test planarity on. (i.e.: will return True or False only for the given embedding.)
- `set_embedding` - whether or not to set the instance field variable that contains a combinatorial embedding (clockwise ordering of neighbors at each vertex). This value will only be set if a circular planar embedding is found. It is stored as a Python dict: v1: [n1,n2,n3] where v1 is a vertex and n1,n2,n3 are its neighbors.
- `set_pos` - whether or not to set the position dictionary (for plotting) to reflect the combinatorial embedding. Note that this value will default to False if set_emb is set to False. Also, the position dictionary will only be updated if a circular planar embedding is found.

EXAMPLES:

```
sage: g439 = Graph({1:[5,7], 2:[5,6], 3:[6,7], 4:[5,6,7]}, implementation='networkx')
sage: g439.set_boundary([1,2,3,4])
sage: g439.show(figsize=[2,2], vertex_labels=True, vertex_size=175)
sage: g439.is_circular_planar()
False
sage: g439.is_circular_planar(kuratowski=True)
(False, Graph on 7 vertices)
sage: g439.set_boundary([1,2,3])
sage: g439.is_circular_planar(set_embedding=True, set_pos=False)
True
sage: g439.is_circular_planar(kuratowski=True)
(True, None)
sage: g439.get_embedding()
{1: [7, 5],
 2: [5, 6],
 3: [6, 7],
 4: [7, 6, 5],
 5: [4, 2, 1],
 6: [4, 3, 2],
 7: [3, 4, 1]}
```

Order matters:

```
sage: K23 = graphs.CompleteBipartiteGraph(2,3)
sage: K23.set_boundary([0,1,2,3])
sage: K23.is_circular_planar()
False
sage: K23.is_circular_planar(ordered=False)
True
sage: K23.set_boundary([0,2,1,3]) # Diff Order!
sage: K23.is_circular_planar(set_embedding=True)
True
```

For graphs without a boundary, circular planar is the same as planar:

```
sage: g = graphs.KrackhardtKiteGraph()
sage: g.is_circular_planar()
True
```

is_clique(*vertices=None, directed_clique=False*)

Returns True if the set `vertices` is a clique, False if not. A clique is a set of vertices such that there is an edge between any two vertices.

INPUT:

- `vertices` - Vertices can be a single vertex or an iterable container of vertices, e.g. a list, set, graph, file or numeric array. If not passed, defaults to the entire graph.
- `directed_clique` - (default False) If set to False, only consider the underlying undirected graph. If set to True and the graph is directed, only return True if all possible edges in _both_ directions exist.

EXAMPLE:

```
sage: g = graphs.CompleteGraph(4)
sage: g.is_clique([1,2,3])
True
sage: g.is_clique()
True
sage: h = graphs.CycleGraph(4)
sage: h.is_clique([1,2])
True
sage: h.is_clique([1,2,3])
False
sage: h.is_clique()
False
sage: i = graphs.CompleteGraph(4).to_directed()
sage: i.delete_edge([0,1])
sage: i.is_clique()
True
sage: i.is_clique(directed_clique=True)
False
```

is_connected()

Indicates whether the (di)graph is connected. Note that in a graph, path connected is equivalent to connected.

EXAMPLE:

```
sage: G = Graph( { 0 : [1, 2], 1 : [2], 3 : [4, 5], 4 : [5] } )
sage: G.is_connected()
False
sage: G.add_edge(0,3)
sage: G.is_connected()
True
sage: D = DiGraph( { 0 : [1, 2], 1 : [2], 3 : [4, 5], 4 : [5] } )
sage: D.is_connected()
False
sage: D.add_edge(0,3)
sage: D.is_connected()
True
sage: D = DiGraph({1:[0], 2:[0]})
sage: D.is_connected()
True
```

is_drawn_free_of_edge_crossings()

Returns True is the position dictionary for this graph is set and that position dictionary gives a planar embedding.

This simply checks all pairs of edges that don't share a vertex to make sure that they don't intersect.

Note: This function require that _pos attribute is set. (Returns False otherwise.)

EXAMPLES:

```
sage: D = graphs.DodecahedralGraph()
sage: D.set_planar_positions()
sage: D.is_drawn_free_of_edge_crossings()
True
```

is_equitable(*partition, quotient_matrix=False*)

Checks whether the given partition is equitable with respect to self.

A partition is equitable with respect to a graph if for every pair of cells C1, C2 of the partition, the number of edges from a vertex of C1 to C2 is the same, over all vertices in C1.

INPUT:

- `partition` - a list of lists
- `quotient_matrix` - (default False) if True, and the partition is equitable, returns a matrix over the integers whose rows and columns represent cells of the partition, and whose i,j entry is the number of vertices in cell j adjacent to each vertex in cell i (since the partition is equitable, this is well defined)

EXAMPLE:

```
sage: G = graphs.PetersenGraph()
sage: G.is_equitable([[0,4],[1,3,5,9],[2,6,8],[7]])
False
sage: G.is_equitable([[0,4],[1,3,5,9],[2,6,8,7]])
True
sage: G.is_equitable([[0,4],[1,3,5,9],[2,6,8,7]], quotient_matrix=True)
[1 2 0]
[1 0 2]
[0 2 1]

sage: ss = (graphs.WheelGraph(6)).line_graph(labels=False)
sage: prt = [[(0, 1)], [(0, 2), (0, 3), (0, 4), (1, 2), (1, 4)], [(2, 3), (3, 4)]]

sage: ss.is_equitable(prt)
...
TypeError: Partition [[(0, 1)], [(0, 2), (0, 3), (0, 4), (1, 2), (1, 4)], [(2, 3), (3, 4)]]

sage: ss = (graphs.WheelGraph(5)).line_graph(labels=False)
sage: ss.is_equitable(prt)
False
```

is_eulerian()

Return true if the graph has an tour that visits each edge exactly once.

EXAMPLES:

```
sage: graphs.CompleteGraph(4).is_eulerian()
False
sage: graphs.CycleGraph(4).is_eulerian()
True
sage: g = DiGraph({0:[1,2], 1:[2]}); g.is_eulerian()
False
sage: g = DiGraph({0:[2], 1:[3], 2:[0,1], 3:[2]}); g.is_eulerian()
True
```

is_forest()

Return True if the graph is a forest, i.e. a disjoint union of trees.

EXAMPLE:

```
sage: seven_acre_wood = sum(graphs.trees(7), Graph())
sage: seven_acre_wood.is_forest()
True
```

is_independent_set(*vertices=None*)

Returns True if the set `vertices` is an independent set, False if not. An independent set is a set of vertices such that there is no edge between any two vertices.

INPUT:

- `vertices` - Vertices can be a single vertex or an iterable container of vertices, e.g. a list, set, graph, file or numeric array. If not passed, defaults to the entire graph.

EXAMPLE:

```
sage: graphs.CycleGraph(4).is_independent_set([1,3])
True
sage: graphs.CycleGraph(4).is_independent_set([1,2,3])
False
```

is_isomorphic(*other, certify=False, verbosity=0, edge_labels=False*)

Tests for isomorphism between self and other.

INPUT:

- `certify` - if True, then output is (a,b), where a is a boolean and b is either a map or None.
- `edge_labels` - default False, otherwise allows only permutations respecting edge labels.

EXAMPLES: Graphs:

```
sage: from sage.groups.perm_gps.permgroup_named import SymmetricGroup
sage: D = graphs.DodecahedralGraph()
sage: E = D.copy()
sage: gamma = SymmetricGroup(20).random_element()
sage: E.relabel(gamma)
sage: D.is_isomorphic(E)
True

sage: D = graphs.DodecahedralGraph()
sage: S = SymmetricGroup(20)
sage: gamma = S.random_element()
sage: E = D.copy()
sage: E.relabel(gamma)
sage: a,b = D.is_isomorphic(E, certify=True); a
True
sage: from sage.plot.plot import GraphicsArray
sage: from sage.graphs.graph_fast import spring_layout_fast
sage: position_D = spring_layout_fast(D)
sage: position_E = {}
sage: for vert in position_D:
...     position_E[b[vert]] = position_D[vert]
sage: GraphicsArray([D.plot(pos=position_D), E.plot(pos=position_E)]).show()

sage: g=graphs.HeawoodGraph()
sage: g.is_isomorphic(g)
True
```

Multigraphs:

```
sage: G = Graph(multiedges=True)
sage: G.add_edge((0,1,1))
sage: G.add_edge((0,1,2))
sage: G.add_edge((0,1,3))
sage: G.add_edge((0,1,4))
sage: H = Graph(multiedges=True, implementation='networkx')
sage: H.add_edge((3,4))
sage: H.add_edge((3,4))
sage: H.add_edge((3,4))
sage: H.add_edge((3,4))
sage: G.is_isomorphic(H)
True
```

Digraphs:

```
sage: A = DiGraph( { 0 : [1,2] } )
sage: B = DiGraph( { 1 : [0,2] } )
sage: A.is_isomorphic(B, certify=True)
(True, {0: 1, 1: 0, 2: 2})
```

Edge labeled graphs:

```
sage: G = Graph(implementation='networkx')
sage: G.add_edges( [(0,1,'a'),(1,2,'b'),(2,3,'c'),(3,4,'b'),(4,0,'a')] )
sage: H = G.relabel([1,2,3,4,0], inplace=False)
sage: G.is_isomorphic(H, edge_labels=True)
True
```

is_planar(*on_embedding=None, kuratowski=False, set_embedding=False, set_pos=False*)

Returns True if the graph is planar, and False otherwise. This wraps the reference implementation provided by John Boyer of the linear time planarity algorithm by edge addition due to Boyer Myrvold. (See reference code in graphs.planarity).

Note - the argument on_embedding takes precedence over set_embedding. This means that only the on_embedding combinatorial embedding will be tested for planarity and no _embedding attribute will be set as a result of this function call, unless on_embedding is None.

REFERENCE:

- [1] John M. Boyer and Wendy J. Myrvold, On the Cutting Edge: Simplified O(n) Planarity by Edge Addition. Journal of Graph Algorithms and Applications, Vol. 8, No. 3, pp. 241-273, 2004.

INPUT:

- `kuratowski` - returns a tuple with boolean as first entry. If the graph is nonplanar, will return the Kuratowski subgraph or minor as the second tuple entry. If the graph is planar, returns None as the second entry.
- `on_embedding` - the embedding dictionary to test planarity on. (i.e.: will return True or False only for the given embedding.)
- `set_embedding` - whether or not to set the instance field variable that contains a combinatorial embedding (clockwise ordering of neighbors at each vertex). This value will only be set if a planar embedding is found. It is stored as a Python dict: v1: [n1,n2,n3] where v1 is a vertex and n1,n2,n3 are its neighbors.
- `set_pos` - whether or not to set the position dictionary (for plotting) to reflect the combinatorial embedding. Note that this value will default to False if set_emb is set to False. Also, the position dictionary will only be updated if a planar embedding is found.

EXAMPLES:

```
sage: g = graphs.CubeGraph(4)
sage: g.is_planar()
False

sage: g = graphs.CircularLadderGraph(4)
sage: g.is_planar(set_embedding=True)
True
sage: g.get_embedding()
{0: [1, 4, 3],
 1: [2, 5, 0],
 2: [3, 6, 1],
 3: [0, 7, 2],
 4: [0, 5, 7],
 5: [1, 6, 4],
 6: [2, 7, 5],
 7: [4, 6, 3]}
```

```
sage: g = graphs.PetersenGraph()
sage: (g.is_planar(kuratowski=True))[1].adjacency_matrix()
[0 1 0 0 0 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 0 0 0]
[0 0 1 0 1 0 0 0 1 0]
[0 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 0 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]

sage: k43 = graphs.CompleteBipartiteGraph(4,3)
sage: result = k43.is_planar(kuratowski=True); result
(False, Graph on 6 vertices)
sage: result[1].is_isomorphic(graphs.CompleteBipartiteGraph(3,3))
True
```

is_subgraph(*other*)

Tests whether self is a subgraph of other.

EXAMPLE:

```
sage: P = graphs.PetersenGraph()
sage: G = P.subgraph(range(6))
sage: G.is_subgraph(P)
True
```

is_transitively_reduced()

Returns True if the digraph is transitively reduced and False otherwise.

A digraph is transitively reduced if it is equal to its transitive reduction.

EXAMPLES:

```
sage: d = DiGraph({0:[1],1:[2],2:[3]})
sage: d.is_transitively_reduced()
True

sage: d = DiGraph({0:[1,2],1:[2]})
sage: d.is_transitively_reduced()
False

sage: d = DiGraph({0:[1,2],1:[2],2:[]})
sage: d.is_transitively_reduced()
False
```

is_tree()

Return True if the graph is a tree.

EXAMPLES:

```
sage: for g in graphs.trees(6):
...       g.is_tree()
True
True
True
True
True
True
```

is_vertex_transitive(*partition=None, verbosity=0, edge_labels=False, order=False, return_group=True, orbits=False*)

Returns whether the automorphism group of self is transitive within the partition provided, by default the unit partition of the vertices of self (thus by default tests for vertex transitivity in the usual sense).

EXAMPLE:

```
sage: G = Graph({0:[1],1:[2]})
sage: G.is_vertex_transitive()
False
sage: P = graphs.PetersenGraph()
sage: P.is_vertex_transitive()
True
sage: D = graphs.DodecahedralGraph()
sage: D.is_vertex_transitive()
True
sage: R = graphs.RandomGNP(2000, .01)
sage: R.is_vertex_transitive()
False
```

kirchhoff_matrix(*weighted=None, **kwds*)

Returns the Kirchhoff matrix (a.k.a. the Laplacian) of the graph.

The Kirchhoff matrix is defined to be D - M, where D is the diagonal degree matrix (each diagonal entry is the degree of the corresponding vertex), and M is the adjacency matrix.

If weighted == True, the weighted adjacency matrix is used for M, and the diagonal entries are the row-sums of M.

Note that any additional keywords will be pased on to either the adjacency_matrix or weighted_adjacency_matrix method.

AUTHORS:

- Tom Boothby

EXAMPLES:

```
sage: G = Graph(sparse=True)
sage: G.add_edges([(0,1,1),(1,2,2),(0,2,3),(0,3,4)])
sage: M = G.kirchhoff_matrix(weighted=True); M
[ 8 -1 -3 -4]
[-1  3 -2  0]
[-3 -2  5  0]
[-4  0  0  4]
sage: M = G.kirchhoff_matrix(); M
[ 3 -1 -1 -1]
[-1  2 -1  0]
[-1 -1  2  0]
[-1  0  0  1]
sage: G.set_boundary([2,3])
sage: M = G.kirchhoff_matrix(weighted=True, boundary_first=True); M
[ 5  0 -3 -2]
[ 0  4 -4  0]
[-3 -4  8 -1]
[-2  0 -1  3]
sage: M = G.kirchhoff_matrix(boundary_first=True); M
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
[-1  0 -1  2]
sage: M = G.laplacian_matrix(boundary_first=True); M
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
[-1  0 -1  2]
```

```
sage: M = G.laplacian_matrix(boundary_first=True, sparse=False); M
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
[-1  0 -1  2]
```

A weighted directed graph with loops:

```
sage: G = DiGraph({1:{1:2,2:3}, 2:{1:4}}, weighted=True)
sage: G.laplacian_matrix()
[ 3 -3]
[-4  4]
```

laplacian_matrix(*weighted=None, **kwds*)

Returns the Kirchhoff matrix (a.k.a. the Laplacian) of the graph.

The Kirchhoff matrix is defined to be D - M, where D is the diagonal degree matrix (each diagonal entry is the degree of the corresponding vertex), and M is the adjacency matrix.

If weighted == True, the weighted adjacency matrix is used for M, and the diagonal entries are the row-sums of M.

Note that any additional keywords will be pased on to either the adjacency_matrix or weighted_adjacency_matrix method.

AUTHORS:

- Tom Boothby

EXAMPLES:

```
sage: G = Graph(sparse=True)
sage: G.add_edges([(0,1,1),(1,2,2),(0,2,3),(0,3,4)])
sage: M = G.kirchhoff_matrix(weighted=True); M
[ 8 -1 -3 -4]
[-1  3 -2  0]
[-3 -2  5  0]
[-4  0  0  4]
sage: M = G.kirchhoff_matrix(); M
[ 3 -1 -1 -1]
[-1  2 -1  0]
[-1 -1  2  0]
[-1  0  0  1]
sage: G.set_boundary([2,3])
sage: M = G.kirchhoff_matrix(weighted=True, boundary_first=True); M
[ 5  0 -3 -2]
[ 0  4 -4  0]
[-3 -4  8 -1]
[-2  0 -1  3]
sage: M = G.kirchhoff_matrix(boundary_first=True); M
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
[-1  0 -1  2]
sage: M = G.laplacian_matrix(boundary_first=True); M
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
[-1  0 -1  2]
sage: M = G.laplacian_matrix(boundary_first=True, sparse=False); M
[ 2  0 -1 -1]
[ 0  1 -1  0]
[-1 -1  3 -1]
```

```
[-1  0 -1  2]
```

A weighted directed graph with loops:

```
sage: G = DiGraph({1:{1:2,2:3}, 2:{1:4}}, weighted=True)
sage: G.laplacian_matrix()
[ 3 -3]
[-4  4]
```

lexicographic_product (*other*)

Returns the lexicographic product of self and other.

The lexicographic product of G and H is the graph L with vertex set V(L) equal to the Cartesian product of the vertices V(G) and V(H), and ((u,v), (w,x)) is an edge iff - (u, w) is an edge of self, or - u = w and (v, x) is an edge of other.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: C = graphs.CycleGraph(5)
sage: L = C.lexicographic_product(Z); L
Graph on 10 vertices
sage: L.plot().show()

sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: L = D.lexicographic_product(P); L
Graph on 200 vertices
sage: L.plot().show()
```

line_graph (*labels=True*)

Returns the line graph of the (di)graph.

The line graph of an undirected graph G is an undirected graph H such that the vertices of H are the edges of G and two vertices e and f of H are adjacent if e and f share a common vertex in G. In other words, an edge in H represents a path of length 2 in G.

The line graph of a directed graph G is a directed graph H such that the vertices of H are the edges of G and two vertices e and f of H are adjacent if e and f share a common vertex in G and the terminal vertex of e is the initial vertex of f. In other words, an edge in H represents a (directed) path of length 2 in G.

EXAMPLE:

```
sage: g=graphs.CompleteGraph(4)
sage: h=g.line_graph()
sage: h.vertices()
[(0, 1, None),
(0, 2, None),
(0, 3, None),
(1, 2, None),
(1, 3, None),
(2, 3, None)]
sage: h.am()
[0 1 1 1 1 0]
[1 0 1 1 0 1]
[1 1 0 0 1 1]
[1 1 0 0 1 1]
[1 0 1 1 0 1]
[0 1 1 1 1 0]
sage: h2=g.line_graph(labels=False)
sage: h2.vertices()
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: h2.am()==h.am()
```

```
True
sage: g = DiGraph([[1..4],lambda i,j: i<j], implementation='networkx')
sage: h = g.line_graph()
sage: h.vertices()
[(1, 2, None),
(1, 3, None),
(1, 4, None),
(2, 3, None),
(2, 4, None),
(3, 4, None)]
sage: h.edges()
[((1, 2, None), (2, 3, None), None),
 ((1, 2, None), (2, 4, None), None),
 ((1, 3, None), (3, 4, None), None),
 ((2, 3, None), (3, 4, None), None)]
```

loop_edges()

Returns a list of all loops in the graph.

EXAMPLE:

```
sage: G = Graph(4, loops=True)
sage: G.add_edges( [ (0,0), (1,1), (2,2), (3,3), (2,3) ] )
sage: G.loop_edges()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None)]

sage: D = DiGraph(4, loops=True)
sage: D.add_edges( [ (0,0), (1,1), (2,2), (3,3), (2,3) ] )
sage: D.loop_edges()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None)]

sage: G = Graph(4, loops=True, multiedges=True)
sage: G.add_edges([(i,i) for i in range(4)])
sage: G.loop_edges()
[(0, 0, None), (1, 1, None), (2, 2, None), (3, 3, None)]
```

loop_vertices()

Returns a list of vertices with loops.

EXAMPLE:

```
sage: G = Graph({0 : [0], 1: [1,2,3], 2: [3]}, loops=True)
sage: G.loop_vertices()
[0, 1]
```

loops(*new=None, labels=True*)

Returns any loops in the (di)graph.

INPUT: new – deprecated labels – whether returned edges have labels ((u,v,l)) or not ((u,v)).

EXAMPLES: sage: G = Graph(loops=True); G Looped graph on 0 vertices sage: G.has_loops() False sage: G.allows_loops() True sage: G.add_edge((0,0)) sage: G.has_loops() True sage: G.loops() [(0, 0, None)] sage: G.allow_loops(False); G Graph on 1 vertex sage: G.has_loops() False sage: G.edges() []

sage: D = DiGraph(loops=True); D Looped digraph on 0 vertices sage: D.has_loops() False sage: D.allows_loops() True sage: D.add_edge((0,0)) sage: D.has_loops() True sage: D.loops() [(0, 0, None)] sage: D.allow_loops(False); D Digraph on 1 vertex sage: D.has_loops() False sage: D.edges() []

multiple_edges(*new=None, to_undirected=False, labels=True*)

Returns any multiple edges in the (di)graph.

EXAMPLES:

```
sage: G = Graph(multiedges=True); G
Multi-graph on 0 vertices
sage: G.has_multiple_edges()
False
sage: G.allows_multiple_edges()
True
sage: G.add_edges([(0,1)]*3)
sage: G.has_multiple_edges()
True
sage: G.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: G.allow_multiple_edges(False); G
Graph on 2 vertices
sage: G.has_multiple_edges()
False
sage: G.edges()
[(0, 1, None)]

sage: D = DiGraph(multiedges=True); D
Multi-digraph on 0 vertices
sage: D.has_multiple_edges()
False
sage: D.allows_multiple_edges()
True
sage: D.add_edges([(0,1)]*3)
sage: D.has_multiple_edges()
True
sage: D.multiple_edges()
[(0, 1, None), (0, 1, None), (0, 1, None)]
sage: D.allow_multiple_edges(False); D
Digraph on 2 vertices
sage: D.has_multiple_edges()
False
sage: D.edges()
[(0, 1, None)]

sage: G = DiGraph({1:{2: 'h'}, 2:{1:'g'}})
sage: G.has_multiple_edges()
False
sage: G.has_multiple_edges(to_undirected=True)
True
sage: G.multiple_edges()
[]
sage: G.multiple_edges(to_undirected=True)
[(1, 2, 'h'), (2, 1, 'g')]
```

name (*new=None*)

INPUT:

- `new` - if not None, then this becomes the new name of the (di)graph. (if new == '', removes any name)

EXAMPLE:

```
sage: d = {0: [1,4,5], 1: [2,6], 2: [3,7], 3: [4,8], 4: [9], 5: [7, 8], 6: [8,9], 7: [9]}
sage: G = Graph(d); G
Graph on 10 vertices
sage: G.name("Petersen Graph"); G
Petersen Graph: Graph on 10 vertices
sage: G.name(new=""); G
Graph on 10 vertices
```

```
sage: G.name()
''
```

neighbor_iterator(*vertex*)

Return an iterator over neighbors of vertex.

EXAMPLE:

```
sage: G = graphs.CubeGraph(3)
sage: for i in G.neighbor_iterator('010'):
...    print i
011
000
110
sage: D = G.to_directed()
sage: for i in D.neighbor_iterator('010'):
...    print i
011
000
110

sage: D = DiGraph({0:[1,2], 3:[0]})
sage: list(D.neighbor_iterator(0))
[1, 2, 3]
```

neighbors(*vertex*)

Return a list of neighbors (in and out if directed) of vertex.

G[vertex] also works.

EXAMPLE:

```
sage: P = graphs.PetersenGraph()
sage: sorted(P.neighbors(3))
[2, 4, 8]
sage: sorted(P[4])
[0, 3, 9]
```

networkx_graph(*copy=True*)

Creates a new NetworkX graph from the Sage graph.

INPUT:

- `copy` - if False, and the underlying implementation is a NetworkX graph, then the actual object itself is returned.

EXAMPLES:

```
sage: G = graphs.TetrahedralGraph()
sage: N = G.networkx_graph()
sage: type(N)
<class 'networkx.xgraph.XGraph'>

sage: G = graphs.TetrahedralGraph()
sage: G = Graph(G, implementation='networkx')
sage: N = G.networkx_graph()
sage: G._backend._nxg is N
False

sage: G = Graph(graphs.TetrahedralGraph(), implementation='networkx')
sage: N = G.networkx_graph(copy=False)
sage: G._backend._nxg is N
True
```

num_edges()
Returns the number of edges.
EXAMPLE:

```
sage: G = graphs.PetersenGraph()
sage: G.size()
15
```

num_verts()
Returns the number of vertices. Note that len(G) returns the number of vertices in G also.
EXAMPLE:

```
sage: G = graphs.PetersenGraph()
sage: G.order()
10

sage: G = graphs.TetrahedralGraph()
sage: len(G)
4
```

number_of_loops()
Returns the number of edges that are loops.
EXAMPLE:

```
sage: G = Graph(4, loops=True)
sage: G.add_edges( [ (0,0), (1,1), (2,2), (3,3), (2,3) ] )
sage: G.edges(labels=False)
[(0, 0), (1, 1), (2, 2), (2, 3), (3, 3)]
sage: G.number_of_loops()
4

sage: D = DiGraph(4, loops=True)
sage: D.add_edges( [ (0,0), (1,1), (2,2), (3,3), (2,3) ] )
sage: D.edges(labels=False)
[(0, 0), (1, 1), (2, 2), (2, 3), (3, 3)]
sage: D.number_of_loops()
4
```

order()
Returns the number of vertices. Note that len(G) returns the number of vertices in G also.
EXAMPLE:

```
sage: G = graphs.PetersenGraph()
sage: G.order()
10

sage: G = graphs.TetrahedralGraph()
sage: len(G)
4
```

periphery()
Returns the set of vertices in the periphery, i.e. whose eccentricity is equal to the diameter of the (di)graph.
In other words, the periphery is the set of vertices achieving the maximum eccentricity.
EXAMPLES:

```
sage: G = graphs.DiamondGraph()
sage: G.periphery()
[0, 3]
sage: P = graphs.PetersenGraph()
```

```
sage: P.subgraph(P.periphery()) == P
True
sage: S = graphs.StarGraph(19)
sage: S.periphery()
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]
sage: G = Graph(sparse=True)
sage: G.periphery()
[]
sage: G.add_vertex()
sage: G.periphery()
[0]
```

plot(**args, **kwds*)

Returns a graphics object representing the (di)graph.

INPUT:

- `pos` - an optional positioning dictionary
- `layout` - what kind of layout to use, takes precedence over pos
  - 'circular' – plots the graph with vertices evenly distributed on a circle
  - 'spring' - uses the traditional spring layout, using the graph's current positions as initial positions
  - 'tree' - the (di)graph must be a tree. One can specify the root of the tree using the keyword tree_root, otherwise a root will be selected at random. Then the tree will be plotted in levels, depending on minimum distance for the root.
- `vertex_labels` - whether to print vertex labels
- `edge_labels` - whether to print edge labels. By default, False, but if True, the result of str(l) is printed on the edge for each label l. Labels equal to None are not printed (to set edge labels, see set_edge_label).
- `vertex_size` - size of vertices displayed
- `vertex_shape` - the shape to draw the vertices (Not available for multiedge digraphs.
- `graph_border` - whether to include a box around the graph
- `vertex_colors` - optional dictionary to specify vertex colors: each key is a color recognizable by matplotlib, and each corresponding entry is a list of vertices. If a vertex is not listed, it looks invisible on the resulting plot (it doesn't get drawn).
- `edge_colors` - a dictionary specifying edge colors: each key is a color recognized by matplotlib, and each entry is a list of edges.
- `partition` - a partition of the vertex set. if specified, plot will show each cell in a different color. vertex_colors takes precedence.
- `scaling_term` – default is 0.05. if vertices are getting chopped off, increase; if graph is too small, decrease. should be positive, but values much bigger than 1/8 won't be useful unless the vertices are huge
- `talk` - if true, prints large vertices with white backgrounds so that labels are legible on slides
- `iterations` - how many iterations of the spring layout algorithm to go through, if applicable
- `color_by_label` - if True, color edges by their labels
- `heights` - if specified, this is a dictionary from a set of floating point heights to a set of vertices
- `edge_style` - keyword arguments passed into the edge-drawing routine. This currently only works for directed graphs, since we pass off the undirected graph to networkx
- `tree_root` - a vertex of the tree to be used as the root for the layout="tree" option. If no root is specified, then one is chosen at random. Ignored unless layout='tree'.
- `tree_orientation` - "up" or "down" (default is "down"). If "up" (resp., "down"), then the root of the tree will appear on the bottom (resp., top) and the tree will grow upwards (resp. downwards). Ignored unless layout='tree'.
- `save_pos` - save position computed during plotting

EXAMPLES:

```
sage: from sage.graphs.graph_plot import graphplot_options
sage: list(sorted(graphplot_options.iteritems()))
[...]

sage: from math import sin, cos, pi
sage: P = graphs.PetersenGraph()
sage: d = {'#FF0000':[0,5], '#FF9900':[1,6], '#FFFF00':[2,7], '#00FF00':[3,8], '#0000FF':[4,
sage: pos_dict = {}
sage: for i in range(5):
...     x = float(cos(pi/2 + ((2*pi)/5)*i))
...     y = float(sin(pi/2 + ((2*pi)/5)*i))
...     pos_dict[i] = [x,y]
...
sage: for i in range(10)[5:]:
...     x = float(0.5*cos(pi/2 + ((2*pi)/5)*i))
...     y = float(0.5*sin(pi/2 + ((2*pi)/5)*i))
...     pos_dict[i] = [x,y]
...
sage: pl = P.plot(pos=pos_dict, vertex_colors=d)
sage: pl.show()

sage: C = graphs.CubeGraph(8)
sage: P = C.plot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: P.show()

sage: G = graphs.HeawoodGraph()
sage: for u,v,l in G.edges():
...     G.set_edge_label(u,v,'(' + str(u) + ',' + str(v) + ')')
sage: G.plot(edge_labels=True).show()

sage: D = DiGraph( { 0: [1, 10, 19], 1: [8, 2], 2: [3, 6], 3: [19, 4], 4: [17, 5], 5: [6, 15
sage: for u,v,l in D.edges():
...     D.set_edge_label(u,v,'(' + str(u) + ',' + str(v) + ')')
sage: D.plot(edge_labels=True, layout='circular').show()

sage: from sage.plot.plot import rainbow
sage: C = graphs.CubeGraph(5)
sage: R = rainbow(5)
sage: edge_colors = {}
sage: for i in range(5):
...     edge_colors[R[i]] = []
sage: for u,v,l in C.edges():
...     for i in range(5):
...         if u[i] != v[i]:
...             edge_colors[R[i]].append((u,v,l))
sage: C.plot(vertex_labels=False, vertex_size=0, edge_colors=edge_colors).show()

sage: D = graphs.DodecahedralGraph()
sage: Pi = [[6,5,15,14,7],[16,13,8,2,4],[12,17,9,3,1],[0,19,18,10,11]]
sage: D.show(partition=Pi)

sage: G = graphs.PetersenGraph()
sage: G.allow_loops(True)
sage: G.add_edge(0,0)
sage: G.show()
```

```
sage: D = DiGraph({0:[0,1], 1:[2], 2:[3]}, loops=True)
sage: D.show()
sage: D.show(edge_colors={(0,1,0):[(0,1,None),(1,2,None)],(0,0,0):[(2,3,None)]})

sage: pos = {0:[0.0, 1.5], 1:[-0.8, 0.3], 2:[-0.6, -0.8], 3:[0.6, -0.8], 4:[0.8, 0.3]}
sage: g = Graph({0:[1], 1:[2], 2:[3], 3:[4], 4:[0]})
sage: g.plot(pos=pos, layout='spring', iterations=0)

sage: G = Graph()
sage: P = G.plot()
sage: P.axes()
False
sage: G = DiGraph()
sage: P = G.plot()
sage: P.axes()
False

sage: G = graphs.PetersenGraph()
sage: G.get_pos()
{0: [6.12..., 1.0...],
 1: [-0.95..., 0.30...],
 2: [-0.58..., -0.80...],
 3: [0.58..., -0.80...],
 4: [0.95..., 0.30...],
 5: [1.53..., 0.5...],
 6: [-0.47..., 0.15...],
 7: [-0.29..., -0.40...],
 8: [0.29..., -0.40...],
 9: [0.47..., 0.15...]}
sage: P = G.plot(save_pos=True, layout='spring')

The following illustrates the format of a position dictionary,
but due to numerical noise we do not check the values themselves.

sage: G.get_pos()
{0: [..., ...],
 1: [..., ...],
 2: [..., ...],
 3: [..., ...],
 4: [..., ...],
 5: [..., ...],
 6: [..., ...],
 7: [..., ...],
 8: [..., ...],
 9: [..., ...]}

sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.plot(heights={0:[0], 1:[4,5,1], 2:[2], 3:[3,6]})

sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.plot(heights={0:[0], 1:[4,5,1], 2:[2], 3:[3,6]})
sage: t.set_edge_label(0,1,-7)
sage: t.set_edge_label(0,5,3)
sage: t.set_edge_label(0,5,99)
sage: t.set_edge_label(1,2,1000)
sage: t.set_edge_label(3,2,'spam')
```

```
sage: t.set_edge_label(2,6,3/2)
sage: t.set_edge_label(0,4,66)
sage: t.plot(heights={0:[0], 1:[4,5,1], 2:[2], 3:[3,6]}, edge_labels=True)

sage: T = list(graphs.trees(7))
sage: t = T[3]
sage: t.plot(layout='tree')

sage: t = DiGraph('JCC???@A??GO??CO??GO??')
sage: t.plot(layout='tree', tree_root=0, tree_orientation="up")
sage: D = DiGraph({0:[1,2,3], 2:[1,4], 3:[0]})
sage: D.plot()

sage: D = DiGraph(multiedges=True)
sage: for i in range(5):
...       D.add_edge((i,i+1,'a'))
...       D.add_edge((i,i-1,'b'))
sage: D.plot(edge_labels=True,edge_colors=D._color_by_label())

sage: g = Graph({}, loops=True, multiedges=True)
sage: g.add_edges([(0,0,'a'),(0,0,'b'),(0,1,'c'),(0,1,'d'),
...       (0,1,'e'),(0,1,'f'),(0,1,'f'),(2,1,'g'),(2,2,'h')])
sage: g.plot(edge_labels=True, color_by_label=True, edge_style='dashed')

sage: S = SupersingularModule(389)
sage: H = S.hecke_matrix(2)
sage: D = DiGraph(H)
sage: P = D.plot()

sage: G=Graph({'a':['a','b','b','b','e'],'b':['c','d','e'],'c':['c','d','d','d'],'d':['e']})
sage: G.show(pos={'a':[0,1],'b':[1,1],'c':[2,0],'d':[1,0],'e':[0,0]})
```

plot3d(*bgcolor=(1, 1, 1), vertex_colors=None, vertex_size=0.059999999999999998, edge_colors=None, edge_size=0.02, edge_size2=0.032500000000000001, pos3d=None, iterations=50, color_by_label=False, engine='jmol', **kwds*)

Plot a graph in three dimensions.

INPUT:

- `bgcolor` - rgb tuple (default: (1,1,1))
- `vertex_size` - float (default: 0.06)
- `vertex_colors` - optional dictionary to specify vertex colors: each key is a color recognizable by tachyon (rgb tuple (default: (1,0,0))), and each corresponding entry is a list of vertices. If a vertex is not listed, it looks invisible on the resulting plot (it doesn't get drawn).
- `edge_colors` - a dictionary specifying edge colors: each key is a color recognized by tachyon (default: (0,0,0)), and each entry is a list of edges.
- `edge_size` - float (default: 0.02)
- `edge_size2` - float (default: 0.0325), used for Tachyon sleeves
- `pos3d` - a position dictionary for the vertices
- `iterations` - how many iterations of the spring layout algorithm to go through, if applicable
- `engine` - which renderer to use. Options:
  - `'jmol'` - default
  - `'tachyon'`
- `xres` - resolution
- `yres` - resolution
- `**kwds` - passed on to the rendering engine

EXAMPLES:

```
sage: G = graphs.CubeGraph(5)
sage: G.plot3d(iterations=500, edge_size=None, vertex_size=0.04)
```

We plot a fairly complicated Cayley graph:

```
sage: A5 = AlternatingGroup(5); A5
Alternating group of order 5!/2 as a permutation group
sage: G = A5.cayley_graph()
sage: G.plot3d(vertex_size=0.03, edge_size=0.01, vertex_colors={(1,1,1):G.vertices()}, bgcol
```

Some Tachyon examples:

```
sage: D = graphs.DodecahedralGraph()
sage: P3D = D.plot3d(engine='tachyon')
sage: P3D.show() # long time

sage: G = graphs.PetersenGraph()
sage: G.plot3d(engine='tachyon', vertex_colors={(0,0,1):G.vertices()}).show() # long time

sage: C = graphs.CubeGraph(4)
sage: C.plot3d(engine='tachyon', edge_colors={(0,1,0):C.edges()}, vertex_colors={(1,1,1):C.v

sage: K = graphs.CompleteGraph(3)
sage: K.plot3d(engine='tachyon', edge_colors={(1,0,0):[(0,1,None)], (0,1,0):[(0,2,None)], (0
```

A directed version of the dodecahedron

```
sage: D = DiGraph( { 0: [1, 10, 19], 1: [8, 2], 2: [3, 6], 3: [19, 4], 4: [17, 5], 5: [6, 15
sage: D.plot3d().show() # long time

sage: P = graphs.PetersenGraph().to_directed()
sage: from sage.plot.all import rainbow
sage: edges = P.edges()
sage: R = rainbow(len(edges), 'rgbtuple')
sage: edge_colors = {}
sage: for i in range(len(edges)):
...       edge_colors[R[i]] = [edges[i]]
sage: P.plot3d(engine='tachyon', edge_colors=edge_colors).show() # long time

sage: G=Graph({'a':['a','b','b','b','e'],'b':['c','d','e'],'c':['c','d','d','d'],'d':['e']})
sage: G.show3d()
...
NotImplementedError: 3D plotting of multiple edges or loops not implemented.
```

radius()

Returns the radius of the (di)graph.

The radius is defined to be the minimum eccentricity of any vertex, where the eccentricity is the maximum distance to any other vertex.

EXAMPLES: The more symmetric a graph is, the smaller (diameter - radius) is.

```
sage: G = graphs.BarbellGraph(9, 3)
sage: G.radius()
3
sage: G.diameter()
6

sage: G = graphs.OctahedralGraph()
sage: G.radius()
```

```
2
sage: G.diameter()
2
```

random_subgraph (*p, inplace=False*)

Return a random subgraph that contains each vertex with prob. p.

EXAMPLE:

```
sage: P = graphs.PetersenGraph()
sage: P.random_subgraph(.25)
Subgraph of (Petersen graph): Graph on 4 vertices
```

relabel (*perm=None, inplace=True, return_map=False*)

Uses a dictionary, list, or permutation to relabel the (di)graph. If perm is a dictionary d, each old vertex v is a key in the dictionary, and its new label is d[v].

If perm is a list, we think of it as a map $i \mapsto perm[i]$ with the assumption that the vertices are $\{0, 1, ..., n-1\}$.

If perm is a permutation, the permutation is simply applied to the graph, under the assumption that the vertices are $\{0, 1, ..., n-1\}$. The permutation acts on the set $\{1, 2, ..., n\}$, where we think of $n = 0$.

If no arguments are provided, the graph is relabeled to be on the vertices $\{0, 1, ..., n-1\}$.

INPUT:

- `inplace` - default is True. If True, modifies the graph and returns nothing. If False, returns a relabeled copy of the graph.
- `return_map` - default is False. If True, returns the dictionary representing the map.

EXAMPLES:

```
sage: G = graphs.PathGraph(3)
sage: G.am()
[0 1 0]
[1 0 1]
[0 1 0]
```

Relabeling using a dictionary:

```
sage: G.relabel({1:2,2:1}, inplace=False).am()
[0 0 1]
[0 0 1]
[1 1 0]
```

Relabeling using a list:

```
sage: G.relabel([0,2,1], inplace=False).am()
[0 0 1]
[0 0 1]
[1 1 0]
```

Relabeling using a Sage permutation:

```
sage: from sage.groups.perm_gps.permgroup_named import SymmetricGroup
sage: S = SymmetricGroup(3)
sage: gamma = S('(1,2)')
sage: G.relabel(gamma, inplace=False).am()
[0 0 1]
[0 0 1]
[1 1 0]
```

Relabeling to simpler labels:

```
sage: G = graphs.CubeGraph(3)
sage: G.vertices()
['000', '001', '010', '011', '100', '101', '110', '111']
sage: G.relabel()
sage: G.vertices()
[0, 1, 2, 3, 4, 5, 6, 7]

sage: G = graphs.CubeGraph(3)
sage: expecting = {'000': 0, '001': 1, '010': 2, '011': 3, '100': 4, '101': 5, '110': 6, '11
sage: G.relabel(return_map=True) == expecting
True
```

TESTS:

```
sage: P = Graph(graphs.PetersenGraph(), sparse=True)
sage: P.delete_edge([0,1])
sage: P.add_edge((4,5))
sage: P.add_edge((2,6))
sage: P.delete_vertices([0,1])
sage: P.relabel()
```

The attributes are properly updated too

```
sage: G = graphs.PathGraph(5)
sage: G.set_vertices({0: 'before', 1: 'delete', 2: 'after'})
sage: G.set_boundary([1,2,3])
sage: G.delete_vertex(1)
sage: G.relabel()
sage: G.get_vertices()
{0: 'before', 1: 'after', 2: None, 3: None}
sage: G.get_boundary()
[1, 2]
sage: G.get_pos()
{0: [0, 0], 1: [2, 0], 2: [3, 0], 3: [4, 0]}
```

remove_loops(*vertices=None*)

Removes loops on vertices in vertices. If vertices is None, removes all loops.

EXAMPLE

```
sage: G = Graph(4, loops=True)
sage: G.add_edges( [ (0,0), (1,1), (2,2), (3,3), (2,3) ] )
sage: G.edges(labels=False)
[(0, 0), (1, 1), (2, 2), (2, 3), (3, 3)]
sage: G.remove_loops()
sage: G.edges(labels=False)
[(2, 3)]
sage: G.allows_loops()
True
sage: G.has_loops()
False

sage: D = DiGraph(4, loops=True)
sage: D.add_edges( [ (0,0), (1,1), (2,2), (3,3), (2,3) ] )
sage: D.edges(labels=False)
[(0, 0), (1, 1), (2, 2), (2, 3), (3, 3)]
sage: D.remove_loops()
sage: D.edges(labels=False)
[(2, 3)]
sage: D.allows_loops()
True
```

```
sage: D.has_loops()
False
```

remove_multiple_edges()

Removes all multiple edges, retaining one edge for each.

EXAMPLE:

```
sage: G = Graph(multiedges=True, implementation='networkx')
sage: G.add_edges( [ (0,1), (0,1), (0,1), (0,1), (1,2) ] )
sage: G.edges(labels=False)
[(0, 1), (0, 1), (0, 1), (0, 1), (1, 2)]

sage: G.remove_multiple_edges()
sage: G.edges(labels=False)
[(0, 1), (1, 2)]

sage: D = DiGraph(multiedges=True)
sage: D.add_edges( [ (0,1,1), (0,1,2), (0,1,3), (0,1,4), (1,2) ] )
sage: D.edges(labels=False)
[(0, 1), (0, 1), (0, 1), (0, 1), (1, 2)]
sage: D.remove_multiple_edges()
sage: D.edges(labels=False)
[(0, 1), (1, 2)]
```

set_boundary(*boundary*)

Sets the boundary of the (di)graph.

EXAMPLE:

```
sage: G = graphs.PetersenGraph()
sage: G.set_boundary([0,1,2,3,4])
sage: G.get_boundary()
[0, 1, 2, 3, 4]
sage: G.set_boundary((1..4))
sage: G.get_boundary()
[1, 2, 3, 4]
```

set_edge_label(*u, v, l*)

Set the edge label of a given edge.

Note: There can be only one edge from u to v for this to make sense. Otherwise, an error is raised.

INPUT:

- `u, v` - the vertices (and direction if digraph) of the edge
- `l` - the new label

EXAMPLES:

```
sage: SD = DiGraph( { 1:[18,2], 2:[5,3], 3:[4,6], 4:[7,2], 5:[4], 6:[13,12], 7:[18,8,10], 8:
sage: SD.set_edge_label(1, 18, 'discrete')
sage: SD.set_edge_label(4, 7, 'discrete')
sage: SD.set_edge_label(2, 5, 'h = 0')
sage: SD.set_edge_label(7, 18, 'h = 0')
sage: SD.set_edge_label(7, 10, 'aut')
sage: SD.set_edge_label(8, 10, 'aut')
sage: SD.set_edge_label(8, 9, 'label')
sage: SD.set_edge_label(8, 6, 'no label')
sage: SD.set_edge_label(13, 17, 'k > h')
sage: SD.set_edge_label(13, 14, 'k = h')
sage: SD.set_edge_label(17, 15, 'v_k finite')
sage: SD.set_edge_label(14, 15, 'v_k m.c.r.')
```

```
sage: posn = {1:[ 3,-3],  2:[0,2],  3:[0, 13],  4:[3,9],  5:[3,3],  6:[16, 13], 7:[6,1],  8:
sage: SD.plot(pos=posn, vertex_size=400, vertex_colors={'#FFFFFF':range(1,19)}, edge_labels=

sage: G = graphs.HeawoodGraph()
sage: for u,v,l in G.edges():
...     G.set_edge_label(u,v,'(' + str(u) + ',' + str(v) + ')')
sage: G.edges()
    [(0, 1, '(0,1)'),
     (0, 5, '(0,5)'),
     (0, 13, '(0,13)'),
     ...
     (11, 12, '(11,12)'),
     (12, 13, '(12,13)')]

sage: g = Graph({0: [0,1,1,2]}, loops=True, multiedges=True, implementation='networkx')
sage: g.set_edge_label(0,0,'test')
sage: g.edges()
[(0, 0, 'test'), (0, 1, None), (0, 1, None), (0, 2, None)]
sage: g.add_edge(0,0,'test2')
sage: g.set_edge_label(0,0,'test3')
...
RuntimeError: Cannot set edge label, since there are multiple edges from 0 to 0.

sage: dg = DiGraph({0 : [1], 1 : [0]}, sparse=True)
sage: dg.set_edge_label(0,1,5)
sage: dg.set_edge_label(1,0,9)
sage: dg.outgoing_edges(1)
[(1, 0, 9)]
sage: dg.incoming_edges(1)
[(0, 1, 5)]
sage: dg.outgoing_edges(0)
[(0, 1, 5)]
sage: dg.incoming_edges(0)
[(1, 0, 9)]

sage: G = Graph({0:{1:1}}, implementation='c_graph')
sage: G.num_edges()
1
sage: G.set_edge_label(0,1,1)
sage: G.num_edges()
1
```

set_embedding(*embedding*)

Sets a combinatorial embedding dictionary to _embedding attribute. Dictionary is organized with vertex labels as keys and a list of each vertex's neighbors in clockwise order.

Dictionary is error-checked for validity.

INPUT:

- `embedding` - a dictionary

EXAMPLES:

```
sage: G = graphs.PetersenGraph()
sage: G.set_embedding({0: [1, 5, 4], 1: [0, 2, 6], 2: [1, 3, 7], 3: [8, 2, 4], 4: [0, 9, 3],
sage: G.set_embedding({'s': [1, 5, 4], 1: [0, 2, 6], 2: [1, 3, 7], 3: [8, 2, 4], 4: [0, 9, 3
...
Exception: embedding is not valid for Petersen graph
```

set_planar_positions(*set_embedding=False, on_embedding=None, external_face=None, test=False, circular=False*)

Uses Schnyder's algorithm to determine positions for a planar embedding of self, raising an error if self is not planar.

INPUT:

- `set_embedding` - if True, sets the combinatorial embedding used (see self.get_embedding())
- `on_embedding` - dict: provide a combinatorial embedding
- `external_face` - ignored
- `test` - if True, perform sanity tests along the way
- `circular` - ignored

EXAMPLES:

```
sage: g = graphs.PathGraph(10)
sage: g.set_planar_positions(test=True)
True
sage: g = graphs.BalancedTree(3,4)
sage: g.set_planar_positions(test=True)
True
sage: g = graphs.CycleGraph(7)
sage: g.set_planar_positions(test=True)
True
sage: g = graphs.CompleteGraph(5)
sage: g.set_planar_positions(test=True,set_embedding=True)
...
Exception: Complete graph is not a planar graph.
```

set_pos (*pos*)

Sets the position dictionary, a dictionary specifying the coordinates of each vertex.

EXAMPLE: Note that set_pos will allow you to do ridiculous things, which will not blow up until plotting:

```
sage: G = graphs.PetersenGraph()
sage: G.get_pos()
{0: [..., ...],
 ...
 9: [..., ...]}

sage: G.set_pos('spam')
sage: P = G.plot()
...
TypeError: string indices must be integers
```

set_vertex (*vertex, object*)

Associate an arbitrary object with a vertex.

INPUT:

- `vertex` - which vertex
- `object` - object to associate to vertex

EXAMPLE:

```
sage: T = graphs.TetrahedralGraph()
sage: T.vertices()
[0, 1, 2, 3]
sage: T.set_vertex(1, graphs.FlowerSnark())
sage: T.get_vertex(1)
Flower Snark: Graph on 20 vertices
```

set_vertices (*vertex_dict*)

Associate arbitrary objects with each vertex, via an association dictionary.

INPUT:

- `vertex_dict` - the association dictionary

EXAMPLES:

```
sage: d = {0 : graphs.DodecahedralGraph(), 1 : graphs.FlowerSnark(), 2 : graphs.MoebiusKanto
sage: d[2]
Moebius-Kantor Graph: Graph on 16 vertices
sage: T = graphs.TetrahedralGraph()
sage: T.vertices()
[0, 1, 2, 3]
sage: T.set_vertices(d)
sage: T.get_vertex(1)
Flower Snark: Graph on 20 vertices
```

shortest_path(*u, v, by_weight=False, bidirectional=True*)

Returns a list of vertices representing some shortest path from u to v: if there is no path from u to v, the list is empty.

INPUT:

- `by_weight` - if False, uses a breadth first search. If True, takes edge weightings into account, using Dijkstra's algorithm.
- `bidirectional` - if True, the algorithm will expand vertices from u and v at the same time, making two spheres of half the usual radius. This generally doubles the speed (consider the total volume in each case).

EXAMPLE:

```
sage: D = graphs.DodecahedralGraph()
sage: D.shortest_path(4, 9)
[4, 17, 16, 12, 13, 9]
sage: D.shortest_path(5, 5)
[5]
sage: D.delete_edges(D.edges_incident(13))
sage: D.shortest_path(13, 4)
[]
sage: G = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} }, sparse=True )
sage: G.plot(edge_labels=True).show()
sage: G.shortest_path(0, 3)
[0, 4, 3]
sage: G.shortest_path(0, 3, by_weight=True)
[0, 1, 2, 3]
```

shortest_path_all_pairs(*by_weight=True, default_weight=1*)

Uses the Floyd-Warshall algorithm to find a shortest weighted path for each pair of vertices.

The weights (labels) on the vertices can be anything that can be compared and can be summed.

INPUT:

- `by_weight` - If False, figure distances by the numbers of edges.
- `default_weight` - (defaults to 1) The default weight to assign edges that don't have a weight (i.e., a label).

OUTPUT: A tuple (dist, pred). They are both dicts of dicts. The first indicates the length dist[u][v] of the shortest weighted path from u to v. The second is more complicated- it indicates the predecessor pred[u][v] of v in the shortest path from u to v.

EXAMPLE:

```
sage: G = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} }, sparse=True )
sage: G.plot(edge_labels=True).show()
sage: dist, pred = G.shortest_path_all_pairs()
sage: dist
```

```
{0: {0: 0, 1: 1, 2: 2, 3: 3, 4: 2}, 1: {0: 1, 1: 0, 2: 1, 3: 2, 4: 3}, 2: {0: 2, 1: 1, 2: 0,
sage: pred
{0: {0: None, 1: 0, 2: 1, 3: 2, 4: 0}, 1: {0: 1, 1: None, 2: 1, 3: 2, 4: 0}, 2: {0: 1, 1: 2,
sage: pred[0]
{0: None, 1: 0, 2: 1, 3: 2, 4: 0}
```

So for example the shortest weighted path from 0 to 3 is obtained as follows. The predecessor of 3 is pred[0][3] == 2, the predecessor of 2 is pred[0][2] == 1, and the predecessor of 1 is pred[0][1] == 0.

```
sage: G = Graph( { 0: {1:None}, 1: {2:None}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} }, sparse=True
sage: G.shortest_path_all_pairs(by_weight=False)
({0: {0: 0, 1: 1, 2: 2, 3: 2, 4: 1},
1: {0: 1, 1: 0, 2: 1, 3: 2, 4: 2},
2: {0: 2, 1: 1, 2: 0, 3: 1, 4: 2},
3: {0: 2, 1: 2, 2: 1, 3: 0, 4: 1},
4: {0: 1, 1: 2, 2: 2, 3: 1, 4: 0}},
{0: {0: None, 1: 0, 2: 1, 3: 4, 4: 0},
1: {0: 1, 1: None, 2: 1, 3: 2, 4: 0},
2: {0: 1, 1: 2, 2: None, 3: 2, 4: 3},
3: {0: 4, 1: 2, 2: 3, 3: None, 4: 3},
4: {0: 4, 1: 0, 2: 3, 3: 4, 4: None}})
sage: G.shortest_path_all_pairs()
({0: {0: 0, 1: 1, 2: 2, 3: 3, 4: 2},
1: {0: 1, 1: 0, 2: 1, 3: 2, 4: 3},
2: {0: 2, 1: 1, 2: 0, 3: 1, 4: 3},
3: {0: 3, 1: 2, 2: 1, 3: 0, 4: 2},
4: {0: 2, 1: 3, 2: 3, 3: 2, 4: 0}},
{0: {0: None, 1: 0, 2: 1, 3: 2, 4: 0},
1: {0: 1, 1: None, 2: 1, 3: 2, 4: 0},
2: {0: 1, 1: 2, 2: None, 3: 2, 4: 3},
3: {0: 1, 1: 2, 2: 3, 3: None, 4: 3},
4: {0: 4, 1: 0, 2: 3, 3: 4, 4: None}})
sage: G.shortest_path_all_pairs(default_weight=200)
({0: {0: 0, 1: 200, 2: 5, 3: 4, 4: 2},
1: {0: 200, 1: 0, 2: 200, 3: 201, 4: 202},
2: {0: 5, 1: 200, 2: 0, 3: 1, 4: 3},
3: {0: 4, 1: 201, 2: 1, 3: 0, 4: 2},
4: {0: 2, 1: 202, 2: 3, 3: 2, 4: 0}},
{0: {0: None, 1: 0, 2: 3, 3: 4, 4: 0},
1: {0: 1, 1: None, 2: 1, 3: 2, 4: 0},
2: {0: 4, 1: 2, 2: None, 3: 2, 4: 3},
3: {0: 4, 1: 2, 2: 3, 3: None, 4: 3},
4: {0: 4, 1: 0, 2: 3, 3: 4, 4: None}})
```

shortest_path_length(*u, v, by_weight=False, bidirectional=True, weight_sum=None*)

Returns the minimal length of paths from u to v: if there is no path from u to v, returns Infinity.

INPUT:

- `by_weight` - if False, uses a breadth first search. If True, takes edge weightings into account, using Dijkstra's algorithm.
- `bidirectional` - if True, the algorithm will expand vertices from u and v at the same time, making two spheres of half the usual radius. This generally doubles the speed (consider the total volume in each case).
- `weight_sum` - if False, returns the number of edges in the path. If True, returns the sum of the weights of these edges. Default behavior is to have the same value as by_weight.

EXAMPLE:

```
sage: D = graphs.DodecahedralGraph()
sage: D.shortest_path_length(4, 9)
5
sage: D.shortest_path_length(5, 5)
0
sage: D.delete_edges(D.edges_incident(13))
sage: D.shortest_path_length(13, 4)
+Infinity
sage: G = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} }, sparse=True )
sage: G.plot(edge_labels=True).show()
sage: G.shortest_path_length(0, 3)
2
sage: G.shortest_path_length(0, 3, by_weight=True)
3
```

shortest_path_lengths(*u, by_weight=False, weight_sums=None*)

Returns a dictionary of shortest path lengths keyed by targets that are connected by a path from u.

INPUT:

- `by_weight` - if False, uses a breadth first search. If True, takes edge weightings into account, using Dijkstra's algorithm.
- `weight_sums` - if False, returns the number of edges in each path. If True, returns the sum of the weights of these edges. Default behavior is to have the same value as by_weight.

EXAMPLES:

```
sage: D = graphs.DodecahedralGraph()
sage: D.shortest_path_lengths(0)
{0: 0, 1: 1, 2: 2, 3: 2, 4: 3, 5: 4, 6: 3, 7: 3, 8: 2, 9: 2, 10: 1, 11: 2, 12: 3, 13: 3, 14:
sage: G = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} }, sparse=True )
sage: G.plot(edge_labels=True).show()
sage: G.shortest_path_lengths(0, by_weight=True)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 2}
```

shortest_paths(*u, by_weight=False, cutoff=None*)

Returns a dictionary d of shortest paths d[v] from u to v, for each vertex v connected by a path from u.

INPUT:

- `by_weight` - if False, uses a breadth first search. If True, uses Dijkstra's algorithm to find the shortest paths by weight.
- `cutoff` - integer depth to stop search. Ignored if by_weight is True.

EXAMPLES:

```
sage: D = graphs.DodecahedralGraph()
sage: D.shortest_paths(0)
{0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 19, 3], 4: [0, 19, 3, 4], 5: [0, 19, 3, 4, 5], 6: [
sage: D.shortest_paths(0, cutoff=2)
{0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 19, 3], 8: [0, 1, 8], 9: [0, 10, 9], 10: [0, 10], 1
sage: G = Graph( { 0: {1: 1}, 1: {2: 1}, 2: {3: 1}, 3: {4: 2}, 4: {0: 2} }, sparse=True )
sage: G.plot(edge_labels=True).show()
sage: G.shortest_paths(0, by_weight=True)
{0: [0], 1: [0, 1], 2: [0, 1, 2], 3: [0, 1, 2, 3], 4: [0, 4]}
```

show(***kwds*)

Shows the (di)graph.

For syntax and lengthy documentation, see G.plot?. Any options not used by plot will be passed on to the Graphics.show method.

EXAMPLE:

```
sage: C = graphs.CubeGraph(8)
sage: P = C.plot(vertex_labels=False, vertex_size=0, graph_border=True)
sage: P.show()
```

show3d(*bgcolor=(1, 1, 1), vertex_colors=None, vertex_size=0.059999999999999998, edge_colors=None, edge_size=0.02, edge_size2=0.032500000000000001, pos3d=None, iterations=50, color_by_label=False, engine='jmol', **kwds*)

Plots the graph using Tachyon, and shows the resulting plot.

INPUT:

- `bgcolor` - rgb tuple (default: (1,1,1))
- `vertex_size` - float (default: 0.06)
- `vertex_colors` - optional dictionary to specify vertex colors: each key is a color recognizable by tachyon (rgb tuple (default: (1,0,0))), and each corresponding entry is a list of vertices. If a vertex is not listed, it looks invisible on the resulting plot (it doesn't get drawn).
- `edge_colors` - a dictionary specifying edge colors: each key is a color recognized by tachyon (default: (0,0,0)), and each entry is a list of edges.
- `edge_size` - float (default: 0.02)
- `edge_size2` - float (default: 0.0325), used for Tachyon sleeves
- `pos3d` - a position dictionary for the vertices
- `iterations` - how many iterations of the spring layout algorithm to go through, if applicable
- `engine` - which renderer to use. Options:
- `'jmol'` - default 'tachyon'
- `xres` - resolution
- `yres` - resolution
- `**kwds` - passed on to the Tachyon command

EXAMPLES:

```
sage: G = graphs.CubeGraph(5)
sage: G.show3d(iterations=500, edge_size=None, vertex_size=0.04)
```

We plot a fairly complicated Cayley graph:

```
sage: A5 = AlternatingGroup(5); A5
Alternating group of order 5!/2 as a permutation group
sage: G = A5.cayley_graph()
sage: G.show3d(vertex_size=0.03, edge_size=0.01, edge_size2=0.02, vertex_colors={(1,1,1):G.v
```

Some Tachyon examples:

```
sage: D = graphs.DodecahedralGraph()
sage: D.show3d(engine='tachyon') # long time

sage: G = graphs.PetersenGraph()
sage: G.show3d(engine='tachyon', vertex_colors={(0,0,1):G.vertices()}) # long time

sage: C = graphs.CubeGraph(4)
sage: C.show3d(engine='tachyon', edge_colors={(0,1,0):C.edges()}, vertex_colors={(1,1,1):C.v

sage: K = graphs.CompleteGraph(3)
sage: K.show3d(engine='tachyon', edge_colors={(1,0,0):[(0,1,None)], (0,1,0):[(0,2,None)], (0
```

size()

Returns the number of edges.

EXAMPLE:

```
sage: G = graphs.PetersenGraph()
sage: G.size()
15
```

spectrum(*laplacian=False*)

Returns the spectrum of the graph, the eigenvalues of the adjacency matrix

INPUT:

- `laplacian` - if True, use the Laplacian matrix instead (see self.kirchhoff_matrix())

EXAMPLE:

```
sage: P = graphs.PetersenGraph()
sage: P.spectrum()
[-2.0, -2.0, -2.0, -2.0, 1.0, 1.0, 1.0, 1.0, 1.0, 3.0]
```

Due to numerical imprecision the first entry for the spectrum below is zero or near zero:

```
sage: P.spectrum(laplacian=True)
[..., 2.0, 2.0, 2.0, 2.0, 2.0, 5.0, 5.0, 5.0, 5.0]

sage: D = P.to_directed()
sage: D.delete_edge(7,9)
sage: D.spectrum()
[-2.0, -2.0, -2.0, -1.7..., 0.8..., 1.0, 1.0, 1.0, 1.0, 2.9...]
```

strong_product(*other*)

Returns the strong product of self and other.

The strong product of G and H is the graph L with vertex set V(L) equal to the Cartesian product of the vertices V(G) and V(H), and ((u,v), (w,x)) is an edge iff either - (u, w) is an edge of self and v = x, or - (v, x) is an edge of other and u = w, or - (u, w) is an edge of self and (v, x) is an edge of other. In other words, the edges of the strong product is the union of the edges of the tensor and Cartesian products.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: C = graphs.CycleGraph(5)
sage: S = C.strong_product(Z); S
Graph on 10 vertices
sage: S.plot().show()

sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: S = D.strong_product(P); S
Graph on 200 vertices
sage: S.plot().show()
```

subgraph(*vertices=None, edges=None, inplace=False, vertex_property=None, edge_property=None*)

Returns the subgraph containing the given vertices and edges. If either vertices or edges are not specified, they are assumed to be all vertices or edges. If edges are not specified, returns the subgraph induced by the vertices.

INPUT:

- `inplace` - Using inplace is True will simply delete the extra vertices and edges from the current graph. This will modify the graph.
- `vertices` - Vertices can be a single vertex or an iterable container of vertices, e.g. a list, set, graph, file or numeric array. If not passed, defaults to the entire graph.
- `edges` - As with vertices, edges can be a single edge or an iterable container of edges (e.g., a list, set, file, numeric array, etc.). If not edges are not specified, then all edges are assumed and the returned graph is an induced subgraph. In the case of multiple edges, specifying an edge as (u,v) means to keep all edges (u,v), regardless of the label.

- `vertex_property` - If specified, this is expected to be a function on vertices, which is intersected with the vertices specified, if any are.
- `edge_property` - If specified, this is expected to be a function on edges, which is intersected with the edges specified, if any are.

EXAMPLES:

```
sage: G = graphs.CompleteGraph(9)
sage: H = G.subgraph([0,1,2]); H
Subgraph of (Complete graph): Graph on 3 vertices
sage: G
Complete graph: Graph on 9 vertices
sage: J = G.subgraph(edges=[(0,1)])
sage: J.edges(labels=False)
[(0, 1)]
sage: J.vertices()==G.vertices()
True
sage: G.subgraph([0,1,2], inplace=True); G
Subgraph of (Complete graph): Graph on 3 vertices
sage: G.subgraph()==G
True

sage: D = graphs.CompleteGraph(9).to_directed()
sage: H = D.subgraph([0,1,2]); H
Subgraph of (Complete graph): Digraph on 3 vertices
sage: H = D.subgraph(edges=[(0,1), (0,2)])
sage: H.edges(labels=False)
[(0, 1), (0, 2)]
sage: H.vertices()==D.vertices()
True
sage: D
Complete graph: Digraph on 9 vertices
sage: D.subgraph([0,1,2], inplace=True); D
Subgraph of (Complete graph): Digraph on 3 vertices
sage: D.subgraph()==D
True
```

A more complicated example involving multiple edges and labels.

```
sage: G = Graph(multiedges=True, implementation='networkx')
sage: G.add_edges([(0,1,'a'), (0,1,'b'), (1,0,'c'), (0,2,'d'), (0,2,'e'), (2,0,'f'), (1,2,'g
sage: G.subgraph(edges=[(0,1), (0,2,'d'), (0,2,'not in graph')]).edges()
[(0, 1, 'a'), (0, 1, 'b'), (0, 1, 'c'), (0, 2, 'd')]
sage: J = G.subgraph(vertices=[0,1], edges=[(0,1,'a'), (0,2,'c')])
sage: J.edges()
[(0, 1, 'a')]
sage: J.vertices()
[0, 1]

sage: D = DiGraph(multiedges=True, implementation='networkx')
sage: D.add_edges([(0,1,'a'), (0,1,'b'), (1,0,'c'), (0,2,'d'), (0,2,'e'), (2,0,'f'), (1,2,'g
sage: D.subgraph(edges=[(0,1), (0,2,'d'), (0,2,'not in graph')]).edges()
[(0, 1, 'a'), (0, 1, 'b'), (0, 2, 'd')]
sage: H = D.subgraph(vertices=[0,1], edges=[(0,1,'a'), (0,2,'c')])
sage: H.edges()
[(0, 1, 'a')]
sage: H.vertices()
[0, 1]
```

Using the property arguments:

```
sage: P = graphs.PetersenGraph()
sage: S = P.subgraph(vertex_property = lambda v : v%2 == 0)
sage: S.vertices()
[0, 2, 4, 6, 8]

sage: C = graphs.CubeGraph(2)
sage: S = C.subgraph(edge_property=(lambda e: e[0][0] == e[1][0]))
sage: C.edges()
[('00', '01', None),
 ('10', '00', None),
 ('11', '01', None),
 ('11', '10', None)]
sage: S.edges()
[('00', '01', None), ('11', '10', None)]
```

TESTS: We should delete unused _pos dictionary entries

```
sage: g = graphs.PathGraph(10)
sage: h = g.subgraph([3..5])
sage: h._pos.keys()
[3, 4, 5]
```

tensor_product(*other*)

Returns the tensor product, also called the categorical product, of self and other.

The tensor product of G and H is the graph L with vertex set V(L) equal to the Cartesian product of the vertices V(G) and V(H), and ((u,v), (w,x)) is an edge iff - (u, w) is an edge of self, and - (v, x) is an edge of other.

EXAMPLES:

```
sage: Z = graphs.CompleteGraph(2)
sage: C = graphs.CycleGraph(5)
sage: T = C.tensor_product(Z); T
Graph on 10 vertices
sage: T.plot().show()

sage: D = graphs.DodecahedralGraph()
sage: P = graphs.PetersenGraph()
sage: T = D.tensor_product(P); T
Graph on 200 vertices
sage: T.plot().show()
```

to_simple()

Returns a simple version of itself (i.e., undirected and loops and multiple edges are removed).

EXAMPLE:

```
sage: G = DiGraph(loops=True,multiedges=True,sparse=True)
sage: G.add_edges( [ (0,0), (1,1), (2,2), (2,3,1), (2,3,2), (3,2) ] )
sage: G.edges(labels=False)
[(0, 0), (1, 1), (2, 2), (2, 3), (2, 3), (3, 2)]
sage: H=G.to_simple()
sage: H.edges(labels=False)
[(2, 3)]
sage: H.is_directed()
False
sage: H.allows_loops()
False
sage: H.allows_multiple_edges()
False
```

trace_faces(*comb_emb*)

A helper function for finding the genus of a graph. Given a graph and a combinatorial embedding (rot_sys), this function will compute the faces (returned as a list of lists of edges (tuples) of the particular embedding.

Note - rot_sys is an ordered list based on the hash order of the vertices of graph. To avoid confusion, it might be best to set the rot_sys based on a 'nice_copy' of the graph.

INPUT:

- `comb_emb` - a combinatorial embedding dictionary. Format: v1:[v2,v3], v2:[v1], v3:[v1] (clockwise ordering of neighbors at each vertex.)

EXAMPLES:

```
sage: T = graphs.TetrahedralGraph()
sage: T.trace_faces({0: [1, 3, 2], 1: [0, 2, 3], 2: [0, 3, 1], 3: [0, 1, 2]})
[[(0, 1), (1, 2), (2, 0)],
 [(3, 2), (2, 1), (1, 3)],
 [(2, 3), (3, 0), (0, 2)],
 [(0, 3), (3, 1), (1, 0)]]
```

transitive_closure()

Computes the transitive closure of a graph and returns it. The original graph is not modified.

The transitive closure of a graph G has an edge (x,y) if and only if there is a path between x and y in G.

The transitive closure of any strongly connected component of a graph is a complete graph. In particular, the transitive closure of a connected undirected graph is a complete graph. The transitive closure of a directed acyclic graph is a directed acyclic graph representing the full partial order.

EXAMPLES:

```
sage: g=graphs.PathGraph(4)
sage: g.transitive_closure()==graphs.CompleteGraph(4)
True
sage: g=DiGraph({0:[1,2], 1:[3], 2:[4,5]})
sage: g.transitive_closure().edges(labels=False)
[(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (1, 3), (2, 4), (2, 5)]
```

transitive_reduction()

Returns a transitive reduction of a graph. The original graph is not modified.

A transitive reduction H of G has a path from x to y if and only if there was a path from x to y in G. Deleting any edge of H destroys this property. A transitive reduction is not unique in general. A transitive reduction has the same transitive closure as the original graph.

A transitive reduction of a complete graph is a tree. A transitive reduction of a tree is itself.

EXAMPLES:

```
sage: g=graphs.PathGraph(4)
sage: g.transitive_reduction()==g
True
sage: g=graphs.CompleteGraph(5)
sage: edges = g.transitive_reduction().edges(); len(edges)
4
sage: g=DiGraph({0:[1,2], 1:[2,3,4,5], 2:[4,5]})
sage: g.transitive_reduction().size()
5
```

union(*other*)

Returns the union of self and other.

If the graphs have common vertices, the common vertices will be identified.

EXAMPLE:

```
sage: G = graphs.CycleGraph(3)
sage: H = graphs.CycleGraph(4)
sage: J = G.union(H); J
Graph on 4 vertices
sage: J.vertices()
[0, 1, 2, 3]
sage: J.edges(labels=False)
[(0, 1), (0, 2), (0, 3), (1, 2), (2, 3)]
```

vertex_boundary(*vertices1, vertices2=None*)

Returns a list of all vertices in the external boundary of vertices1, intersected with vertices2. If vertices2 is None, then vertices2 is the complement of vertices1. This is much faster if vertices1 is smaller than vertices2.

The external boundary of a set of vertices is the union of the neighborhoods of each vertex in the set. Note that in this implementation, since vertices2 defaults to the complement of vertices1, if a vertex v has a loop, then vertex_boundary(v) will not contain v.

In a digraph, the external boundary of a vertex v are those vertices u with an arc (v, u).

EXAMPLE:

```
sage: G = graphs.CubeGraph(4)
sage: l = ['0111', '0000', '0001', '0011', '0010', '0101', '0100', '1111', '1101', '1011', '
sage: G.vertex_boundary(['0000', '1111'], l)
['0111', '0001', '0010', '0100', '1101', '1011']

sage: D = DiGraph({0:[1,2], 3:[0]})
sage: D.vertex_boundary([0])
[1, 2]
```

vertex_iterator(*vertices=None*)

Returns an iterator over the given vertices. Returns False if not given a vertex, sequence, iterator or None. None is equivalent to a list of every vertex. Note that `for v in G` syntax is allowed.

INPUT:

- `vertices` - iterated vertices are these intersected with the vertices of the (di)graph

EXAMPLE:

```
sage: P = graphs.PetersenGraph()
sage: for v in P.vertex_iterator():
...     print v
...
0
1
2
...
8
9

sage: G = graphs.TetrahedralGraph()
sage: for i in G:
...     print i
0
1
2
3
```

Note that since the intersection option is available, the vertex_iterator() function is sub-optimal, speedwise, but note the following optimization:

```
sage: timeit V = P.vertices()                    # not tested
100000 loops, best of 3: 8.85 [micro]s per loop
sage: timeit V = list(P.vertex_iterator())       # not tested
100000 loops, best of 3: 5.74 [micro]s per loop
sage: timeit V = list(P._nxg.adj.iterkeys())     # not tested
100000 loops, best of 3: 3.45 [micro]s per loop
```

In other words, if you want a fast vertex iterator, call the dictionary directly.

vertices(*boundary_first=False*)

Return a list of the vertices.

INPUT:

- `boundary_first` - Return the boundary vertices first.

EXAMPLE:

```
sage: P = graphs.PetersenGraph()
sage: P.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Note that the output of the vertices() function is always sorted. This is sub-optimal, speedwise, but note the following optimizations:

```
sage: timeit V = P.vertices()                      # not tested
100000 loops, best of 3: 8.85 [micro]s per loop
sage: timeit V = list(P.vertex_iterator())         # not tested
100000 loops, best of 3: 5.74 [micro]s per loop
sage: timeit V = list(P._nxg.adj.iterkeys())       # not tested
100000 loops, best of 3: 3.45 [micro]s per loop
```

In other words, if you want a fast vertex iterator, call the dictionary directly.

weighted(*new=None*)

Returns whether the (di)graph is to be considered as a weighted (di)graph.

Note that edge weightings can still exist for (di)graphs G where G.weighted() is False.

EXAMPLE: Here we have two graphs with different labels, but weighted is False for both, so we just check for the presence of edges:

```
sage: G = Graph({0:{1:'a'}}, implementation='networkx')
sage: H = Graph({0:{1:'b'}}, implementation='networkx')
sage: G == H
True
```

Now one is weighted and the other is not, and thus the graphs are not equal.

:: sage: G.weighted(True) sage: H.weighted() False sage: G == H False

However, if both are weighted, then we finally compare 'a' to 'b'.

```
sage: H.weighted(True)
sage: G == H
False
```

weighted_adjacency_matrix(*sparse=True, boundary_first=False*)

Returns the weighted adjacency matrix of the graph. Each vertex is represented by its position in the list returned by the vertices() function.

EXAMPLES:

```
sage: G = Graph(sparse=True, weighted=True)
sage: G.add_edges([(0,1,1),(1,2,2),(0,2,3),(0,3,4)])
sage: M = G.weighted_adjacency_matrix(); M
[0 1 3 4]
```

```
[1 0 2 0]
[3 2 0 0]
[4 0 0 0]
sage: H = Graph(data=M, format='weighted_adjacency_matrix', sparse=True)
sage: H == G
True
```

The following doctest verifies that #4888 is fixed:

```
sage: G = DiGraph({0:{}, 1:{0:1}, 2:{0:1}}, weighted = True)
sage: G.weighted_adjacency_matrix()
[0 0 0]
[1 0 0]
[1 0 0]
```

class Graph(*data=None, pos=None, loops=None, format=None, boundary=, [], weighted=None, implementation='networkx', sparse=True, vertex_labels=True, **kwds*)

Undirected graph.

INPUT:

- `data` - can be any of the following:
  1. A NetworkX digraph
  2. A dictionary of dictionaries
  3. A dictionary of lists
  4. A numpy matrix or ndarray
  5. A Sage adjacency matrix or incidence matrix
  6. A pygraphviz agraph
  7. A SciPy sparse matrix
- `pos` - a positioning dictionary: for example, the spring layout from NetworkX for the 5-cycle is:

```
{0: [-0.91679746, 0.88169588],
 1: [ 0.47294849, 1.125     ],
 2: [ 1.125     ,-0.12867615],
 3: [ 0.12743933,-1.125     ],
 4: [-1.125     ,-0.50118505]}
```

- `name` - (must be an explicitly named parameter, i.e., name="complete") gives the graph a name
- `loops` - boolean, whether to allow loops (ignored if data is an instance of the Graph class)
- `multiedges` - boolean, whether to allow multiple edges (ignored if data is an instance of the Graph class)
- `weighted` - whether graph thinks of itself as weighted or not. See self.weighted()
- `format` - if None, Graph tries to guess- can be several values, including:
  - `'graph6'` - Brendan McKay's graph6 format, in a string (if the string has multiple graphs, the first graph is taken)
  - `'sparse6'` - Brendan McKay's sparse6 format, in a string (if the string has multiple graphs, the first graph is taken)
  - `'adjacency_matrix'` - a square Sage matrix M, with M[i,j] equal to the number of edges {i,j}
  - `'weighted_adjacency_matrix'` - a square Sage matrix M, with M[i,j] equal to the weight of the single edge {i,j}. Given this format, weighted is ignored (assumed True).
  - `'incidence_matrix'` - a Sage matrix, with one column C for each edge, where if C represents {i, j}, C[i] is -1 and C[j] is 1
  - `'elliptic_curve_congruence'` - data must be an iterable container of elliptic curves, and the graph produced has each curve as a vertex (it's Cremona label) and an edge E-F labelled p if and only if E is congruent to F mod p

- `boundary` - a list of boundary vertices, if empty, graph is considered as a 'graph without boundary'
- `implementation` - what to use as a backend for the graph. Currently, the options are either 'networkx' or 'c_graph'
- `sparse` - only for implementation == 'c_graph'. Whether to use sparse or dense graphs as backend. Note that currently dense graphs do not have edge labels, nor can they be multigraphs
- `vertex_labels` - only for implementation == 'c_graph'. Whether to allow any object as a vertex (slower), or only the integers 0, ..., n-1, where n is the number of vertices.

EXAMPLES: We illustrate the first six input formats (the other two involve packages that are currently not standard in Sage):

1. A NetworkX XGraph:

```
sage: import networkx
sage: g = networkx.XGraph({0:[1,2,3], 2:[4]})
sage: Graph(g)
Graph on 5 vertices
```

2. A NetworkX graph:

```
sage: import networkx
sage: g = networkx.Graph({0:[1,2,3], 2:[4]})
sage: DiGraph(g)
Digraph on 5 vertices
```

Note that in all cases, we copy the NetworkX structure.

```
sage: import networkx
sage: g = networkx.Graph({0:[1,2,3], 2:[4]})
sage: G = Graph(g, implementation='networkx')
sage: H = Graph(g, implementation='networkx')
sage: G._backend._nxg is H._backend._nxg
False
```

1. A dictionary of dictionaries:

```
sage: g = Graph({0:{1:'x',2:'z',3:'a'}, 2:{5:'out'}}, implementation='networkx'); g
Graph on 5 vertices
```

The labels ('x', 'z', 'a', 'out') are labels for edges. For example, 'out' is the label for the edge on 2 and 5. Labels can be used as weights, if all the labels share some common parent.

sage: a,b,c,d,e,f = sorted(SymmetricGroup(3)) sage: Graph({b:{d:'c',e:'p'}, c:{d:'p',e:'c'}}) Graph on 4 vertices

2. A dictionary of lists:

```
sage: g = Graph({0:[1,2,3], 2:[4]}); g
Graph on 5 vertices
```

3. A list of vertices and a function describing adjacencies. Note that the list of vertices and the function must be enclosed in a list (i.e., [list of vertices, function]).

Construct the Paley graph over GF(13).

```
sage: g=Graph([GF(13), lambda i,j: i!=j and (i-j).is_square()], implementation='networkx')
sage: g.vertices()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
sage: g.adjacency_matrix()
[0 1 0 1 1 0 0 0 0 1 1 0 1]
```

```
[1 0 1 0 1 1 0 0 0 0 1 1 0]
[0 1 0 1 0 1 1 0 0 0 0 1 1]
[1 0 1 0 1 0 1 1 0 0 0 0 1]
[1 1 0 1 0 1 0 1 1 0 0 0 0]
[0 1 1 0 1 0 1 0 1 1 0 0 0]
[0 0 1 1 0 1 0 1 0 1 1 0 0]
[0 0 0 1 1 0 1 0 1 0 1 1 0]
[0 0 0 0 1 1 0 1 0 1 0 1 1]
[1 0 0 0 0 1 1 0 1 0 1 0 1]
[1 1 0 0 0 0 1 1 0 1 0 1 0]
[0 1 1 0 0 0 0 1 1 0 1 0 1]
[1 0 1 1 0 0 0 0 1 1 0 1 0]
```

Construct the line graph of a complete graph.

```
sage: g=graphs.CompleteGraph(4)
sage: line_graph=Graph([g.edges(labels=false), \
       lambda i,j: len(set(i).intersection(set(j)))>0], \
       implementation='networkx', loops=False)
sage: line_graph.vertices()
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: line_graph.adjacency_matrix()
[0 1 1 1 1 0]
[1 0 1 1 0 1]
[1 1 0 0 1 1]
[1 1 0 0 1 1]
[1 0 1 1 0 1]
[0 1 1 1 1 0]
```

4.A numpy matrix or ndarray:

```
sage: import numpy
sage: A = numpy.array([[0,1,1],[1,0,1],[1,1,0]])
sage: Graph(A, implementation='networkx')
Graph on 3 vertices
```

5.A graph6 or sparse6 string: Sage automatically recognizes whether a string is in graph6 or sparse6 format:

```
sage: s = ':I`AKGsaOs`cI]Gb~'
sage: Graph(s, sparse=True)
Looped multi-graph on 10 vertices

sage: G = Graph('G?????')
sage: G = Graph("G'?G?C")
...
RuntimeError: The string seems corrupt: valid characters are
?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
sage: G = Graph('G??????')
...
RuntimeError: The string (G??????) seems corrupt: for n = 8, the string is too long.

sage: G = Graph(":I'AKGsaOs`cI]Gb~")
...
RuntimeError: The string seems corrupt: valid characters are
?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abcdefghijklmnopqrstuvwxyz{|}~
```

There are also list functions to take care of lists of graphs:

```
sage: s = ':IgMoqoCUOqeb\n:I`AKGsaOs`cI]Gb~\n:I`EDOAEQ?PccSsge\N\n'
sage: graphs_list.from_sparse6(s)
```

```
[Looped multi-graph on 10 vertices, Looped multi-graph on 10 vertices, Looped multi-graph or
```

6.A Sage matrix: Note: If format is not specified, then Sage assumes a symmetric square matrix is an adjacency matrix, otherwise an incidence matrix.

•an adjacency matrix:

```
sage: M = graphs.PetersenGraph().am(); M
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
sage: Graph(M)
Graph on 10 vertices

sage: Graph(matrix([[1,2],[2,4]]),loops=True)
Looped multi-graph on 2 vertices

sage: M = Matrix([[0,1,-1],[1,0,-1/2],[-1,-1/2,0]]); M
[   0    1   -1]
[   1    0 -1/2]
[  -1 -1/2    0]
sage: G = Graph(M); G
Graph on 3 vertices
sage: G.weighted()
True
```

•an incidence matrix:

```
sage: M = Matrix(6, [-1,0,0,0,1, 1,-1,0,0,0, 0,1,-1,0,0, 0,0,1,-1,0, 0,0,0,1,-1, 0,0,0,0
[-1  0  0  0  1]
[ 1 -1  0  0  0]
[ 0  1 -1  0  0]
[ 0  0  1 -1  0]
[ 0  0  0  1 -1]
[ 0  0  0  0  0]
sage: Graph(M)
Graph on 6 vertices

sage: Graph(Matrix([[1],[1],[1]]))
...
ValueError: Non-symmetric or non-square matrix assumed to be an incidence matrix: There n
sage: Graph(Matrix([[1],[1],[0]]))
...
ValueError: Non-symmetric or non-square matrix assumed to be an incidence matrix: Each co

sage: M = Matrix([[0,1,-1],[1,0,-1],[-1,-1,0]]); M
[ 0  1 -1]
[ 1  0 -1]
[-1 -1  0]
sage: Graph(M)
Graph on 3 vertices

sage: M = Matrix([[0,1,1],[1,0,0],[0,0,0]]); M
```

```
[0 1 1]
[1 0 0]
[0 0 0]
sage: Graph(M)
...
ValueError: Non-symmetric or non-square matrix assumed to be an incidence matrix: There r

sage: M = Matrix([[0,1,1],[1,0,1],[-1,-1,0]]); M
[ 0  1  1]
[ 1  0  1]
[-1 -1  0]
sage: Graph(M)
...
ValueError: Non-symmetric or non-square matrix assumed to be an incidence matrix: Each c
```

bipartite_color()

Returns a dictionary with vertices as the keys and the color class as the values. Fails with an error if the graph is not bipartite.

EXAMPLE:

```
sage: graphs.CycleGraph(4).bipartite_color()
{0: 1, 1: 0, 2: 1, 3: 0}
sage: graphs.CycleGraph(5).bipartite_color()
...
RuntimeError: Graph is not bipartite.
```

bipartite_sets()

Returns (X,Y) where X and Y are the nodes in each bipartite set of graph G. Fails with an error if graph is not bipartite.

EXAMPLE:

```
sage: graphs.CycleGraph(4).bipartite_sets()
([0, 2], [1, 3])
sage: graphs.CycleGraph(5).bipartite_sets()
...
RuntimeError: Graph is not bipartite.
```

centrality_betweenness(*normalized=True*)

Returns the betweenness centrality (fraction of number of shortest paths that go through each vertex) as a dictionary keyed by vertices. The betweenness is normalized by default to be in range (0,1). This wraps Networkx's implementation of the algorithm described in [1].

Measures of the centrality of a vertex within a graph determine the relative importance of that vertex to its graph. Vertices that occur on more shortest paths between other vertices have higher betweenness than vertices that occur on less.

INPUT:

- `normalized` - boolean (default True) - if set to False, result is not normalized.

REFERENCE:

- [1] Ulrik Brandes. (2003). Faster Evaluation of Shortest-Path Based Centrality Indices. [Online] Available: http://citeseer.nj.nec.com/brandes00faster.html

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).centrality_betweenness()
{0: 0.069696969696969688, 1: 0.069696969696969688, 2: 0.060606060606060601, 3: 0.0606060606
sage: (graphs.ChvatalGraph()).centrality_betweenness(normalized=False)
{0: 7.6666666666666661, 1: 7.6666666666666661, 2: 6.6666666666666661, 3: 6.6666666666666661,
```

```
sage: D = DiGraph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: D.show(figsize=[2,2])
sage: D = D.to_undirected()
sage: D.show(figsize=[2,2])
sage: D.centrality_betweenness()
{0: 0.16666666666666666, 1: 0.16666666666666666, 2: 0.0, 3: 0.0}
```

centrality_closeness(*v=None*)

Returns the closeness centrality (1/average distance to all vertices) as a dictionary of values keyed by vertex. The degree centrality is normalized to be in range (0,1).

Measures of the centrality of a vertex within a graph determine the relative importance of that vertex to its graph. 'Closeness centrality may be defined as the total graph-theoretic distance of a given vertex from all other vertices... Closeness is an inverse measure of centrality in that a larger value indicates a less central actor while a smaller value indicates a more central actor,' [1].

INPUT:

- v - a vertex label (to find degree centrality of only one vertex)

REFERENCE:

- [1] Stephen P Borgatti. (1995). Centrality and AIDS. [Online] Available: http://www.analytictech.com/networks/centaids.htm

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).centrality_closeness()
{0: 0.6111111111111116, 1: 0.6111111111111116, 2: 0.6111111111111116, 3: 0.611111111111
sage: D = DiGraph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: D.show(figsize=[2,2])
sage: D = D.to_undirected()
sage: D.show(figsize=[2,2])
sage: D.centrality_closeness()
{0: 1.0, 1: 1.0, 2: 0.75, 3: 0.75}
sage: D.centrality_closeness(v=1)
1.0
```

centrality_degree(*v=None*)

Returns the degree centrality (fraction of vertices connected to) as a dictionary of values keyed by vertex. The degree centrality is normalized to be in range (0,1).

Measures of the centrality of a vertex within a graph determine the relative importance of that vertex to its graph. Degree centrality measures the number of links incident upon a vertex.

INPUT:

- v - a vertex label (to find degree centrality of only one vertex)

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).centrality_degree()
{0: 0.36363636363636365, 1: 0.36363636363636365, 2: 0.36363636363636365, 3: 0.3636363636363
sage: D = DiGraph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: D.show(figsize=[2,2])
sage: D = D.to_undirected()
sage: D.show(figsize=[2,2])
sage: D.centrality_degree()
{0: 1.0, 1: 1.0, 2: 0.6666666666666663, 3: 0.6666666666666663}
sage: D.centrality_degree(v=1)
1.0
```

chromatic_number()

Returns the minimal number of colors needed to color the vertices of the graph G.

EXAMPLES:

```
sage: G = Graph({0:[1,2,3],1:[2]})
sage: G.chromatic_number()
3
```

chromatic_polynomial()

Returns the chromatic polynomial of the graph G.

EXAMPLES:

```
sage: G = Graph({0:[1,2,3],1:[2]})
sage: factor(G.chromatic_polynomial())
(x - 2) * x * (x - 1)^2

sage: g = graphs.trees(5).next()
sage: g.chromatic_polynomial().factor()
x * (x - 1)^4

sage: seven_acre_wood = sum(graphs.trees(7), Graph())
sage: seven_acre_wood.chromatic_polynomial()
x^77 - 66*x^76 ... - 2515943049305400*x^60 ... - 66*x^12 + x^11

sage: for i in range(2,7):
...       graphs.CompleteGraph(i).chromatic_polynomial().factor()
(x - 1) * x
(x - 2) * (x - 1) * x
(x - 3) * (x - 2) * (x - 1) * x
(x - 4) * (x - 3) * (x - 2) * (x - 1) * x
(x - 5) * (x - 4) * (x - 3) * (x - 2) * (x - 1) * x
```

clique_number(*cliques=None*)

Returns the size of the largest clique of the graph (clique number).

Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

INPUT:

- `cliques` - list of cliques (if already computed)

EXAMPLES:

```
sage: C = Graph('DJ{')
sage: C.clique_number()
4
sage: E = C.cliques()
sage: E
[[4, 1, 2, 3], [4, 0]]
sage: C.clique_number(cliques=E)
4
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.clique_number()
3
```

cliques()

Returns the list of maximal cliques. Each maximal clique is represented by a list of vertices.

Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

Maximal cliques are the largest complete subgraphs containing a given point. This function is based on Networkx's implementation of the Bron and Kerbosch Algorithm, [1].

REFERENCE:

- [1] Coen Bron and Joep Kerbosch. (1973). Algorithm 457: Finding All Cliques of an Undirected Graph. Commun. ACM. v 16. n 9. pages 575-577. ACM Press. [Online] Available: http://www.ram.org/computing/rambin/rambin.html

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).cliques()
[[0, 1], [0, 4], [0, 6], [0, 9], [2, 1], [2, 3], [2, 6], [2, 8], [3, 4], [3, 7], [3, 9], [5,
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques()
[[0, 1, 2], [0, 1, 3]]
```

cliques_containing_vertex(*vertices=None, cliques=None, with_labels=False*)

Returns the cliques containing each vertex, represented as a list of lists. (Returns a single list if only one input vertex).

Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

INPUT:

- `vertices` - the vertices to inspect (default is entire graph)
- `with_labels` - (boolean) default False returns list as above True returns a dictionary keyed by vertex labels
- `cliques` - list of cliques (if already computed)

EXAMPLES:

```
sage: C = Graph('DJ{')
sage: C.cliques_containing_vertex()
[[[4, 0]], [[4, 1, 2, 3]], [[4, 1, 2, 3]], [[4, 1, 2, 3]], [[4, 1, 2, 3], [4, 0]]]
sage: E = C.cliques()
sage: E
[[4, 1, 2, 3], [4, 0]]
sage: C.cliques_containing_vertex(cliques=E)
[[[4, 0]], [[4, 1, 2, 3]], [[4, 1, 2, 3]], [[4, 1, 2, 3]], [[4, 1, 2, 3], [4, 0]]]
sage: F = graphs.Grid2dGraph(2,3)
sage: F.cliques_containing_vertex(with_labels=True)
{(0, 1): [[(0, 1), (0, 0)], [(0, 1), (0, 2)], [(0, 1), (1, 1)]], (1, 2): [[(1, 2), (0, 2)],
sage: F.cliques_containing_vertex(vertices=[(0, 1), (1, 2)])
[[[(0, 1), (0, 0)], [(0, 1), (0, 2)], [(0, 1), (1, 1)]], [[(1, 2), (0, 2)], [(1, 2), (1, 1)]
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_containing_vertex()
[[[0, 1, 2], [0, 1, 3]], [[0, 1, 2], [0, 1, 3]], [[0, 1, 2]], [[0, 1, 3]]]
```

cliques_get_clique_bipartite(***kwds*)

Returns a bipartite graph constructed such that cliques are the right vertices and the left vertices are retained from the given graph. Right and left vertices are connected if the bottom vertex belongs to the clique represented by a top vertex.

Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).cliques_get_clique_bipartite()
Bipartite graph on 36 vertices
sage: ((graphs.ChvatalGraph()).cliques_get_clique_bipartite()).show(figsize=[2,2], vertex_si
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_get_clique_bipartite()
```

```
Bipartite graph on 6 vertices
sage: (G.cliques_get_clique_bipartite()).show(figsize=[2,2])
```

cliques_get_max_clique_graph(*name=''*)

Returns a graph constructed with maximal cliques as vertices, and edges between maximal cliques with common members in the original graph.

Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

INPUT:

- `name` - The name of the new graph.

EXAMPLES:

```
sage: (graphs.ChvatalGraph()).cliques_get_max_clique_graph()
Graph on 24 vertices
sage: ((graphs.ChvatalGraph()).cliques_get_max_clique_graph()).show(figsize=[2,2], vertex_si
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_get_max_clique_graph()
Graph on 2 vertices
sage: (G.cliques_get_max_clique_graph()).show(figsize=[2,2])
```

cliques_number_of(*vertices=None, cliques=None, with_labels=False*)

Returns a list of the number of maximal cliques containing each vertex. (Returns a single value if only one input vertex).

Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

INPUT:

- `vertices` - the vertices to inspect (default is entire graph)
- `with_labels` - (boolean) default False returns list as above True returns a dictionary keyed by vertex labels
- `cliques` - list of cliques (if already computed)

EXAMPLES:

```
sage: C = Graph('DJ{')
sage: C.cliques_number_of()
[1, 1, 1, 1, 2]
sage: E = C.cliques()
sage: E
[[4, 1, 2, 3], [4, 0]]
sage: C.cliques_number_of(cliques=E)
[1, 1, 1, 1, 2]
sage: F = graphs.Grid2dGraph(2,3)
sage: F.cliques_number_of(with_labels=True)
{(0, 1): 3, (1, 2): 2, (0, 0): 2, (1, 1): 3, (1, 0): 2, (0, 2): 2}
sage: F.cliques_number_of(vertices=[(0, 1), (1, 2)])
[3, 2]
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_number_of()
[2, 2, 1, 1]
```

cliques_vertex_clique_number(*vertices=None, with_labels=False, cliques=None*)

Returns a list of sizes of the largest maximal cliques containing each vertex. (Returns a single value if only one input vertex).

Currently only implemented for undirected graphs. Use to_undirected to convert a digraph to an undirected graph.

INPUT:

- `vertices` - the vertices to inspect (default is entire graph)
- `with_labels` - (boolean) default False returns list as above True returns a dictionary keyed by vertex labels
- `cliques` - list of cliques (if already computed)

EXAMPLES:

```
sage: C = Graph('DJ{')
sage: C.cliques_vertex_clique_number()
[2, 4, 4, 4, 4]
sage: E = C.cliques()
sage: E
[[4, 1, 2, 3], [4, 0]]
sage: C.cliques_vertex_clique_number(cliques=E)
[2, 4, 4, 4, 4]
sage: F = graphs.Grid2dGraph(2,3)
sage: F.cliques_vertex_clique_number(with_labels=True)
{(0, 1): 2, (1, 2): 2, (0, 0): 2, (1, 1): 2, (1, 0): 2, (0, 2): 2}
sage: F.cliques_vertex_clique_number(vertices=[(0, 1), (1, 2)])
[2, 2]
sage: G = Graph({0:[1,2,3], 1:[2], 3:[0,1]})
sage: G.show(figsize=[2,2])
sage: G.cliques_vertex_clique_number()
[3, 3, 3, 3]
```

coloring(*hex_colors=False*)

Returns the first (optimal) coloring found.

INPUT:

```
hex_colors -- if True, return a dict which can
              easily be used for plotting
```

EXAMPLES:

```
sage: G = Graph("Fooba")

sage: P = G.coloring(); P
[[1, 2, 3], [0, 5, 6], [4]]
sage: G.plot(partition=P)

sage: H = G.coloring(hex_colors=True)
sage: for c in sorted(H.keys()):
...    print c, H[c]
#0000ff [4]
#00ff00 [1, 2, 3]
#ff0000 [0, 5, 6]
sage: G.plot(vertex_colors=H)
```

eulerian_circuit(*return_vertices=False, labels=True*)

Return a list of edges forming an eulerian circuit if one exists. Otherwise return False.

This is implemented using Fleury's algorithm. This could be extended to find eulerian paths too (check for existence and make sure you start on an odd-degree vertex if one exists).

INPUT:

- `return_vertices` - optionally provide a list of vertices for the path
- `labels` - whether to return edges with labels (3-tuples)

OUTPUT: either ([edges], [vertices]) or [edges] of an Eulerian circuit

EXAMPLES:

```
sage: g=graphs.CycleGraph(5);
sage: g.eulerian_circuit()
[(0, 1, None), (1, 2, None), (2, 3, None), (3, 4, None), (4, 0, None)]
sage: g.eulerian_circuit(labels=False)
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
sage: g = graphs.CompleteGraph(7)
sage: edges, vertices = g.eulerian_circuit(return_vertices=True)
sage: vertices
[0, 1, 2, 0, 3, 1, 4, 0, 5, 1, 6, 2, 3, 4, 2, 5, 3, 6, 4, 5, 6, 0]
sage: graphs.CompleteGraph(4).eulerian_circuit()
False
```

graph6_string()

Returns the graph6 representation of the graph as an ASCII string. Only valid for simple (no loops, multiple edges) graphs on 0 to 262143 vertices.

EXAMPLE:

```
sage: G = graphs.KrackhardtKiteGraph()
sage: G.graph6_string()
'IvUqwK@?G'
```

graphviz_string()

Returns a representation in the DOT language, ready to render in graphviz.

REFERENCES:

- http://www.graphviz.org/doc/info/lang.html

EXAMPLE:

```
sage: G = Graph({0:{1:None,2:None}, 1:{0:None,2:None}, 2:{0:None,1:None,3:'foo'}, 3:{2:'foo'
sage: s = G.graphviz_string()
sage: s
'graph {\n"0";"1";"2";"3";\n"0"--"1";"0"--"2";"1"--"2";"2"--"3"[label="foo"];\n}'
```

is_bipartite()

Returns True if graph G is bipartite, False if not.

Traverse the graph G with depth-first-search and color nodes. This function uses the corresponding NetworkX function.

EXAMPLE:

```
sage: graphs.CycleGraph(4).is_bipartite()
True
sage: graphs.CycleGraph(5).is_bipartite()
False
```

is_directed()

Since graph is undirected, returns False.

EXAMPLE:

```
sage: Graph().is_directed()
False
```

min_spanning_tree(*weight_function=<function <lambda> at 0x2666d70>, algorithm='Kruskal', starting_vertex=None*)

Returns the edges of a minimum spanning tree, if one exists, otherwise returns False.

INPUT:

- `weight_function` - A function that takes an edge and returns a numeric weight. Defaults to assigning each edge a weight of 1.

- `algorithm` - Three variants of algorithms are implemented: 'Kruskal', 'Prim fringe', and 'Prim edge' (the last two are variants of Prim's algorithm). Defaults to 'Kruskal'. Currently, 'Prim fringe' ignores the labels on the edges.
- `starting_vertex` - The vertex with which to start Prim's algorithm.

OUTPUT: the edges of a minimum spanning tree.

EXAMPLES:

```
sage: g=graphs.CompleteGraph(5)
sage: len(g.min_spanning_tree())
4
sage: weight = lambda e: 1/( (e[0]+1)*(e[1]+1) )
sage: g.min_spanning_tree(weight_function=weight)
[(3, 4, None), (2, 4, None), (1, 4, None), (0, 4, None)]
sage: g.min_spanning_tree(algorithm='Prim edge', starting_vertex=2, weight_function=weight)
[(2, 4, None), (3, 4, None), (1, 3, None), (0, 4, None)]
sage: g.min_spanning_tree(algorithm='Prim fringe', starting_vertex=2, weight_function=weight
[(4, 2), (3, 4), (1, 4), (0, 4)]
```

sparse6_string()

Returns the sparse6 representation of the graph as an ASCII string. Only valid for undirected graphs on 0 to 262143 vertices, but loops and multiple edges are permitted.

EXAMPLE:

```
sage: G = graphs.BullGraph()
sage: G.sparse6_string()
':Da@en'

sage: G = Graph()
sage: G.sparse6_string()
':?'

sage: G = Graph(loops=True, multiedges=True)
sage: Graph(':?') == G
True
```

to_directed(*implementation='networkx'*)

Returns a directed version of the graph. A single edge becomes two edges, one in each direction.

EXAMPLE:

```
sage: graphs.PetersenGraph().to_directed()
Petersen graph: Digraph on 10 vertices
```

to_undirected()

Since the graph is already undirected, simply returns a copy of itself.

EXAMPLE:

```
sage: graphs.PetersenGraph().to_undirected()
Petersen graph: Graph on 10 vertices
```

write_to_eps (*filename, iterations=50*)

Writes a plot of the graph to filename in eps format.

It is relatively simple to include this file in a latex document:

INPUT: filename

- `iterations` - how many iterations of the spring layout algorithm to go through, if applicable

`usepackagegraphics` must appear before the beginning of the document, and `includegraphics filename.eps` will include it in your latex doc. Note: you cannot use pdflatex to print the resulting document, use TeX and Ghostscript or something similar instead.

EXAMPLE:

```
sage: P = graphs.PetersenGraph()
sage: P.write_to_eps(tmp_dir() + 'sage.eps')
```

compare_edges(*x, y*)

Compare edge x to edge y, return -1 if x y, 1 if x y, else 0.

EXAMPLE:

```
sage: G = graphs.PetersenGraph()
sage: E = G.edges()
sage: from sage.graphs.graph import compare_edges
sage: compare_edges(E[0], E[2])
-1
sage: compare_edges(E[0], E[1])
-1
sage: compare_edges(E[0], E[0])
0
sage: compare_edges(E[1], E[0])
1
```

graph_isom_equivalent_non_edge_labeled_graph(*g, partition*)

Helper function for canonical labeling of edge labeled (di)graphs.

Translates to a bipartite incidence-structure type graph appropriate for computing canonical labels of edge labeled graphs. Note that this is actually computationally equivalent to implementing a change on an inner loop of the main algorithm- namely making the refinement procedure sort for each label.

If the graph is a multigraph, it is translated to a non-multigraph, where each edge is labeled with a dictionary describing how many edges of each label were originally there. Then in either case we are working on a graph without multiple edges. At this point, we create another (bipartite) graph, whose left vertices are the original vertices of the graph, and whose right vertices represent the edges. We partition the left vertices as they were originally, and the right vertices by common labels: only automorphisms taking edges to like-labeled edges are allowed, and this additional partition information enforces this on the bipartite graph.

EXAMPLE:

```
sage: G = Graph(multiedges=True, implementation='networkx')
sage: G.add_edges([(0,1,i) for i in range(10)])
sage: G.add_edge(1,2,'string')
sage: G.add_edge(2,3)
sage: from sage.graphs.graph import graph_isom_equivalent_non_edge_labeled_graph
sage: graph_isom_equivalent_non_edge_labeled_graph(G, [G.vertices()])
(Graph on 7 vertices, [[('o', 0), ('o', 1), ('o', 2), ('o', 3)], [('x', 2)], [('x', 0)], [('x',
```

graph_isom_equivalent_non_multi_graph(*g, partition*)

Helper function for canonical labeling of multi-(di)graphs.

The idea for this function is that the main algorithm for computing isomorphism of graphs does not allow multiple edges. Instead of making some very difficult changes to that, we can simply modify the multigraph into a non-multi graph that carries essentially the same information. For each pair of vertices $\{u, v\}$, if there is at most one edge between u and v, we do nothing, but if there are more than one, we split each edge into two, introducing a new vertex. These vertices really represent edges, so we keep them in their own part of a partition - to distinguish them from genuine vertices. Then the canonical label and automorphism group is computed, and in the end, we strip off the parts of the generators that describe how these new vertices move, and we have the automorphism group of the original multi-graph. Similarly, by putting the additional vertices in their own cell of the partition, we guarantee that the relabeling leading to a canonical label moves genuine vertices amongst themselves, and hence the canonical label is still well-defined, when we forget about the additional vertices.

EXAMPLE:

```
sage: from sage.graphs.graph import graph_isom_equivalent_non_multi_graph
sage: G = Graph(multiedges=True)
sage: G.add_edge((0,1,1))
sage: G.add_edge((0,1,2))
sage: G.add_edge((0,1,3))
sage: graph_isom_equivalent_non_multi_graph(G, [[0,1]])
(Graph on 5 vertices, [[('o', 0), ('o', 1)], [('x', 0), ('x', 1), ('x', 2)]])
```

tachyon_vertex_plot(*g, bgcolor=(1, 1, 1), vertex_colors=None, vertex_size=0.059999999999999998, pos3d=None, iterations=50, **kwds*)

Helper function for plotting graphs in 3d with Tachyon. Returns a plot containing only the vertices, as well as the 3d position dictionary used for the plot.

EXAMPLE:

```
sage: G = graphs.TetrahedralGraph()
sage: from sage.graphs.graph import tachyon_vertex_plot
sage: T,p = tachyon_vertex_plot(G)
sage: type(T)
<class 'sage.plot.tachyon.Tachyon'>
sage: type(p)
<type 'dict'>
```

7.2 A collection of constructors of common graphs.

USE:

To see a list of all graph constructors, type "graphs." and then press the tab key. The documentation for each constructor includes information about each graph, which provides a useful reference.

PLOTTING:

All graphs (i.e., networks) have an associated Sage graphics object, which you can display:

```
sage: G = graphs.WheelGraph(15)
sage: P = G.plot()
sage: P.show() # long time
```

If you create a graph in Sage using the `Graph` command, then plot that graph, the positioning of nodes is determined using the spring-layout algorithm. For the special graph constructors, which you get using `graphs.[tab]`, the positions are preset. For example, consider the Petersen graph with default node positioning vs. the Petersen graph constructed by this database:

```
sage: petersen_spring = Graph({0:[1,4,5], 1:[0,2,6], 2:[1,3,7], 3:[2,4,8], 4:[0,3,9], 5:[0,7,8], 6:[1
sage: petersen_spring.show() # long time
sage: petersen_database = graphs.PetersenGraph()
sage: petersen_database.show() # long time
```

For all the constructors in this database (except the octahedral, dodecahedral, random and empty graphs), the position dictionary is filled in, instead of using the spring-layout algorithm.

For further visual examples and explanation, see the docstrings below, particularly for CycleGraph, StarGraph, WheelGraph, CompleteGraph and CompleteBipartiteGraph.

ORGANIZATION:

The constructors available in this database are organized as follows:

```
Basic Structures:
    - BarbellGraph
    - BullGraph
    - CircularLadderGraph
    - ClawGraph
    - CycleGraph
    - DiamondGraph
    - EmptyGraph
    - Grid2dGraph
    - GridGraph
    - HouseGraph
    - HouseXGraph
    - KrackhardtKiteGraph
    - LadderGraph
    - LollipopGraph
    - PathGraph
    - StarGraph
    - WheelGraph
Platonic Solids:
    - TetrahedralGraph
    - HexahedralGraph
    - OctahedralGraph
    - IcosahedralGraph
    - DodecahedralGraph
Named Graphs:
    - ChvatalGraph
    - DesarguesGraph
    - FlowerSnark
    - FruchtGraph
    - HeawoodGraph
    - HoffmanSingletonGraph
    - MoebiusKantorGraph
    - Pappus Graph
    - PetersenGraph
    - ThomsenGraph
Families of Graphs:
    - CirculantGraph
    - CompleteGraph
    - CompleteBipartiteGraph
    - CubeGraph
    - BalancedTree
    - LCFGraph
Pseudofractal Graphs:
    - DorogovtsevGoltsevMendesGraph
Random Graphs:
    - RandomGNP
    - RandomBarabasiAlbert
    - RandomGNM
    - RandomNewmanWattsStrogatz
    - RandomHolmeKim
    - RandomLobster
    - RandomTreePowerlaw
    - RandomRegular
    - RandomShell
Random Directed Graphs:
    - RandomDirectedGN
    - RandomDirectedGNC
    - RandomDirectedGNR
```

```
Graphs with a given degree sequence:
    - DegreeSequence
    - DegreeSequenceConfigurationModel
    - DegreeSequenceTree
    - DegreeSequenceExpected
```

AUTHORS:

- Robert Miller (2006-11-05): initial version, empty, random, petersen
- Emily Kirkman (2006-11-12): basic structures, node positioning for all constructors
- Emily Kirkman (2006-11-19): docstrings, examples
- William Stein (2006-12-05): Editing.
- Robert Miller (2007-01-16): Cube generation and plotting
- Emily Kirkman (2007-01-16): more basic structures, docstrings
- Emily Kirkman (2007-02-14): added more named graphs
- Robert Miller (2007-06-08-11): Platonic solids, random graphs, graphs with a given degree sequence, random directed graphs
- Robert Miller (2007-10-24): Isomorph free exhaustive generation

class `DiGraphGenerators`()

A class consisting of constructors for several common digraphs, including orderly generation of isomorphism class representatives.

A list of all graphs and graph structures in this database is available via tab completion. Type "digraphs." and then hit tab to see which graphs are available.

The docstrings include educational information about each named digraph with the hopes that this class can be used as a reference.

The constructors currently in this class include:

```
Random Directed Graphs:
    - RandomDirectedGN
    - RandomDirectedGNC
    - RandomDirectedGNR
```

ORDERLY GENERATION: digraphs(vertices, property=lambda x: True, augment='edges', size=None)

Accesses the generator of isomorphism class representatives. Iterates over distinct, exhaustive representatives.

INPUT:

- `vertices` - natural number
- `property` - any property to be tested on digraphs before generation.
- `augment` - choices:
- `'vertices'` - augments by adding a vertex, and edges incident to that vertex. In this case, all digraphs on up to n=vertices are generated. If for any digraph G satisfying the property, every subgraph, obtained from G by deleting one vertex and only edges incident to that vertex, satisfies the property, then this will generate all digraphs with that property. If this does not hold, then all the digraphs generated will satisfy the property, but there will be some missing.

- 'edges' - augments a fixed number of vertices by adding one edge In this case, all digraphs on exactly n=vertices are generated. If for any graph G satisfying the property, every subgraph, obtained from G by deleting one edge but not the vertices incident to that edge, satisfies the property, then this will generate all digraphs with that property. If this does not hold, then all the digraphs generated will satisfy the property, but there will be some missing.
- implementation - which underlying implementation to use (see DiGraph?)
- sparse - ignored if implementation is not c_graph

EXAMPLES: Print digraphs on 2 or less vertices.

```
sage: for D in digraphs(2, augment='vertices'):
...     print D
...
Digraph on 0 vertices
Digraph on 1 vertex
Digraph on 2 vertices
Digraph on 2 vertices
Digraph on 2 vertices
```

Note that we can also get digraphs with underlying Cython implementation:

```
sage: for D in digraphs(2, augment='vertices', implementation='c_graph'):
...     print D
...
Digraph on 0 vertices
Digraph on 1 vertex
Digraph on 2 vertices
Digraph on 2 vertices
Digraph on 2 vertices
```

Print digraphs on 3 vertices.

```
sage: for D in digraphs(3):
...     print D
Digraph on 3 vertices
Digraph on 3 vertices
...
Digraph on 3 vertices
Digraph on 3 vertices
```

Generate all digraphs with 4 vertices and 3 edges.

```
sage: L = digraphs(4, size=3)
sage: len(list(L))
13
```

Generate all digraphs with 4 vertices and up to 3 edges.

```
sage: L = list(digraphs(4, lambda G: G.size() <= 3))
sage: len(L)
20
sage: graphs_list.show_graphs(L)  # long time
```

Generate all digraphs with degree at most 2, up to 5 vertices.

```
sage: property = lambda G: ( max([G.degree(v) for v in G] + [0]) <= 2 )
sage: L = list(digraphs(5, property, augment='vertices'))
sage: len(L)
75
```

Generate digraphs on the fly: (see http://www.research.att.com/~njas/sequences/A000273)

```
sage: for i in range(0, 5):
...       print len(list(digraphs(i)))
1
1
3
16
218
```

REFERENCE:

•Brendan D. McKay, Isomorph-Free Exhaustive generation. Journal of Algorithms Volume 26, Issue 2, February 1998, pages 306-324.

ButterflyGraph (*n, vertices='strings'*)

Returns a n-dimensional butterfly graph. The vertices consist of pairs (v,i), where v is an n-dimensional tuple (vector) with binary entries (or a string representation of such) and i is an integer in [0..n]. A directed edge goes from (v,i) to (w,i+1) if v and w are identical except for possibly v[i] != w[i].

A butterfly graph has $(2^n)(n+1)$ vertices and $n2^{n+1}$ edges.

INPUT:

•`vertices` - 'strings' (default) or 'vectors', specifying whether the vertices are zero-one strings or actually tuples over GF(2).

EXAMPLES:

```
sage: digraphs.ButterflyGraph(2).edges(labels=False)
[(('00', 0), ('00', 1)),
(('00', 0), ('10', 1)),
(('00', 1), ('00', 2)),
(('00', 1), ('01', 2)),
(('01', 0), ('01', 1)),
(('01', 0), ('11', 1)),
(('01', 1), ('00', 2)),
(('01', 1), ('01', 2)),
(('10', 0), ('00', 1)),
(('10', 0), ('10', 1)),
(('10', 1), ('10', 2)),
(('10', 1), ('11', 2)),
(('11', 0), ('01', 1)),
(('11', 0), ('11', 1)),
(('11', 1), ('10', 2)),
(('11', 1), ('11', 2))]
sage: digraphs.ButterflyGraph(2,vertices='vectors').edges(labels=False)
[(((0, 0), 0), ((0, 0), 1)),
(((0, 0), 0), ((1, 0), 1)),
(((0, 0), 1), ((0, 0), 2)),
(((0, 0), 1), ((0, 1), 2)),
(((0, 1), 0), ((0, 1), 1)),
(((0, 1), 0), ((1, 1), 1)),
(((0, 1), 1), ((0, 0), 2)),
(((0, 1), 1), ((0, 1), 2)),
(((1, 0), 0), ((0, 0), 1)),
```

```
(((1, 0), 0), ((1, 0), 1)),
(((1, 0), 1), ((1, 0), 2)),
(((1, 0), 1), ((1, 1), 2)),
(((1, 1), 0), ((0, 1), 1)),
(((1, 1), 0), ((1, 1), 1)),
(((1, 1), 1), ((1, 0), 2)),
(((1, 1), 1), ((1, 1), 2))]
```

RandomDirectedGN(*n, kernel=<function <lambda> at 0x266b398>, seed=None*)

Returns a random GN (growing network) digraph with n vertices.

The digraph is constructed by adding vertices with a link to one previously added vertex. The vertex to link to is chosen with a preferential attachment model, i.e. probability is proportional to degree. The default attachment kernel is a linear function of degree. The digraph is always a tree, so in particular it is a directed acyclic graph.

INPUT:

- `n` - number of vertices.
- `kernel` - the attachment kernel
- `seed` - for the random number generator

EXAMPLE:

```
sage: D = digraphs.RandomDirectedGN(25)
sage: D.edges(labels=False)
[(1, 0), (2, 0), (3, 1), (4, 0), (5, 0), (6, 1), (7, 0), (8, 3), (9, 0), (10, 8), (11, 3), (
sage: D.show()  # long time
```

REFERENCE:

- [1] Krapivsky, P.L. and Redner, S. Organization of Growing Random Networks, Phys. Rev. E vol. 63 (2001), p. 066123.

RandomDirectedGNC(*n, seed=None*)

Returns a random GNC (growing network with copying) digraph with n vertices.

The digraph is constructed by adding vertices with a link to one previously added vertex. The vertex to link to is chosen with a preferential attachment model, i.e. probability is proportional to degree. The new vertex is also linked to all of the previously added vertex's successors.

INPUT:

- `n` - number of vertices.
- `seed` - for the random number generator

EXAMPLE:

```
sage: D = digraphs.RandomDirectedGNC(25)
sage: D.edges(labels=False)
[(1, 0), (2, 0), (2, 1), (3, 0), (4, 0), (4, 1), (5, 0), (5, 1), (5, 2), (6, 0), (6, 1), (7,
sage: D.show()  # long time
```

REFERENCE:

- [1] Krapivsky, P.L. and Redner, S. Network Growth by Copying, Phys. Rev. E vol. 71 (2005), p. 036118.

RandomDirectedGNP(*n, p*)

Returns a random digraph on n nodes. Each edge is inserted independently with probability p.

REFERENCES:

- [1] P. Erdos and A. Renyi, On Random Graphs, Publ. Math. 6, 290 (1959).
- [2] E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

PLOTTING: When plotting, this graph will use the default spring-layout algorithm, unless a position dictionary is specified.

EXAMPLE:

```
sage: digraphs.RandomDirectedGNP(10, .2).num_verts()
10
```

RandomDirectedGNR(*n, p, seed=None*)

Returns a random GNR (growing network with redirection) digraph with n vertices and redirection probability p.

The digraph is constructed by adding vertices with a link to one previously added vertex. The vertex to link to is chosen uniformly. With probability p, the arc is instead redirected to the successor vertex. The digraph is always a tree.

INPUT:

- `n` - number of vertices.
- `p` - redirection probability
- `seed` - for the random number generator.

EXAMPLE:

```
sage: D = digraphs.RandomDirectedGNR(25, .2)
sage: D.edges(labels=False)
[(1, 0), (2, 0), (2, 1), (3, 0), (4, 0), (4, 1), (5, 0), (5, 1), (5, 2), (6, 0), (6, 1), (7,
sage: D.show()  # long time
```

REFERENCE:

- [1] Krapivsky, P.L. and Redner, S. Organization of Growing Random Networks, Phys. Rev. E vol. 63 (2001), p. 066123.

class GraphGenerators()

A class consisting of constructors for several common graphs, as well as orderly generation of isomorphism class representatives.

A list of all graphs and graph structures (other than iso. class rep's) in this database is available via tab completion. Type "graphs." and then hit tab to see which graphs are available.

The docstrings include educational information about each named graph with the hopes that this class can be used as a reference.

For all the constructors in this class (except the octahedral, dodecahedral, random and empty graphs), the position dictionary is filled to override the spring-layout algorithm.

The constructors currently in this class include:

```
Basic Structures:
    - BarbellGraph
    - BullGraph
    - CircularLadderGraph
    - ClawGraph
    - CycleGraph
    - DiamondGraph
    - EmptyGraph
    - Grid2dGraph
    - GridGraph
    - HouseGraph
    - HouseXGraph
    - KrackhardtKiteGraph
    - LadderGraph
    - LollipopGraph
    - PathGraph
```

```
    - StarGraph
    - WheelGraph
Platonic Solids:
    - TetrahedralGraph
    - HexahedralGraph
    - OctahedralGraph
    - IcosahedralGraph
    - DodecahedralGraph
Named Graphs:
    - ChvatalGraph
    - DesarguesGraph
    - FlowerSnark
    - FruchtGraph
    - HeawoodGraph
    - HoffmanSingletonGraph
    - MoebiusKantorGraph
    - PappusGraph
    - PetersenGraph
    - ThomsenGraph
Families of Graphs:
    - CirculantGraph
    - CompleteGraph
    - CompleteBipartiteGraph
    - CubeGraph
    - BalancedTree
    - LCFGraph
Pseudofractal Graphs:
    - DorogovtsevGoltsevMendesGraph
Random Graphs:
    - RandomGNP
    - RandomBarabasiAlbert
    - RandomGNM
    - RandomNewmanWattsStrogatz
    - RandomHolmeKim
    - RandomLobster
    - RandomTreePowerlaw
    - RandomRegular
    - RandomShell
Graphs with a given degree sequence:
    - DegreeSequence
    - DegreeSequenceConfigurationModel
    - DegreeSequenceTree
    - DegreeSequenceExpected
```

ORDERLY GENERATION: graphs(vertices, property=lambda x: True, augment='edges', size=None)

This syntax accesses the generator of isomorphism class representatives. Iterates over distinct, exhaustive representatives.

INPUT:

- `vertices` - natural number
- `property` - any property to be tested on graphs before generation. (Ignored if deg_seq is specified.)
- `augment` - choices:
- `'vertices'` - augments by adding a vertex, and edges incident to that vertex. In this case, all graphs on up to n=vertices are generated. If for any graph G satisfying the property, every subgraph, obtained from G by deleting one vertex and only edges incident to that vertex, satisfies the property, then this will generate

all graphs with that property. If this does not hold, then all the graphs generated will satisfy the property, but there will be some missing.

- `'edges'` - augments a fixed number of vertices by adding one edge In this case, all graphs on exactly n=vertices are generated. If for any graph G satisfying the property, every subgraph, obtained from G by deleting one edge but not the vertices incident to that edge, satisfies the property, then this will generate all graphs with that property. If this does not hold, then all the graphs generated will satisfy the property, but there will be some missing.
- `deg_seq` - a sequence of non-negative integers, or None. If specified, the generated graphs will have these integers for degrees. In this case property and size are both ignored.
- `loops` - whether to allow loops in the graph or not.
- `implementation` - which underlying implementation to use (see Graph?)
- `sparse` - ignored if implementation is not `c_graph`

EXAMPLES: Print graphs on 3 or less vertices.

```
sage: for G in graphs(3, augment='vertices'):
...     print G
Graph on 0 vertices
Graph on 1 vertex
Graph on 2 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 2 vertices
Graph on 3 vertices
```

Note that we can also get graphs with underlying Cython implementation:

```
sage: for G in graphs(3, augment='vertices', implementation='c_graph'):
...     print G
Graph on 0 vertices
Graph on 1 vertex
Graph on 2 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 2 vertices
Graph on 3 vertices
```

Print graphs on 3 vertices.

```
sage: for G in graphs(3):
...     print G
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
```

Generate all graphs with 5 vertices and 4 edges.

```
sage: L = graphs(5, size=4)
sage: len(list(L))
6
```

Generate all graphs with 5 vertices and up to 4 edges.

```
sage: L = list(graphs(5, lambda G: G.size() <= 4))
sage: len(L)
14
sage: graphs_list.show_graphs(L) # long time
```

Generate all graphs with up to 5 vertices and up to 4 edges.

```
sage: L = list(graphs(5, lambda G: G.size() <= 4, augment='vertices'))
sage: len(L)
31
sage: graphs_list.show_graphs(L)               # long time
```

Generate all graphs with degree at most 2, up to 6 vertices.

```
sage: property = lambda G: ( max([G.degree(v) for v in G] + [0]) <= 2 )
sage: L = list(graphs(6, property, augment='vertices'))
sage: len(L)
45
```

Generate all bipartite graphs on up to 7 vertices: (see http://www.research.att.com/~njas/sequences/A033995)

```
sage: L = list( graphs(7, lambda G: G.is_bipartite(), augment='vertices') )
sage: [len([g for g in L if g.order() == i]) for i in [1..7]]
[1, 2, 3, 7, 13, 35, 88]
```

Generate all bipartite graphs on exactly 7 vertices:

```
sage: L = list( graphs(7, lambda G: G.is_bipartite()) )
sage: len(L)
88
```

Generate all bipartite graphs on exactly 8 vertices:

```
sage: L = list( graphs(8, lambda G: G.is_bipartite()) ) # long time
sage: len(L)                                            # long time
303
```

Generate graphs on the fly: (see http://www.research.att.com/~njas/sequences/A000088)

```
sage: for i in range(0, 7):
...     print len(list(graphs(i)))
1
1
2
4
11
34
156
```

Generate all simple graphs, allowing loops: (see http://www.research.att.com/~njas/sequences/A000666)

```
sage: L = list(graphs(6,augment='vertices',loops=True))               # long time
sage: for i in [0..6]: print i, len([g for g in L if g.order() == i]) # long time
0 1
1 2
2 6
3 20
4 90
```

```
5 544
6 5096
```

Generate all graphs with a specified degree sequence: (see http://www.research.att.com/~njas/sequences/A002851)

```
sage: for i in [4,6,8]:
...     print i, len([g for g in graphs(i,deg_seq=[3]*i) if g.is_connected()])
4 1
6 2
8 5
sage: for i in [4,6,8]:
...     print i, len([g for g in graphs(i,augment='vertices',deg_seq=[3]*i) if g.is_connected()])
4 1
6 2
8 5


sage: print 10, len([g for g in graphs(10,deg_seq=[3]*10) if g.is_connected()]) # not tested
10 19
```

REFERENCE:

- Brendan D. McKay, Isomorph-Free Exhaustive generation. Journal of Algorithms Volume 26, Issue 2, February 1998, pages 306-324.

BalancedTree(*r, h*)

Returns the perfectly balanced tree of height $h \geq 1$, whose root has degree $r \geq 2$.

The number of vertices of this graph is $1 + r + r^2 + \cdots + r^h$, that is, $\frac{r^{h+1}-1}{r-1}$. The number of edges is one less than the number of vertices.

EXAMPLE: Plot a balanced tree of height 4 with r = 3

```
sage: G = graphs.BalancedTree(3, 5)
sage: G.show()   # long time
```

BarbellGraph(*n1, n2*)

Returns a barbell graph with 2*n1 + n2 nodes. n1 must be greater than or equal to 2.

A barbell graph is a basic structure that consists of a path graph of order n2 connecting two complete graphs of order n1 each.

This constructor depends on NetworkX numeric labels. In this case, the (n1)th node connects to the path graph from one complete graph and the (n1+n2+1)th node connects to the path graph from the other complete graph.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each barbell graph will be displayed with the two complete graphs in the lower-left and upper-right corners, with the path graph connecting diagonally between the two. Thus the (n1)th node will be drawn at a 45 degree angle from the horizontal right center of the first complete graph, and the (n1+n2+1)th node will be drawn 45 degrees below the left horizontal center of the second complete graph.

EXAMPLES: Construct and show a barbell graph Bar = 4, Bells = 9

```
sage: g = graphs.BarbellGraph(9,4)
sage: g.show() # long time
```

Create several barbell graphs in a Sage graphics array

```
sage: g = []
sage: j = []
sage: for i in range(6):
...     k = graphs.BarbellGraph(i+2,4)
...     g.append(k)
```

```
...
sage: for i in range(2):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

BullGraph()

Returns a bull graph with 5 nodes.

A bull graph is named for its shape. It's a triangle with horns.

This constructor depends on NetworkX numeric labeling.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the bull graph is drawn as a triangle with the first node (0) on the bottom. The second and third nodes (1 and 2) complete the triangle. Node 3 is the horn connected to 1 and node 4 is the horn connected to node 2.

EXAMPLES: Construct and show a bull graph

```
sage: g = graphs.BullGraph()
sage: g.show() # long time
```

ChvatalGraph()

Returns the Chvatal graph.

The Chvatal graph has 12 vertices. It is a 4-regular, 4-chromatic graph. It is one of the few known graphs to satisfy Grunbaum's conjecture that for every m 1, n 2, there is an m-regular, m-chromatic graph of girth at least n.

EXAMPLE:

```
sage: G = graphs.ChvatalGraph()
sage: G.degree()
[4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4]
```

CirculantGraph(*n, adjacency*)

Returns a circulant graph with n nodes.

A circulant graph has the property that the vertex i is connected with the vertices i+j and i-j for each j in adj.

INPUT:

- `n` - number of vertices in the graph
- `adjacency` - the list of j values

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each circulant graph will be displayed with the first (0) node at the top, with the rest following in a counterclockwise manner.

Filling the position dictionary in advance adds O(n) to the constructor.

EXAMPLES: Compare plotting using the predefined layout and networkx:

```
sage: import networkx
sage: n = networkx.cycle_graph(23)
sage: spring23 = Graph(n)
sage: posdict23 = graphs.CirculantGraph(23,2)
sage: spring23.show() # long time
sage: posdict23.show() # long time
```

We next view many cycle graphs as a Sage graphics array. First we use the `CirculantGraph` constructor, which fills in the position dictionary:

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.CirculantGraph(i+3,i)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

Compare to plotting with the spring-layout algorithm:

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     spr = networkx.cycle_graph(i+3)
...     k = Graph(spr)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

Passing a 1 into adjacency should give the cycle.

```
sage: graphs.CirculantGraph(6,1)==graphs.CycleGraph(6)
True
sage: graphs.CirculantGraph(7,[1,3]).edges(labels=false)
[(0, 1),
(0, 3),
(0, 4),
(0, 6),
(1, 2),
(1, 4),
(1, 5),
(2, 3),
(2, 5),
(2, 6),
(3, 4),
(3, 6),
(4, 5),
(5, 6)]
```

CircularLadderGraph(*n*)

Returns a circular ladder graph with 2*n nodes.

A Circular ladder graph is a ladder graph that is connected at the ends, i.e.: a ladder bent around so that top meets bottom. Thus it can be described as two parallel cycle graphs connected at each corresponding node pair.

This constructor depends on NetworkX numeric labels.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the circular ladder graph is displayed as an inner and outer cycle pair, with the first n nodes drawn on the inner circle. The first (0) node is drawn at the top of the inner-circle, moving clockwise after that. The outer circle is drawn with the (n+1)th node at the top, then counterclockwise as well.

EXAMPLES: Construct and show a circular ladder graph with 26 nodes

```
sage: g = graphs.CircularLadderGraph(13)
sage: g.show() # long time
```

Create several circular ladder graphs in a Sage graphics array

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.CircularLadderGraph(i+3)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

ClawGraph()

Returns a claw graph.

A claw graph is named for its shape. It is actually a complete bipartite graph with (n1, n2) = (1, 3).

PLOTTING: See CompleteBipartiteGraph.

EXAMPLES: Show a Claw graph

```
sage: (graphs.ClawGraph()).show() # long time
```

Inspect a Claw graph

```
sage: G = graphs.ClawGraph()
sage: G
Claw graph: Graph on 4 vertices
```

CompleteBipartiteGraph(*n1, n2*)

Returns a Complete Bipartite Graph sized n1+n2, with each of the nodes [0,(n1-1)] connected to each of the nodes [n1,(n2-1)] and vice versa.

A Complete Bipartite Graph is a graph with its vertices partitioned into two groups, V1 and V2. Each v in V1 is connected to every v in V2, and vice versa.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each complete bipartite graph will be displayed with the first n1 nodes on the top row (at y=1) from left to right. The remaining n2 nodes appear at y=0, also from left to right. The shorter row (partition with fewer nodes) is stretched to the same length as the longer row, unless the shorter row has 1 node; in which case it is centered. The x values in the plot are in domain [0,maxn1,n2].

In the Complete Bipartite graph, there is a visual difference in using the spring-layout algorithm vs. the position dictionary used in this constructor. The position dictionary flattens the graph and separates the partitioned nodes, making it clear which nodes an edge is connected to. The Complete Bipartite graph plotted with the spring-layout algorithm tends to center the nodes in n1 (see spring_med in examples below), thus overlapping its nodes and edges, making it typically hard to decipher.

Filling the position dictionary in advance adds O(n) to the constructor. Feel free to race the constructors below in the examples section. The much larger difference is the time added by the spring-layout algorithm

when plotting. (Also shown in the example below). The spring model is typically described as $O(n^3)$, as appears to be the case in the NetworkX source code.

EXAMPLES: Two ways of constructing the complete bipartite graph, using different layout algorithms:

```
sage: import networkx
sage: n = networkx.complete_bipartite_graph(389,157); spring_big = Graph(n)    # long time
sage: posdict_big = graphs.CompleteBipartiteGraph(389,157)                     # long time
```

Compare the plotting:

```
sage: n = networkx.complete_bipartite_graph(11,17)
sage: spring_med = Graph(n)
sage: posdict_med = graphs.CompleteBipartiteGraph(11,17)
```

Notice here how the spring-layout tends to center the nodes of n1

```
sage: spring_med.show() # long time
sage: posdict_med.show() # long time
```

View many complete bipartite graphs with a Sage Graphics Array, with this constructor (i.e., the position dictionary filled):

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.CompleteBipartiteGraph(i+1,4)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

We compare to plotting with the spring-layout algorithm:

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     spr = networkx.complete_bipartite_graph(i+1,4)
...     k = Graph(spr)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

CompleteGraph(*n*)

Returns a complete graph on n nodes.

A Complete Graph is a graph in which all nodes are connected to all other nodes.

This constructor is dependant on vertices numbered 0 through n-1 in NetworkX complete_graph()

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each complete graph will be displayed with the first (0) node at the top, with the rest

following in a counterclockwise manner.

In the complete graph, there is a big difference visually in using the spring-layout algorithm vs. the position dictionary used in this constructor. The position dictionary flattens the graph, making it clear which nodes an edge is connected to. But the complete graph offers a good example of how the spring-layout works. The edges push outward (everything is connected), causing the graph to appear as a 3-dimensional pointy ball. (See examples below).

EXAMPLES: We view many Complete graphs with a Sage Graphics Array, first with this constructor (i.e., the position dictionary filled):

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.CompleteGraph(i+3)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

We compare to plotting with the spring-layout algorithm:

```
sage: import networkx
sage: g = []
sage: j = []
sage: for i in range(9):
...     spr = networkx.complete_graph(i+3)
...     k = Graph(spr)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

Compare the constructors (results will vary)

```
sage: import networkx
sage: t = cputime()
sage: n = networkx.complete_graph(389); spring389 = Graph(n)
sage: cputime(t)            # random
0.59203700000000126
sage: t = cputime()
sage: posdict389 = graphs.CompleteGraph(389)
sage: cputime(t)            # random
0.6680419999999998
```

We compare plotting:

```
sage: import networkx
sage: n = networkx.complete_graph(23)
sage: spring23 = Graph(n)
```

```
sage: posdict23 = graphs.CompleteGraph(23)
sage: spring23.show() # long time
sage: posdict23.show() # long time
```

CubeGraph(*n*)

AUTHORS:

- Robert Miller

PLOTTING: See commented source code.

EXAMPLES: Plot several n-cubes in a Sage Graphics Array

```
sage: g = []
sage: j = []
sage: for i in range(6):
...     k = graphs.CubeGraph(i+1)
...     g.append(k)
...
sage: for i in range(2):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show(figsize=[6,4]) # long time
```

Use the plot options to display larger n-cubes

```
sage: g = graphs.CubeGraph(9)
sage: g.show(figsize=[12,12],vertex_labels=False, vertex_size=20) # long time
```

CycleGraph(*n*)

Returns a cycle graph with n nodes.

A cycle graph is a basic structure which is also typically called an n-gon.

This constructor is dependant on vertices numbered 0 through n-1 in NetworkX `cycle_graph()`

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each cycle graph will be displayed with the first (0) node at the top, with the rest following in a counterclockwise manner.

The cycle graph is a good opportunity to compare efficiency of filling a position dictionary vs. using the spring-layout algorithm for plotting. Because the cycle graph is very symmetric, the resulting plots should be similar (in cases of small n).

Filling the position dictionary in advance adds O(n) to the constructor.

EXAMPLES: Compare plotting using the predefined layout and networkx:

```
sage: import networkx
sage: n = networkx.cycle_graph(23)
sage: spring23 = Graph(n)
sage: posdict23 = graphs.CycleGraph(23)
sage: spring23.show() # long time
sage: posdict23.show() # long time
```

We next view many cycle graphs as a Sage graphics array. First we use the `CycleGraph` constructor, which fills in the position dictionary:

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.CycleGraph(i+3)
```

```
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

Compare to plotting with the spring-layout algorithm:

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     spr = networkx.cycle_graph(i+3)
...     k = Graph(spr)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

DegreeSequence(*deg_sequence*)

Returns a graph with the given degree sequence. Raises a NetworkX error if the proposed degree sequence cannot be that of a graph.

Graph returned is the one returned by the Havel-Hakimi algorithm, which constructs a simple graph by connecting vertices of highest degree to other vertices of highest degree, resorting the remaining vertices by degree and repeating the process. See Theorem 1.4 in [1].

INPUT:

- `deg_sequence` - a list of integers with each entry corresponding to the degree of a different vertex.

EXAMPLES:

```
sage: G = graphs.DegreeSequence([3,3,3,3])
sage: G.edges(labels=False)
[(0, 1), (0, 2), (0, 3), (1, 2), (1, 3), (2, 3)]
sage: G.show()  # long time

sage: G = graphs.DegreeSequence([3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3])
sage: G.show()  # long time

sage: G = graphs.DegreeSequence([4,4,4,4,4,4,4,4])
sage: G.show()  # long time

sage: G = graphs.DegreeSequence([1,2,3,4,3,4,3,2,3,2,1])
sage: G.show()  # long time
```

REFERENCE:

- [1] Chartrand, G. and Lesniak, L. Graphs and Digraphs. Chapman and Hall/CRC, 1996.

DegreeSequenceConfigurationModel(*deg_sequence, seed=None*)

Returns a random pseudograph with the given degree sequence. Raises a NetworkX error if the proposed degree sequence cannot be that of a graph with multiple edges and loops.

One requirement is that the sum of the degrees must be even, since every edge must be incident with two vertices.

INPUT:

- `deg_sequence` - a list of integers with each entry corresponding to the expected degree of a different vertex.
- `seed` - for the random number generator.

EXAMPLES:

```
sage: G = graphs.DegreeSequenceConfigurationModel([1,1])
sage: G.adjacency_matrix()
[0 1]
[1 0]
```

Note: as of this writing, plotting of loops and multiple edges is not supported, and the output is allowed to contain both types of edges.

```
sage: G = graphs.DegreeSequenceConfigurationModel([3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3,3])
sage: G.edges(labels=False)
[(0, 2), (0, 10), (0, 15), (1, 6), (1, 16), (1, 17), (2, 5), (2, 19), (3, 7), (3, 14), (3, 1
sage: G.show()  # long time
```

REFERENCE:

- [1] Newman, M.E.J. The Structure and function of complex networks, SIAM Review vol. 45, no. 2 (2003), pp. 167-256.

DegreeSequenceExpected(*deg_sequence, seed=None*)

Returns a random graph with expected given degree sequence. Raises a NetworkX error if the proposed degree sequence cannot be that of a graph.

One requirement is that the sum of the degrees must be even, since every edge must be incident with two vertices.

INPUT:

- `deg_sequence` - a list of integers with each entry corresponding to the expected degree of a different vertex.
- `seed` - for the random number generator.

EXAMPLE:

```
sage: G = graphs.DegreeSequenceExpected([1,2,3,2,3])
sage: G.edges(labels=False)
[(0, 2), (1, 1), (1, 3), (2, 2), (2, 4), (3, 3)]
sage: G.show()  # long time
```

REFERENCE:

- [1] Chung, Fan and Lu, L. Connected components in random graphs with given expected degree sequences. Ann. Combinatorics (6), 2002 pp. 125-145.

DegreeSequenceTree(*deg_sequence*)

Returns a tree with the given degree sequence. Raises a NetworkX error if the proposed degree sequence cannot be that of a tree.

Since every tree has one more vertex than edge, the degree sequence must satisfy len(deg_sequence) - sum(deg_sequence)/2 == 1.

INPUT:

- `deg_sequence` - a list of integers with each entry corresponding to the expected degree of a different vertex.

EXAMPLE:

```
sage: G = graphs.DegreeSequenceTree([3,1,3,3,1,1,1,2,1])
sage: G.show()  # long time
```

DesarguesGraph()

Returns the Desargues graph.

PLOTTING: The layout chosen is the same as on the cover of [1].

EXAMPLE:

```
sage: D = graphs.DesarguesGraph()
sage: L = graphs.LCFGraph(20,[5,-5,9,-9],5)
sage: D.is_isomorphic(L)
True
sage: D.show()  # long time
```

REFERENCE:

- [1] Harary, F. Graph Theory. Reading, MA: Addison-Wesley, 1994.

DiamondGraph()

Returns a diamond graph with 4 nodes.

A diamond graph is a square with one pair of diagonal nodes connected.

This constructor depends on NetworkX numeric labeling.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the diamond graph is drawn as a diamond, with the first node on top, second on the left, third on the right, and fourth on the bottom; with the second and third node connected.

EXAMPLES: Construct and show a diamond graph

```
sage: g = graphs.DiamondGraph()
sage: g.show() # long time
```

DodecahedralGraph()

Returns a Dodecahedral graph (with 20 nodes)

The dodecahedral graph is cubic symmetric, so the spring-layout algorithm will be very effective for display. It is dual to the icosahedral graph.

PLOTTING: The Dodecahedral graph should be viewed in 3 dimensions. We chose to use the default spring-layout algorithm here, so that multiple iterations might yield a different point of reference for the user. We hope to add rotatable, 3-dimensional viewing in the future. In such a case, a string argument will be added to select the flat spring-layout over a future implementation.

EXAMPLES: Construct and show a Dodecahdedral graph

```
sage: g = graphs.DodecahedralGraph()
sage: g.show() # long time
```

Create several dodecahedral graphs in a Sage graphics array They will be drawn differently due to the use of the spring-layout algorithm

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.DodecahedralGraph()
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
```

```
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

DorogovtsevGoltsevMendesGraph(*n*)

Construct the n-th generation of the Dorogovtsev-Goltsev-Mendes graph.

EXAMPLE:

```
sage: G = graphs.DorogovtsevGoltsevMendesGraph(8)
sage: G.size()
6561
```

REFERENCE:

- [1] Dorogovtsev, S. N., Goltsev, A. V., and Mendes, J. F. F., Pseudofractal scale-free web, Phys. Rev. E 066122 (2002).

EmptyGraph()

Returns an empty graph (0 nodes and 0 edges).

This is useful for constructing graphs by adding edges and vertices individually or in a loop.

PLOTTING: When plotting, this graph will use the default spring-layout algorithm, unless a position dictionary is specified.

EXAMPLES: Add one vertex to an empty graph and then show:

```
sage: empty1 = graphs.EmptyGraph()
sage: empty1.add_vertex()
sage: empty1.show() # long time
```

Use for loops to build a graph from an empty graph:

```
sage: empty2 = graphs.EmptyGraph()
sage: for i in range(5):
...     empty2.add_vertex() # add 5 nodes, labeled 0-4
...
sage: for i in range(3):
...     empty2.add_edge(i,i+1) # add edges {[0:1],[1:2],[2:3]}
...
sage: for i in range(4)[1:]:
...     empty2.add_edge(4,i) # add edges {[1:4],[2:4],[3:4]}
...
sage: empty2.show() # long time
```

FlowerSnark()

Returns a Flower Snark.

A flower snark has 20 vertices. It is part of the class of biconnected cubic graphs with edge chromatic number = 4, known as snarks. (i.e.: the Petersen graph). All snarks are not Hamiltonian, non-planar and have Petersen graph graph minors.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algoirithm. By convention, the nodes are drawn 0-14 on the outer circle, and 15-19 in an inner pentagon.

REFERENCES:

- [1] Weisstein, E. (1999). "Flower Snark - from Wolfram MathWorld". [Online] Available: http://mathworld.wolfram.com/FlowerSnark.html [2007, February 17]

EXAMPLES: Inspect a flower snark:

```
sage: F = graphs.FlowerSnark()
sage: F
Flower Snark: Graph on 20 vertices
sage: F.graph6_string()
'ShCGHC@?GGg@?@?Gp?K??C?CA?G?_G?Cc'
```

Now show it:

```
sage: F.show() # long time
```

FruchtGraph()

Returns a Frucht Graph.

A Frucht graph has 12 nodes and 18 edges. It is the smallest cubic identity graph. It is planar and it is Hamiltonian.

This constructor is dependant on Networkx's numeric labeling.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the first seven nodes are on the outer circle, with the next four on an inner circle and the last in the center.

REFERENCES:

- [1] Weisstein, E. (1999). "Frucht Graph - from Wolfram MathWorld". [Online] Available: http://mathworld.wolfram.com/FruchtGraph.html [2007, February 17]

EXAMPLES:

```
sage: FRUCHT = graphs.FruchtGraph()
sage: FRUCHT
Frucht graph: Graph on 12 vertices
sage: FRUCHT.graph6_string()
'KhCKM?_EGK?L'
sage: (graphs.FruchtGraph()).show() # long time
```

Grid2dGraph(*n1, n2*)

Returns a 2-dimensional grid graph with n1*n2 nodes (n1 rows and n2 columns).

A 2d grid graph resembles a 2 dimensional grid. All inner nodes are connected to their 4 neighbors. Outer (non-corner) nodes are connected to their 3 neighbors. Corner nodes are connected to their 2 neighbors.

This constructor depends on NetworkX numeric labels.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, nodes are labelled in (row, column) pairs with (0, 0) in the top left corner. Edges will always be horizontal and vertical - another advantage of filling the position dictionary.

EXAMPLES: Construct and show a grid 2d graph Rows = 5, Columns = 7

```
sage: g = graphs.Grid2dGraph(5,7)
sage: g.show() # long time
```

GridGraph(*dim_list*)

Returns an n-dimensional grid graph.

INPUT:

- `dim_list` - a list of integers representing the number of nodes to extend in each dimension.

PLOTTING: When plotting, this graph will use the default spring-layout algorithm, unless a position dictionary is specified.

EXAMPLES:

```
sage: G = graphs.GridGraph([2,3,4])
sage: G.show()  # long time

sage: C = graphs.CubeGraph(4)
sage: G = graphs.GridGraph([2,2,2,2])
sage: C.show()  # long time
sage: G.show()  # long time
```

HeawoodGraph()

Returns a Heawood graph.

The Heawood graph is a cage graph that has 14 nodes. It is a cubic symmetric graph. (See also the Moebius-Kantor graph). It is nonplanar and Hamiltonian. It has diameter = 3, radius = 3, girth = 6, chromatic number = 2. It is 4-transitive but not 5-transitive.

This constructor is dependant on Networkx's numeric labeling.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the nodes are positioned in a circular layout with the first node appearing at the top, and then continuing counterclockwise.

REFERENCES:

- [1] Weisstein, E. (1999). "Heawood Graph - from Wolfram MathWorld". [Online] Available: http://mathworld.wolfram.com/HeawoodGraph.html [2007, February 17]

EXAMPLES:

```
sage: H = graphs.HeawoodGraph()
sage: H
Heawood graph: Graph on 14 vertices
sage: H.graph6_string()
'MhEGHC@AI?_PC@_G_'
sage: (graphs.HeawoodGraph()).show() # long time
```

HexahedralGraph()

Returns a hexahedral graph (with 8 nodes).

A regular hexahedron is a 6-sided cube. The hexahedral graph corresponds to the connectivity of the vertices of the hexahedron. This graph is equivalent to a 3-cube.

PLOTTING: The hexahedral graph should be viewed in 3 dimensions. We chose to use the default spring-layout algorithm here, so that multiple iterations might yield a different point of reference for the user. We hope to add rotatable, 3-dimensional viewing in the future. In such a case, a string argument will be added to select the flat spring-layout over a future implementation.

EXAMPLES: Construct and show a Hexahedral graph

```
sage: g = graphs.HexahedralGraph()
sage: g.show() # long time
```

Create several hexahedral graphs in a Sage graphics array. They will be drawn differently due to the use of the spring-layout algorithm.

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.HexahedralGraph()
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

HoffmanSingletonGraph()

Returns the Hoffman-Singleton graph.

The Hoffman-Singleton graph is the Moore graph of degree 7, diameter 2 and girth 5. The Hoffman-Singleton theorem states that any Moore graph with girth 5 must have degree 2, 3, 7 or 57. The first three respectively are the pentagon, the Petersen graph, and the Hoffman-Singleton graph. The existence of a Moore graph with girth 5 and degree 57 is still open.

A Moore graph is a graph with diameter d and girth $2d + 1$. This implies that the graph is regular, and distance regular.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. A novel algorithm written by Tom Boothby gives a random layout which is pleasing to the eye.

REFERENCES:

•[1] Godsil, C. and Royle, G. Algebraic Graph Theory. Springer, 2001.

EXAMPLES:

```
sage: HS = graphs.HoffmanSingletonGraph()
sage: Set(HS.degree())
{7}
sage: HS.girth()
5
sage: HS.diameter()
2
sage: HS.num_verts()
50
```

HouseGraph()

Returns a house graph with 5 nodes.

A house graph is named for its shape. It is a triange (roof) over a square (walls).

This constructor depends on NetworkX numeric labeling.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the house graph is drawn with the first node in the lower-left corner of the house, the second in the lower-right corner of the house. The third node is in the upper-left corner connecting the roof to the wall, and the fourth is in the upper-right corner connecting the roof to the walll. The fifth node is the top of the roof, connected only to the third and fourth.

EXAMPLES: Construct and show a house graph

```
sage: g = graphs.HouseGraph()
sage: g.show() # long time
```

HouseXGraph()

Returns a house X graph with 5 nodes.

A house X graph is a house graph with two additional edges. The upper-right corner is connected to the lower-left. And the upper-left corner is connected to the lower-right.

This constructor depends on NetworkX numeric labeling.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the house X graph is drawn with the first node in the lower-left corner of the house, the second in the lower-right corner of the house. The third node is in the upper-left corner connecting the roof to the wall, and the fourth is in the upper-right corner connecting the roof to the walll. The fifth node is the top of the roof, connected only to the third and fourth.

EXAMPLES: Construct and show a house X graph

```
sage: g = graphs.HouseXGraph()
sage: g.show() # long time
```

IcosahedralGraph()

Returns an Icosahedral graph (with 12 nodes).

The regular icosahedron is a 20-sided triangular polyhedron. The icosahedral graph corresponds to the connectivity of the vertices of the icosahedron. It is dual to the dodecahedral graph. The icosahedron is symmetric, so the spring-layout algorithm will be very effective for display.

PLOTTING: The Icosahedral graph should be viewed in 3 dimensions. We chose to use the default spring-layout algorithm here, so that multiple iterations might yield a different point of reference for the user. We

hope to add rotatable, 3-dimensional viewing in the future. In such a case, a string argument will be added to select the flat spring-layout over a future implementation.

EXAMPLES: Construct and show an Octahedral graph

```
sage: g = graphs.IcosahedralGraph()
sage: g.show() # long time
```

Create several icosahedral graphs in a Sage graphics array. They will be drawn differently due to the use of the spring-layout algorithm.

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.IcosahedralGraph()
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

KrackhardtKiteGraph()

Returns a Krackhardt kite graph with 10 nodes.

The Krackhardt kite graph was originally developed by David Krackhardt for the purpose of studying social networks. It is used to show the distinction between: degree centrality, betweeness centrality, and closeness centrality. For more information read the plotting section below in conjunction with the example.

REFERENCES:

- [1] Kreps, V. (2002). "Social Network Analysis". [Online] Available: http://www.fsu.edu/~spap/water/network/intro.htm [2007, January 17]

This constructor depends on NetworkX numeric labeling.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the graph is drawn left to right, in top to bottom row sequence of [2, 3, 2, 1, 1, 1] nodes on each row. This places the fourth node (3) in the center of the kite, with the highest degree. But the fourth node only connects nodes that are otherwise connected, or those in its clique (i.e.: Degree Centrality). The eigth (7) node is where the kite meets the tail. It has degree = 3, less than the average, but is the only connection between the kite and tail (i.e.: Betweenness Centrality). The sixth and seventh nodes (5 and 6) are drawn in the third row and have degree = 5. These nodes have the shortest path to all other nodes in the graph (i.e.: Closeness Centrality). Please execute the example for visualization.

EXAMPLE: Construct and show a Krackhardt kite graph

```
sage: g = graphs.KrackhardtKiteGraph()
sage: g.show() # long time
```

LCFGraph(*n, shift_list, repeats*)

Returns the cubic graph specified in LCF notation.

LCF (Lederberg-Coxeter-Fruchte) notation is a concise way of describing cubic Hamiltonian graphs. The way a graph is constructed is as follows. Since there is a Hamiltonian cycle, we first create a cycle on n nodes. The variable shift_list = [s_0, s_1, ..., s_k-1] describes edges to be created by the following scheme: for each i, connect vertex i to vertex (i + s_i). Then, repeats specifies the number of times to repeat this process, where on the jth repeat we connect vertex (i + j*len(shift_list)) to vertex (i + j*len(shift_list) + s_i).

INPUT:

- n - the number of nodes.
- shift_list - a list of integer shifts mod n.
- repeats - the number of times to repeat the process.

EXAMPLES:

```
sage: G = graphs.LCFGraph(4, [2,-2], 2)
sage: G.is_isomorphic(graphs.TetrahedralGraph())
True

sage: G = graphs.LCFGraph(20, [10,7,4,-4,-7,10,-4,7,-7,4], 2)
sage: G.is_isomorphic(graphs.DodecahedralGraph())
True

sage: G = graphs.LCFGraph(14, [5,-5], 7)
sage: G.is_isomorphic(graphs.HeawoodGraph())
True
```

The largest cubic nonplanar graph of diameter three:

```
sage: G = graphs.LCFGraph(20, [-10,-7,-5,4,7,-10,-7,-4,5,7,-10,-7,6,-5,7,-10,-7,5,-6,7], 1)
sage: G.degree()
[3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3]
sage: G.diameter()
3
sage: G.show()  # long time
```

PLOTTING: LCF Graphs are plotted as an n-cycle with edges in the middle, as described above.

REFERENCES:

- [1] Frucht, R. "A Canonical Representation of Trivalent Hamiltonian Graphs." J. Graph Th. 1, 45-60, 1976.
- [2] Grunbaum, B. Convex Polytop es. New York: Wiley, pp. 362-364, 1967.
- [3] Lederberg, J. 'DENDRAL-64: A System for Computer Construction, Enumeration and Notation of Organic Molecules as Tree Structures and Cyclic Graphs. Part II. Topology of Cyclic Graphs.' Interim Report to the National Aeronautics and Space Administration. Grant NsG 81-60. December 15, 1965. http://profiles.nlm.nih.gov/BB/A/B/I/U/_/bbabiu.pdf.

LadderGraph(*n*)

Returns a ladder graph with 2*n nodes.

A ladder graph is a basic structure that is typically displayed as a ladder, i.e.: two parallel path graphs connected at each corresponding node pair.

This constructor depends on NetworkX numeric labels.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each ladder graph will be displayed horizontally, with the first n nodes displayed left to right on the top horizontal line.

EXAMPLES: Construct and show a ladder graph with 14 nodes

```
sage: g = graphs.LadderGraph(7)
sage: g.show() # long time
```

Create several ladder graphs in a Sage graphics array

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.LadderGraph(i+2)
...     g.append(k)
...
sage: for i in range(3):
```

```
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

LollipopGraph(*n1, n2*)

Returns a lollipop graph with n1+n2 nodes.

A lollipop graph is a path graph (order n2) connected to a complete graph (order n1). (A barbell graph minus one of the bells).

This constructor depends on NetworkX numeric labels.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the complete graph will be drawn in the lower-left corner with the (n1)th node at a 45 degree angle above the right horizontal center of the complete graph, leading directly into the path graph.

EXAMPLES: Construct and show a lollipop graph Candy = 13, Stick = 4

```
sage: g = graphs.LollipopGraph(13,4)
sage: g.show() # long time
```

Create several lollipop graphs in a Sage graphics array

```
sage: g = []
sage: j = []
sage: for i in range(6):
...     k = graphs.LollipopGraph(i+3,4)
...     g.append(k)
...
sage: for i in range(2):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

MoebiusKantorGraph()

Returns a Moebius-Kantor Graph.

A Moebius-Kantor graph is a cubic symmetric graph. (See also the Heawood graph). It has 16 nodes and 24 edges. It is nonplanar and Hamiltonian. It has diameter = 4, girth = 6, and chromatic number = 2. It is identical to the Generalized Petersen graph, P[8,3].

PLOTTING: Upon construction, the position dictionary is filled to overwrite the spring-layout algorithm. By convention, the first 8 nodes are drawn counter-clockwise in an outer circle, with the remaining eight drawn likewise nested in a smaller circular pattern. The Moebius-Kantor graph is constructed directly below from a dictionary with nodes as keys and entries represented the nodes they are connected to. Please browse this dictionary or display an example to further understand the plotting convention.

REFERENCES:

- [1] Weisstein, E. (1999). "Moebius-Kantor Graph - from Wolfram MathWorld". [Online] Available: http://mathworld.wolfram.com/Moebius-KantorGraph.html [2007, February 17]

EXAMPLES:

```
sage: MK = graphs.MoebiusKantorGraph()
sage: MK
Moebius-Kantor Graph: Graph on 16 vertices
```

```
sage: MK.graph6_string()
'OhCGKE?O@?ACAC@I?Q_AS'
sage: (graphs.MoebiusKantorGraph()).show() # long time
```

OctahedralGraph()

Returns an Octahedral graph (with 6 nodes).

The regular octahedron is an 8-sided polyhedron with triangular faces. The octahedral graph corresponds to the connectivity of the vertices of the octahedron. It is the line graph of the tetrahedral graph. The octahedral is symmetric, so the spring-layout algorithm will be very effective for display.

PLOTTING: The Octahedral graph should be viewed in 3 dimensions. We chose to use the default spring-layout algorithm here, so that multiple iterations might yield a different point of reference for the user. We hope to add rotatable, 3-dimensional viewing in the future. In such a case, a string argument will be added to select the flat spring-layout over a future implementation.

EXAMPLES: Construct and show an Octahedral graph

```
sage: g = graphs.OctahedralGraph()
sage: g.show() # long time
```

Create several octahedral graphs in a Sage graphics array They will be drawn differently due to the use of the spring-layout algorithm

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.OctahedralGraph()
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

PappusGraph()

Returns the Pappus graph, a graph on 18 vertices.

The Pappus graph is cubic, symmetric, and distance-regular.

EXAMPLES:

```
sage: G = graphs.PappusGraph()
sage: G.show()  # long time
sage: L = graphs.LCFGraph(18, [5,7,-7,7,-7,-5], 3)
sage: L.show()  # long time
sage: G.is_isomorphic(L)
True
```

PathGraph(*n, pos=None*)

Returns a path graph with n nodes. Pos argument takes a string which is either 'circle' or 'line', (otherwise the default is used). See the plotting section below for more detail.

A path graph is a graph where all inner nodes are connected to their two neighbors and the two end-nodes are connected to their one inner neighbors. (i.e.: a cycle graph without the first and last node connected).

This constructor depends on NetworkX numeric labels.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, the graph may be drawn in one of two ways: The 'line' argument will draw the graph in a horizontal line (left to right) if there are less than 11 nodes. Otherwise the 'line' argument will append

horizontal lines of length 10 nodes below, alternating left to right and right to left. The 'circle' argument will cause the graph to be drawn in a cycle-shape, with the first node at the top and then about the circle in a clockwise manner. By default (without an appropriate string argument) the graph will be drawn as a 'circle' if 10 n 41 and as a 'line' for all other n.

EXAMPLES: Show default drawing by size: 'line': n 11

```
sage: p = graphs.PathGraph(10)
sage: p.show() # long time
```

'circle': 10 n 41

```
sage: q = graphs.PathGraph(25)
sage: q.show() # long time
```

'line': n 40

```
sage: r = graphs.PathGraph(55)
sage: r.show() # long time
```

Override the default drawing:

```
sage: s = graphs.PathGraph(5,'circle')
sage: s.show() # long time
```

PetersenGraph()

The Petersen Graph is a named graph that consists of 10 vertices and 15 edges, usually drawn as a five-point star embedded in a pentagon.

The Petersen Graph is a common counterexample. For example, it is not Hamiltonian.

PLOTTING: When plotting the Petersen graph with the spring-layout algorithm, we see that this graph is not very symmetric and thus the display may not be very meaningful. Efficiency of construction and plotting is not an issue, as the Petersen graph only has 10 vertices.

Our labeling convention here is to start on the outer pentagon from the top, moving counterclockwise. Then the nodes on the inner star, starting at the top and moving counterclockwise.

EXAMPLES: We compare below the Petersen graph with the default spring-layout versus a planned position dictionary of [x,y] tuples:

```
sage: petersen_spring = Graph({0:[1,4,5], 1:[0,2,6], 2:[1,3,7], 3:[2,4,8], 4:[0,3,9], 5:[0,7
sage: petersen_spring.show() # long time
sage: petersen_database = graphs.PetersenGraph()
sage: petersen_database.show() # long time
```

RandomBarabasiAlbert(*n, m, seed=None*)

Return a random graph created using the Barabasi-Albert preferential attachment model.

A graph with m vertices and no edges is initialized, and a graph of n vertices is grown by attaching new veritces each with m edges that are attached to existing vertices, preferentially with high degree.

INPUT:

- `n` - number of vertices in the graph
- `m` - number of edges to attach from each new node
- `seed` - for random number generator

EXAMPLES:

We show the edge list of a random graph on 6 nodes with m = 2.

```
sage: graphs.RandomBarabasiAlbert(6,2).edges(labels=False)
[(0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (2, 4), (2, 5), (3, 5)]
```

We plot a random graph on 12 nodes with m = 3.

```
sage: ba = graphs.RandomBarabasiAlbert(12,3)
sage: ba.show()  # long time
```

We view many random graphs using a graphics array:

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.RandomBarabasiAlbert(i+3, 3)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show()  # long time
```

RandomGNM(*n, m, dense=False, seed=None*)

Returns a graph randomly picked out of all graphs on n vertices with m edges.

INPUT:

- `n` - number of vertices.
- `m` - number of edges.
- `dense` - whether to use NetworkX's dense_gnm_random_graph or gnm_random_graph

EXAMPLES: We show the edge list of a random graph on 5 nodes with 10 edges.

```
sage: graphs.RandomGNM(5, 10).edges(labels=False)
[(0, 1), (0, 2), (0, 3), (0, 4), (1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

We plot a random graph on 12 nodes with m = 12.

```
sage: gnm = graphs.RandomGNM(12, 12)
sage: gnm.show()  # long time
```

We view many random graphs using a graphics array:

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.RandomGNM(i+3, i^2-i)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show()  # long time
```

RandomGNP(*n, p, seed=None, fast=True*)

Returns a Random graph on n nodes. Each edge is inserted independently with probability p.

IMPLEMENTATION: This function calls the NetworkX function `fast_gnp_random_graph`, unless fast==False, then `gnp_random_graph`.

REFERENCES:

- [1] P. Erdos and A. Renyi, On Random Graphs, Publ. Math. 6, 290 (1959).
- [2] E. N. Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

PLOTTING: When plotting, this graph will use the default spring-layout algorithm, unless a position dictionary is specified.

EXAMPLES: We show the edge list of a random graph on 6 nodes with probability $p = .4$:

```
sage: graphs.RandomGNP(6, .4).edges(labels=False)
[(0, 1), (0, 3), (0, 4), (0, 5), (1, 2), (1, 3), (1, 4), (1, 5)]
```

We plot a random graph on 12 nodes with probability $p = .71$:

```
sage: gnp = graphs.RandomGNP(12,.71)
sage: gnp.show() # long time
```

We view many random graphs using a graphics array:

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.RandomGNP(i+3,.43)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
sage: graphs.RandomGNP(4,1)
Complete graph: Graph on 4 vertices
```

TIMINGS: The following timings compare the speed with fast==False and fast==True for sparse and dense graphs. (It's no different?)

```
sage: t=cputime(); regular_sparse = graphs.RandomGNP(389,.22)
sage: cputime(t)      # slightly random
0.2240130000000029

sage: t=cputime(); fast_sparse =  graphs.RandomGNP(389,.22,fast=True)
sage: cputime(t)      # slightly random
0.22401400000000038

sage: t=cputime(); regular_dense = graphs.RandomGNP(389,.88)     # long time
sage: cputime(t)      # slightly random, long time
0.87205499999999958

sage: t=cputime(); fast_dense = graphs.RandomGNP(389,.88,fast=True)     # long time
sage: cputime(t)      # slightly random, long time
0.90005700000000033
```

RandomHolmeKim(*n, m, p, seed=None*)

Returns a random graph generated by the Holme and Kim algorithm for graphs with powerlaw degree distribution and approximate average clustering.

INPUT:

- n - number of vertices.
- m - number of random edges to add for each new node.
- p - probability of adding a triangle after adding a random edge.

- seed - for the random number generator.

From the NetworkX documentation: The average clustering has a hard time getting above a certain cutoff that depends on m. This cutoff is often quite low. Note that the transitivity (fraction of triangles to possible triangles) seems to go down with network size. It is essentially the Barabasi-Albert growth model with an extra step that each random edge is followed by a chance of making an edge to one of its neighbors too (and thus a triangle). This algorithm improves on B-A in the sense that it enables a higher average clustering to be attained if desired. It seems possible to have a disconnected graph with this algorithm since the initial m nodes may not be all linked to a new node on the first iteration like the BA model.

EXAMPLE: We show the edge list of a random graph on 8 nodes with 2 random edges per node and a probability $p = 0.5$ of forming triangles.

```
sage: graphs.RandomHolmeKim(8, 2, 0.5).edges(labels=False)
[(0, 2), (0, 4), (1, 2), (1, 3), (2, 3), (3, 4), (3, 5), (3, 6), (3, 7), (4, 5), (4, 6)]

sage: G = graphs.RandomHolmeKim(12, 3, .3)
sage: G.show()  # long time
```

REFERENCE:

- [1] Holme, P. and Kim, B.J. Growing scale-free networks with tunable clustering, Phys. Rev. E (2002). vol 65, no 2, 026107.

RandomLobster(*n, p, q, seed=None*)

Returns a random lobster.

A lobster is a tree that reduces to a caterpillar when pruning all leaf vertices. A caterpillar is a tree that reduces to a path when pruning all leaf vertices (q=0).

INPUT:

- n - expected number of vertices in the backbone
- p - probability of adding an edge to the backbone
- q - probability of adding an edge (claw) to the arms
- seed - for the random number generator

EXAMPLE: We show the edge list of a random graph with 3 backbone nodes and probabilities $p = 0.7$ and $q = 0.3$:

```
sage: graphs.RandomLobster(3, 0.7, 0.3).edges(labels=False)
[(0, 1), (1, 2)]

sage: G = graphs.RandomLobster(9, .6, .3)
sage: G.show()  # long time
```

RandomNewmanWattsStrogatz(*n, k, p, seed=None*)

Returns a Newman-Watts-Strogatz small world random graph on n vertices.

From the NetworkX documentation: First create a ring over n nodes. Then each node in the ring is connected with its k nearest neighbors. Then shortcuts are created by adding new edges as follows: for each edge u-v in the underlying "n-ring with k nearest neighbors"; with probability p add a new edge u-w with randomly-chosen existing node w. In contrast with watts_strogatz_graph(), no edges are removed.

INPUT:

- n - number of vertices.
- k - each vertex is connected to its k nearest neighbors
- p - the probability of adding a new edge for each edge
- seed - for the random number generator

EXAMPLE: We show the edge list of a random graph on 7 nodes with 2 "nearest neighbors" and probability $p = 0.2$:

```
sage: graphs.RandomNewmanWattsStrogatz(7, 2, 0.2).edges(labels=False)
[(0, 1), (0, 2), (0, 3), (0, 6), (1, 2), (2, 3), (2, 4), (3, 4), (3, 6), (4, 5), (5, 6)]

sage: G = graphs.RandomNewmanWattsStrogatz(12, 2, .3)
sage: G.show()  # long time
```

REFERENCE:

- [1] Newman, M.E.J., Watts, D.J. and Strogatz, S.H. Random graph models of social networks. Proc. Nat. Acad. Sci. USA 99, 2566-2572.

RandomRegular(*d, n, seed=None*)

Returns a random d-regular graph on n vertices, or returns False on failure.

Since every edge is incident to two vertices, n*d must be even.

INPUT:

- `n` - number of vertices
- `d` - degree
- `seed` - for the random number generator

EXAMPLE: We show the edge list of a random graph with 8 nodes each of degree 3.

```
sage: graphs.RandomRegular(3, 8)
Graph on 0 vertices
sage: graphs.RandomRegular(3, 8)
Graph on 0 vertices
sage: graphs.RandomRegular(3, 8).edges(labels=False)
[(0, 1), (0, 4), (0, 5), (1, 6), (1, 7), (2, 3), (2, 4), (2, 7), (3, 4), (3, 5), (5, 6), (6,

sage: G = graphs.RandomRegular(3, 20)
sage: if G:
...     G.show()  # random output, long time
```

REFERENCES:

- [1] Kim, Jeong Han and Vu, Van H. Generating random regular graphs. Proc. 35th ACM Symp. on Thy. of Comp. 2003, pp 213-222. ACM Press, San Diego, CA, USA. http://doi.acm.org/10.1145/780542.780576
- [2] Steger, A. and Wormald, N. Generating random regular graphs quickly. Prob. and Comp. 8 (1999), pp 377-396.

RandomShell(*constructor, seed=None*)

Returns a random shell graph for the constructor given.

INPUT:

- `constructor` - a list of 3-tuples (n,m,d), each representing a shell
- `n` - the number of vertices in the shell
- `m` - the number of edges in the shell
- `d` - the ratio of inter (next) shell edges to intra shell edges
- `seed` - for the random number generator

EXAMPLE:

```
sage: G = graphs.RandomShell([(10,20,0.8),(20,40,0.8)])
sage: G.edges(labels=False)
[(0, 3), (0, 7), (0, 8), (1, 2), (1, 5), (1, 8), (1, 9), (3, 6), (3, 11), (4, 6), (4, 7), (4
sage: G.show()  # long time
```

RandomTreePowerlaw(*n, gamma=3, tries=100, seed=None*)

Returns a tree with a powerlaw degree distribution. Returns False on failure.

From the NetworkX documentation: A trial powerlaw degree sequence is chosen and then elements are swapped with new elements from a powerlaw distribution until the sequence makes a tree (size = order - 1).

INPUT:

- `n` - number of vertices
- `gamma` - exponent of power law
- `tries` - number of attempts to adjust sequence to make a tree
- `seed` - for the random number generator

EXAMPLE: We show the edge list of a random graph with 10 nodes and a power law exponent of 2.

```
sage: graphs.RandomTreePowerlaw(10, 2).edges(labels=False)
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 7), (6, 8), (6, 9)]

sage: G = graphs.RandomTreePowerlaw(15, 2)
sage: if G:
...     G.show()  # random output, long time
```

StarGraph(*n*)

Returns a star graph with n+1 nodes.

A Star graph is a basic structure where one node is connected to all other nodes.

This constructor is dependant on NetworkX numeric labels.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each star graph will be displayed with the first (0) node in the center, the second node (1) at the top, with the rest following in a counterclockwise manner. (0) is the node connected to all other nodes.

The star graph is a good opportunity to compare efficiency of filling a position dictionary vs. using the spring-layout algorithm for plotting. As far as display, the spring-layout should push all other nodes away from the (0) node, and thus look very similar to this constructor's positioning.

EXAMPLES:

```
sage: import networkx
```

Compare the plots:

```
sage: n = networkx.star_graph(23)
sage: spring23 = Graph(n)
sage: posdict23 = graphs.StarGraph(23)
sage: spring23.show() # long time
sage: posdict23.show() # long time
```

View many star graphs as a Sage Graphics Array

With this constructor (i.e., the position dictionary filled)

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.StarGraph(i+3)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

Compared to plotting with the spring-layout algorithm

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     spr = networkx.star_graph(i+3)
...     k = Graph(spr)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

TetrahedralGraph()

Returns a tetrahedral graph (with 4 nodes).

A tetrahedron is a 4-sided triangular pyramid. The tetrahedral graph corresponds to the connectivity of the vertices of the tetrahedron. This graph is equivalent to a wheel graph with 4 nodes and also a complete graph on four nodes. (See examples below).

PLOTTING: The tetrahedral graph should be viewed in 3 dimensions. We chose to use the default spring-layout algorithm here, so that multiple iterations might yield a different point of reference for the user. We hope to add rotatable, 3-dimensional viewing in the future. In such a case, a string argument will be added to select the flat spring-layout over a future implementation.

EXAMPLES: Construct and show a Tetrahedral graph

```
sage: g = graphs.TetrahedralGraph()
sage: g.show() # long time
```

The following example requires networkx:

```
sage: import networkx as NX
```

Compare this Tetrahedral, Wheel(4), Complete(4), and the Tetrahedral plotted with the spring-layout algorithm below in a Sage graphics array:

```
sage: tetra_pos = graphs.TetrahedralGraph()
sage: tetra_spring = Graph(NX.tetrahedral_graph())
sage: wheel = graphs.WheelGraph(4)
sage: complete = graphs.CompleteGraph(4)
sage: g = [tetra_pos, tetra_spring, wheel, complete]
sage: j = []
sage: for i in range(2):
...     n = []
...     for m in range(2):
...         n.append(g[i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

ThomsenGraph()

Returns the Thomsen Graph.

The Thomsen Graph is actually a complete bipartite graph with (n1, n2) = (3, 3). It is also called the Utility graph.

PLOTTING: See CompleteBipartiteGraph.

EXAMPLES:

```
sage: T = graphs.ThomsenGraph()
sage: T
Thomsen graph: Graph on 6 vertices
sage: T.graph6_string()
'EFz_'
sage: (graphs.ThomsenGraph()).show() # long time
```

WheelGraph(*n*)

Returns a Wheel graph with n nodes.

A Wheel graph is a basic structure where one node is connected to all other nodes and those (outer) nodes are connected cyclically.

This constructor depends on NetworkX numeric labels.

PLOTTING: Upon construction, the position dictionary is filled to override the spring-layout algorithm. By convention, each wheel graph will be displayed with the first (0) node in the center, the second node at the top, and the rest following in a counterclockwise manner.

With the wheel graph, we see that it doesn't take a very large n at all for the spring-layout to give a counter-intuitive display. (See Graphics Array examples below).

EXAMPLES: We view many wheel graphs with a Sage Graphics Array, first with this constructor (i.e., the position dictionary filled):

```
sage: g = []
sage: j = []
sage: for i in range(9):
...     k = graphs.WheelGraph(i+3)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

Next, using the spring-layout algorithm:

```
sage: import networkx
sage: g = []
sage: j = []
sage: for i in range(9):
...     spr = networkx.wheel_graph(i+3)
...     k = Graph(spr)
...     g.append(k)
...
sage: for i in range(3):
...     n = []
...     for m in range(3):
...         n.append(g[3*i + m].plot(vertex_size=50, vertex_labels=False))
...     j.append(n)
...
sage: G = sage.plot.plot.GraphicsArray(j)
sage: G.show() # long time
```

Compare the plotting:

```
sage: n = networkx.wheel_graph(23)
sage: spring23 = Graph(n)
```

```
sage: posdict23 = graphs.WheelGraph(23)
sage: spring23.show() # long time
sage: posdict23.show() # long time
```

nauty_geng(*options=''*)

Calls the geng program in the optional nauty spkg to generate graphs. The options argument is passed straight to nauty.

INPUT:

- `options` - a string passed to the command line of geng. You *must* pass the number of vertices you desire.

EXAMPLES:

```
sage: graph_list = graphs.nauty_geng("-q 3") # requires the optional nauty package
sage: len(graph_list) # requires the optional nauty package
4
```

trees(*vertices*)

Accesses the generator of trees (graphs without cycles). Iterates over distinct, exhaustive representatives.

INPUT:

- `vertices` - natural number

EXAMPLES: Sloane A000055:

```
sage: for i in range(0, 7):
...     print len(list(graphs.trees(i)))
1
1
1
1
2
3
6
sage: for i in range(7, 10):              # long time
...     print len(list(graphs.trees(i))) # long time
11
23
47
```

canaug_traverse_edge(*g, aut_gens, property, dig=False, loops=False, implementation='networkx', sparse=True*)

Main function for exhaustive generation. Recursive traversal of a canonically generated tree of isomorph free graphs satisfying a given property.

INPUT:

- `g` - current position on the tree.
- `aut_gens` - list of generators of Aut(g), in list notation.
- `property` - check before traversing below g.

EXAMPLES:

```
sage: from sage.graphs.graph_generators import canaug_traverse_edge
sage: G = Graph(3)
sage: list(canaug_traverse_edge(G, [], lambda x: True))
[Graph on 3 vertices, ... Graph on 3 vertices]
```

The best way to access this function is through the graphs() iterator:

Print graphs on 3 or less vertices.

```
sage: for G in graphs(3):
...     print G
...
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
```

Print digraphs on 3 or less vertices.

```
sage: for G in digraphs(3):
...     print G
...
Digraph on 3 vertices
Digraph on 3 vertices
...
Digraph on 3 vertices
Digraph on 3 vertices
```

canaug_traverse_vert(*g, aut_gens, max_verts, property, dig=False, loops=False, implementation='networkx', sparse=True*)

Main function for exhaustive generation. Recursive traversal of a canonically generated tree of isomorph free (di)graphs satisfying a given property.

INPUT:

- `g` - current position on the tree.
- `aut_gens` - list of generators of Aut(g), in list notation.
- `max_verts` - when to retreat.
- `property` - check before traversing below g.
- `deg_seq` - specify a degree sequence to try to obtain.

EXAMPLES:

```
sage: from sage.graphs.graph_generators import canaug_traverse_vert
sage: list(canaug_traverse_vert(Graph(), [], 3, lambda x: True))
[Graph on 0 vertices, ... Graph on 3 vertices]
```

The best way to access this function is through the graphs() iterator:

Print graphs on 3 or less vertices.

```
sage: for G in graphs(3, augment='vertices'):
...     print G
...
Graph on 0 vertices
Graph on 1 vertex
Graph on 2 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 3 vertices
Graph on 2 vertices
Graph on 3 vertices
```

Print digraphs on 2 or less vertices.

```
sage: for D in digraphs(2, augment='vertices'):
...     print D
...
Digraph on 0 vertices
Digraph on 1 vertex
Digraph on 2 vertices
Digraph on 2 vertices
Digraph on 2 vertices
```

check_aut (*aut_gens, cut_vert, n*)

Helper function for exhaustive generation.

At the start, check_aut is given a set of generators for the automorphism group, aut_gens. We already know we are looking for an element of the auto- morphism group that sends cut_vert to n, and check_aut generates these for the canaug_traverse function.

EXAMPLE: Note that the last two entries indicate that none of the automorphism group has yet been searched - we are starting at the identity [0, 1, 2, 3] and so far that is all we have seen. We return automorphisms mapping 2 to 3.

```
sage: from sage.graphs.graph_generators import check_aut
sage: list( check_aut( [ [0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3] ], 2, 3))
[[1, 0, 3, 2], [1, 2, 3, 0]]
```

check_aut_edge (*aut_gens, cut_edge, i, j, n, dig=False*)

Helper function for exhaustive generation.

At the start, check_aut_edge is given a set of generators for the automorphism group, aut_gens. We already know we are looking for an element of the auto- morphism group that sends cut_edge to {i, j}, and check_aut generates these for the canaug_traverse function.

EXAMPLE: Note that the last two entries indicate that none of the automorphism group has yet been searched - we are starting at the identity [0, 1, 2, 3] and so far that is all we have seen. We return automorphisms mapping 2 to 3.

```
sage: from sage.graphs.graph_generators import check_aut
sage: list( check_aut( [ [0, 3, 2, 1], [1, 0, 3, 2], [2, 1, 0, 3] ], 2, 3))
[[1, 0, 3, 2], [1, 2, 3, 0]]
```

7.3 N.I.C.E. - Nice (as in open source) Isomorphism Check Engine

Automorphism group computation and isomorphism checking for graphs.

This is an open source implementation of Brendan McKay's algorithm for graph automorphism and isomorphism. McKay released a C version of his algorithm, named nauty (No AUTomorphisms, Yes?) under a license that is not GPL compatible. Although the program is open source, reading the source disallows anyone from recreating anything similar and releasing it under the GPL. Also, many people have complained that the code is difficult to understand. The first main goal of NICE was to produce a genuinely open graph isomorphism program, which has been accomplished. The second goal is for this code to be understandable, so that computed results can be trusted and further derived work will be possible.

To determine the isomorphism type of a graph, it is convenient to define a canonical label for each isomorphism class- essentially an equivalence class representative. Loosely (albeit incorrectly), the canonical label is defined by enumerating all labeled graphs, then picking the maximal one in each isomorphism class. The NICE algorithm is essentially a backtrack search. It searches through the rooted tree of partition nests (where each partition is equitable) for implicit and explicit automorphisms, and uses this information to eliminate large parts of the tree from further

searching. Since the leaves of the search tree are all discrete ordered partitions, each one of these corresponds to an ordering of the vertices of the graph, i.e. another member of the isomorphism class. Once the algorithm has finished searching the tree, it will know which leaf corresponds to the canonical label. In the process, generators for the automorphism group are also produced.

AUTHORS:

- Robert L. Miller (2007-03-20): initial version
- Tom Boothby (2007-03-20): help with indicator function
- Robert L. Miller (2007-04-07-30): optimizations
- Robert L. Miller (2007-07-07-14): PartitionStack and OrbitPartition
- Tom Boothby (2007-07-14) datastructure advice
- Robert L. Miller (2007-07-16-20): bug fixes

REFERENCE:

- [1] McKay, Brendan D. Practical Graph Isomorphism. Congressus Numerantium, Vol. 30 (1981), pp. 45-87.

Note:

1. Often we assume that G is a graph on vertices 0,1,...,n-1.
2. There is no s == loads(dumps(s)) type test since none of the classes defined here are meant to be instantiated for longer than the algorithm runs (i.e. pickling is not relevant here).

class `OrbitPartition`()

An OrbitPartition is simply a partition which keeps track of the orbits of the part of the automorphism group so far discovered. Essentially a union-find datastructure.

EXAMPLES:

```
sage: from sage.graphs.graph_isom import OrbitPartition
sage: K = OrbitPartition(20)
sage: K.find(7)
7
sage: K.union_find(7, 12)
sage: K.find(12)
7
sage: J = OrbitPartition(20)
sage: J.is_finer_than(K, 20)
True
sage: K.is_finer_than(J, 20)
False
```

```
sage: from sage.graphs.graph_isom import OrbitPartition
sage: Theta1 = OrbitPartition(10)
sage: Theta2 = OrbitPartition(10)
sage: Theta1.union_find(0,1)
sage: Theta1.union_find(2,3)
sage: Theta1.union_find(3,4)
sage: Theta1.union_find(5,6)
sage: Theta1.union_find(8,9)
sage: Theta2.vee_with(Theta1, 10)
```

```
sage: for i in range(10):
...       print i, Theta2.find(i)
0 0
1 0
2 2
3 2
4 2
5 5
6 5
7 7
8 8
9 8
```

find()

Returns an element of the cell which depends only on the cell.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import OrbitPartition
sage: K = OrbitPartition(20)
```

0 and 1 begin in different cells:

```
sage: K.find(0)
0
sage: K.find(1)
1
```

Now we put them in the same cell:

```
sage: K.union_find(0,1)
sage: K.find(0)
0
sage: K.find(1)
0
```

incorporate_permutation()

Unions the cells of self which contain common elements of some orbit of gamma.

INPUT:

- gamma - a permutation, in list notation

EXAMPLE:

```
sage: from sage.graphs.graph_isom import OrbitPartition
sage: O = OrbitPartition(9)
sage: O.incorporate_permutation([0,1,3,2,5,6,7,4,8])
sage: for i in range(9):
...    print i, O.find(i)
0 0
1 1
2 2
3 2
4 4
5 4
6 4
7 4
8 8
```

is_finer_than()

Partition P is finer than partition Q if every cell of P is a subset of a cell of Q.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import OrbitPartition
sage: K = OrbitPartition(20)
sage: K.find(7)
7
sage: K.union_find(7, 12)
sage: K.find(12)
7
sage: J = OrbitPartition(20)
sage: J.is_finer_than(K, 20)
True
sage: K.is_finer_than(J, 20)
False
```

union_find()

Merges the cells containing a and b.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import OrbitPartition
sage: K = OrbitPartition(20)
```

0 and 1 begin in different cells:

```
sage: K.find(0)
0
sage: K.find(1)
1
```

Now we put them in the same cell:

```
sage: K.union_find(0,1)
sage: K.find(0)
0
sage: K.find(1)
0
```

vee_with()

Merges the minimal number of cells such that other is finer than self.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import OrbitPartition
sage: K = OrbitPartition(20)
sage: K.union_find(7, 12)
sage: J = OrbitPartition(20)
sage: J.is_finer_than(K, 20)
True
sage: K.is_finer_than(J, 20)
False
sage: J.vee_with(K, 20)
sage: K.is_finer_than(J, 20)
True
```

class **PartitionStack**()

TODO: documentation

EXAMPLES:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: P = PartitionStack([range(9, -1, -1)])
sage: P.set_k(1)
sage: P.sort_by_function(0, [2,1,2,1,2,1,3,4,2,1], 10)
```

```
0
sage: P.set_k(2)
sage: P.sort_by_function(0, [2,1,2,1], 10)
0
sage: P.set_k(3)
sage: P.sort_by_function(4, [2,1,2,1], 10)
4
sage: P.set_k(4)
sage: P.sort_by_function(0, [0,1], 10)
0
sage: P.set_k(5)
sage: P.sort_by_function(2, [1,0], 10)
2
sage: P.set_k(6)
sage: P.sort_by_function(4, [1,0], 10)
4
sage: P.set_k(7)
sage: P.sort_by_function(6, [1,0], 10)
6
sage: P
(5,9,7,1,6,2,8,0,4,3)
(5,9,7,1|6,2,8,0|4|3)
(5,9|7,1|6,2,8,0|4|3)
(5,9|7,1|6,2|8,0|4|3)
(5|9|7,1|6,2|8,0|4|3)
(5|9|7|1|6,2|8,0|4|3)
(5|9|7|1|6|2|8,0|4|3)
(5|9|7|1|6|2|8|0|4|3)
sage: P.is_discrete()
1
sage: P.set_k(6)
sage: P.is_discrete()
0


sage: G = SparseGraph(10)
sage: for i,j,_ in graphs.PetersenGraph().edge_iterator():
...     G.add_arc(i,j)
...     G.add_arc(j,i)
sage: P = PartitionStack(10)
sage: P.set_k(1)
sage: P.split_vertex(0)
sage: P.refine(G, [0], 10, 0, 1)
sage: P
(0,2,3,6,7,8,9,1,4,5)
(0|2,3,6,7,8,9|1,4,5)
sage: P.set_k(2)
sage: P.split_vertex(1)
sage: P.refine(G, [7], 10, 0, 1)
sage: P
(0,3,7,8,9,2,6,1,4,5)
(0|3,7,8,9,2,6|1,4,5)
(0|3,7,8,9|2,6|1|4,5)
```

clear()

Merges all cells in the partition stack.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: P = PartitionStack([range(9, -1, -1)])
sage: P.set_k(1)
sage: P
(0,9,8,7,6,5,4,3,2,1)
(0,9,8,7,6,5,4,3,2,1)
sage: P.sort_by_function(0, [2,1,2,1,2,1,3,4,2,1], 10)
0
sage: P
(1,9,7,5,0,2,8,6,4,3)
(1,9,7,5|0,2,8,6|4|3)
sage: P
(1,9,7,5,0,2,8,6,4,3)
(1,9,7,5|0,2,8,6|4|3)
sage: P.clear()
sage: P
(1,9,7,5,0,2,8,6,4,3)
(1,9,7,5,0,2,8,6,4,3)
```

degree()

Returns the number of edges in G from self.entries[v] to a vertex in W.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: P = PartitionStack([range(9, -1, -1)])
sage: P
(0,9,8,7,6,5,4,3,2,1)
sage: G = SparseGraph(10)
sage: G.add_arc(2,9)
sage: G.add_arc(3,9)
sage: G.add_arc(4,9)
sage: P.degree(G, 1, 0)
3
```

is_discrete()

Returns whether the partition consists of only singletons.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: P = PartitionStack([range(9, -1, -1)])
sage: P.set_k(1)
sage: P.sort_by_function(0, [2,1,2,1,2,1,3,4,2,1], 10)
0
sage: P.set_k(2)
sage: P.sort_by_function(0, [2,1,2,1], 10)
0
sage: P.set_k(3)
sage: P.sort_by_function(4, [2,1,2,1], 10)
4
sage: P.set_k(4)
sage: P.sort_by_function(0, [0,1], 10)
0
sage: P.set_k(5)
sage: P.sort_by_function(2, [1,0], 10)
2
sage: P.set_k(6)
sage: P.sort_by_function(4, [1,0], 10)
```

```
4
sage: P.set_k(7)
sage: P.sort_by_function(6, [1,0], 10)
6
sage: P
(5,9,7,1,6,2,8,0,4,3)
(5,9,7,1|6,2,8,0|4|3)
(5,9|7,1|6,2,8,0|4|3)
(5,9|7,1|6,2|8,0|4|3)
(5|9|7,1|6,2|8,0|4|3)
(5|9|7|1|6,2|8,0|4|3)
(5|9|7|1|6|2|8,0|4|3)
(5|9|7|1|6|2|8|0|4|3)
sage: P.is_discrete()
True
sage: P.set_k(2)
sage: P
(5,9,7,1,6,2,8,0,4,3)
(5,9,7,1|6,2,8,0|4|3)
(5,9|7,1|6,2,8,0|4|3)
sage: P.is_discrete()
False
```

num_cells()

Return the number of cells in the finest partition.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: P = PartitionStack([range(9, -1, -1)])
sage: P.set_k(1)
sage: P.sort_by_function(0, [2,1,2,1,2,1,3,4,2,1], 10)
0
sage: P
(1,9,7,5,0,2,8,6,4,3)
(1,9,7,5|0,2,8,6|4|3)
sage: P.num_cells()
4
sage: P.set_k(2)
sage: P.sort_by_function(0, [2,1,2,1], 10)
0
sage: P
(5,9,1,7,0,2,8,6,4,3)
(5,9,1,7|0,2,8,6|4|3)
(5,9|1,7|0,2,8,6|4|3)
sage: P.num_cells()
5
```

percolate()

Perform one round of bubble sort, moving the smallest element to the front.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: P = PartitionStack([range(9, -1, -1)])
sage: P
(0,9,8,7,6,5,4,3,2,1)
sage: P.percolate(2,7)
sage: P
(0,9,3,8,7,6,5,4,2,1)
```

refine()

Implementation of Algorithm 2.5 in [1].

EXAMPLE:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: P = PartitionStack([range(9, -1, -1)])
sage: P.set_k(1)
sage: P.sort_by_function(0, [2,1,2,1,2,1,3,4,2,1], 10)
0
sage: P.set_k(2)
sage: P.sort_by_function(0, [2,1,2,1], 10)
0
sage: P.set_k(3)
sage: P.sort_by_function(4, [2,1,2,1], 10)
4
sage: P.set_k(4)
sage: P.sort_by_function(0, [0,1], 10)
0
sage: P.set_k(5)
sage: P.sort_by_function(2, [1,0], 10)
2
sage: P.set_k(6)
sage: P.sort_by_function(4, [1,0], 10)
4
sage: P.set_k(7)
sage: P.sort_by_function(6, [1,0], 10)
6
sage: P
(5,9,7,1,6,2,8,0,4,3)
(5,9,7,1|6,2,8,0|4|3)
(5,9|7,1|6,2,8,0|4|3)
(5,9|7,1|6,2|8,0|4|3)
(5|9|7,1|6,2|8,0|4|3)
(5|9|7|1|6,2|8,0|4|3)
(5|9|7|1|6|2|8,0|4|3)
(5|9|7|1|6|2|8|0|4|3)
sage: P.is_discrete()
1
sage: P.set_k(6)
sage: P.is_discrete()
0

sage: G = SparseGraph(10)
sage: for i,j,_ in graphs.PetersenGraph().edge_iterator():
...     G.add_arc(i,j)
...     G.add_arc(j,i)
sage: P = PartitionStack(10)
sage: P.set_k(1)
sage: P.split_vertex(0)
sage: P.refine(G, [0], 10, 0, 1)
sage: P
(0,2,3,6,7,8,9,1,4,5)
(0|2,3,6,7,8,9|1,4,5)
sage: P.set_k(2)
sage: P.split_vertex(1)
sage: P.refine(G, [7], 10, 0, 1)
sage: P
```

```
(0,3,7,8,9,2,6,1,4,5)
(0|3,7,8,9,2,6|1,4,5)
(0|3,7,8,9|2,6|1|4,5)
```

repr_at_k()

Return the k-th line of the representation of self, i.e. the k-th partition in the stack.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: P = PartitionStack([range(9, -1, -1)])
sage: P.set_k(1)
sage: P.sort_by_function(0, [2,1,2,1,2,1,3,4,2,1], 10)
0
sage: P.set_k(2)
sage: P.sort_by_function(0, [2,1,2,1], 10)
0
sage: P.set_k(3)
sage: P.sort_by_function(4, [2,1,2,1], 10)
4
sage: P.set_k(4)
sage: P.sort_by_function(0, [0,1], 10)
0
sage: P.set_k(5)
sage: P.sort_by_function(2, [1,0], 10)
2
sage: P.set_k(6)
sage: P.sort_by_function(4, [1,0], 10)
4
sage: P.set_k(7)
sage: P.sort_by_function(6, [1,0], 10)
6
sage: P
(5,9,7,1,6,2,8,0,4,3)
(5,9,7,1|6,2,8,0|4|3)
(5,9|7,1|6,2,8,0|4|3)
(5,9|7,1|6,2|8,0|4|3)
(5|9|7,1|6,2|8,0|4|3)
(5|9|7|1|6,2|8,0|4|3)
(5|9|7|1|6|2|8,0|4|3)
(5|9|7|1|6|2|8|0|4|3)

sage: P.repr_at_k(0)
'(5,9,7,1,6,2,8,0,4,3)'
sage: P.repr_at_k(1)
'(5,9,7,1|6,2,8,0|4|3)'
```

sat_225()

Whether the finest partition satisfies the hypotheses of Lemma 2.25 in [1].

EXAMPLE:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: P = PartitionStack([range(9, -1, -1)])
sage: P
(0,9,8,7,6,5,4,3,2,1)
sage: P.sat_225(10)
False
sage: P.set_k(1)
sage: P.sort_by_function(0, [2,1,2,1,2,1,3,4,2,1], 10)
```

```
0
sage: P
(1,9,7,5,0,2,8,6,4,3)
(1,9,7,5|0,2,8,6|4|3)
sage: P.sat_225(10)
False
sage: P.set_k(2)
sage: P.sort_by_function(0, [2,1,2,1], 10)
0
sage: P
(5,9,1,7,0,2,8,6,4,3)
(5,9,1,7|0,2,8,6|4|3)
(5,9|1,7|0,2,8,6|4|3)
sage: P.sat_225(10)
False
sage: P.set_k(3)
sage: P.sort_by_function(4, [2,1,2,1], 10)
4
sage: P
(5,9,1,7,2,6,0,8,4,3)
(5,9,1,7|2,6,0,8|4|3)
(5,9|1,7|2,6,0,8|4|3)
(5,9|1,7|2,6|0,8|4|3)
sage: P.sat_225(10)
True
```

set_k()

Sets self.k, the index of the finest partition.

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: P = PartitionStack([range(9, -1, -1)])
sage: P.set_k(1)
sage: P.sort_by_function(0, [2,1,2,1,2,1,3,4,2,1], 10)
0
sage: P.set_k(2)
sage: P.sort_by_function(0, [2,1,2,1], 10)
0
sage: P.set_k(3)
sage: P.sort_by_function(4, [2,1,2,1], 10)
4
sage: P.set_k(4)
sage: P.sort_by_function(0, [0,1], 10)
0
sage: P.set_k(5)
sage: P.sort_by_function(2, [1,0], 10)
2
sage: P.set_k(6)
sage: P.sort_by_function(4, [1,0], 10)
4
sage: P.set_k(7)
sage: P.sort_by_function(6, [1,0], 10)
6
sage: P
(5,9,7,1,6,2,8,0,4,3)
(5,9,7,1|6,2,8,0|4|3)
(5,9|7,1|6,2,8,0|4|3)
(5,9|7,1|6,2|8,0|4|3)
(5|9|7,1|6,2|8,0|4|3)
```

```
(5|9|7|1|6,2|8,0|4|3)
(5|9|7|1|6|2|8,0|4|3)
(5|9|7|1|6|2|8|0|4|3)

sage: P.set_k(2)
sage: P
(5,9,7,1,6,2,8,0,4,3)
(5,9,7,1|6,2,8,0|4|3)
(5,9|7,1|6,2,8,0|4|3)
```

sort_by_function()

Sort the cell starting at start using a counting sort, where degrees is the function giving the sort. Result is the cell is subdivided into cells which have elements all of the same 'degree,' in order.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: P = PartitionStack([range(9, -1, -1)])
sage: P.set_k(1)
sage: P
(0,9,8,7,6,5,4,3,2,1)
(0,9,8,7,6,5,4,3,2,1)
sage: P.sort_by_function(0, [2,1,2,1,2,1,3,4,2,1], 10)
0
sage: P
(1,9,7,5,0,2,8,6,4,3)
(1,9,7,5|0,2,8,6|4|3)
```

split_vertex()

Splits the cell in self(k) containing v, putting new cells in place in self(k).

EXAMPLE:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: P = PartitionStack([range(9, -1, -1)])
sage: P
(0,9,8,7,6,5,4,3,2,1)
sage: P.split_vertex(2)
sage: P
(2|0,9,8,7,6,5,4,3,1)
```

all_labeled_digraphs()

EXAMPLES:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import search_tree
sage: from sage.graphs.graph_isom import all_labeled_digraphs
sage: Glist = {}
sage: Giso  = {}
sage: for n in range(1,4):
...       Glist[n] = all_labeled_digraphs(n)
...       Giso[n] = []
...       for g in Glist[n]:
...           a, b = search_tree(g, [range(n)], dig=True)
...           inn = False
...           for gi in Giso[n]:
...               if b == gi:
...                   inn = True
...           if not inn:
...               Giso[n].append(b)
sage: for n in Giso:
```

```
...       print n, len(Giso[n])
1 1
2 3
3 16
```

all_labeled_digraphs_with_loops()

Returns all labeled digraphs on n vertices 0,1,...,n-1. Used in classifying isomorphism types (naive approach), and more importantly in benchmarking the search algorithm.

EXAMPLE:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import search_tree
sage: from sage.graphs.graph_isom import all_labeled_digraphs_with_loops
sage: Glist = {}
sage: Giso  = {}
sage: for n in range(1,4):
...     Glist[n] = all_labeled_digraphs_with_loops(n)
...     Giso[n] = []
...     for g in Glist[n]:
...         a, b = search_tree(g, [range(n)], dig=True)
...         inn = False
...         for gi in Giso[n]:
...             if b == gi:
...                 inn = True
...         if not inn:
...             Giso[n].append(b)
sage: for n in Giso:
...     print n, len(Giso[n])
1 2
2 10
3 104
```

all_labeled_graphs()

Returns all labeled graphs on n vertices 0,1,...,n-1. Used in classifying isomorphism types (naive approach), and more importantly in benchmarking the search algorithm.

EXAMPLE:

```
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import search_tree
sage: from sage.graphs.graph_isom import all_labeled_graphs
sage: Glist = {}
sage: Giso  = {}
sage: for n in range(1,5):
...     Glist[n] = all_labeled_graphs(n)
...     Giso[n] = []
...     for g in Glist[n]:
...         a, b = search_tree(g, [range(n)])
...         inn = False
...         for gi in Giso[n]:
...             if b == gi:
...                 inn = True
...         if not inn:
...             Giso[n].append(b)
sage: for n in Giso:
...     print n, len(Giso[n])
1 1
2 2
3 4
4 11
```

```
sage: n = 5
sage: Glist[n] = all_labeled_graphs(n)
sage: Giso[n] = []
sage: for g in Glist[5]:
...     a, b = search_tree(g, [range(n)])
...     inn = False
...     for gi in Giso[n]:
...         if b == gi:
...             inn = True
...     if not inn:
...         Giso[n].append(b)
sage: print n, len(Giso[n]) # long time
5 34
sage: graphs_list.show_graphs(Giso[4])
```

all_ordered_partitions()

Returns all ordered partitions of the set 0,1,...,n-1. Used in benchmarking the search algorithm.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import all_ordered_partitions
sage: all_ordered_partitions(['a', 1, {}])
[[['a'], [1], [{}]],
 [['a'], [{}], [1]],
 [['a'], [{}, 1]],
 [['a'], [1, {}]],
 [[1], ['a'], [{}]],
 [[1], [{}], ['a']],
 [[1], [{}, 'a']],
 [[1], ['a', {}]],
 [[{}], ['a'], [1]],
 [[{}], [1], ['a']],
 [[{}], [1, 'a']],
 [[{}], ['a', 1]],
 [[1, 'a'], [{}]],
 [[{}, 'a'], [1]],
 [['a', 1], [{}]],
 [[{}, 1], ['a']],
 [['a', {}], [1]],
 [[1, {}], ['a']],
 [[{}, 1, 'a']],
 [[1, {}, 'a']],
 [[{}, 'a', 1]],
 [['a', {}, 1]],
 [[1, 'a', {}]],
 [['a', 1, {}]]]
```

kpow()

Returns the subset of the power set of listy consisting of subsets of size k. Used in all_ordered_partitions.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import kpow
sage: kpow(['a', 1, {}], 2)
[[1, 'a'], [{}, 'a'], ['a', 1], [{}, 1], ['a', {}], [1, {}]]
```

orbit_partition()

Assuming that G is a graph on vertices 0,1,...,n-1, and gamma is an element of SymmetricGroup(n), returns the

partition of the vertex set determined by the orbits of gamma, considered as action on the set 1,2,...,n where we take 0 = n. In other words, returns the partition determined by a cyclic representation of gamma.

INPUT:

- `list_perm` - if True, assumes `gamma` is a list representing the map $i \mapsto$ "$gamma$"$[i]$.

EXAMPLES:

```
sage: from sage.graphs.graph_isom import orbit_partition
sage: G = graphs.PetersenGraph()
sage: S = SymmetricGroup(10)
sage: gamma = S('(10,1,2,3,4)(5,6,7)(8,9)')
sage: orbit_partition(gamma)
[[1, 2, 3, 4, 0], [5, 6, 7], [8, 9]]
sage: gamma = S('(10,5)(1,6)(2,7)(3,8)(4,9)')
sage: orbit_partition(gamma)
[[1, 6], [2, 7], [3, 8], [4, 9], [5, 0]]
```

perm_group_elt()

Given a list permutation of the set 0, 1, ..., n-1, returns the corresponding PermutationGroupElement where we take 0 = n.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import perm_group_elt
sage: perm_group_elt([0,2,1])
(1,2)
sage: perm_group_elt([1,2,0])
(1,2,3)
```

search_tree()

Assumes that the vertex set of G is 0,1,...,n-1 for some n.

Note that this conflicts with the SymmetricGroup we are using to represent automorphisms. The solution is to let the group act on the set 1,2,...,n, under the assumption n = 0.

INPUT: lab- if True, return the canonical label in addition to the auto- morphism group. dig- if True, does not use Lemma 2.25 in [1], and the algorithm is valid for digraphs and graphs with loops. dict_rep- if True, explain which vertices are which elements of the set 1,2,...,n in the representation of the automorphism group. certify- if True, return the relabeling from G to its canonical label. Forces lab=True. verbosity- 0 - print nothing 1 - display state trace 2 - with timings 3 - display partition nests 4 - display orbit partition 5 - plot the part of the tree traversed during search

- `use_indicator_function` - option to turn off indicator function (False - slower)
- `sparse` - whether to use sparse or dense representation of the graph (ignored if G is already a CGraph - see sage.graphs.base)
- `base` - whether to return the first sequence of split vertices (used in computing the order of the group)
- `order` - whether to return the order of the automorphism group

STATE DIAGRAM:

```
sage: SD = DiGraph( { 1:[18,2], 2:[5,3], 3:[4,6], 4:[7,2], 5:[4], 6:[13,12], 7:[18,8,10], 8:[6,9
sage: SD.set_edge_label(1, 18, 'discrete')
sage: SD.set_edge_label(4, 7, 'discrete')
sage: SD.set_edge_label(2, 5, 'h = 0')
sage: SD.set_edge_label(7, 18, 'h = 0')
sage: SD.set_edge_label(7, 10, 'aut')
sage: SD.set_edge_label(8, 10, 'aut')
```

```
sage: SD.set_edge_label(8, 9, 'label')
sage: SD.set_edge_label(8, 6, 'no label')
sage: SD.set_edge_label(13, 17, 'k > h')
sage: SD.set_edge_label(13, 14, 'k = h')
sage: SD.set_edge_label(17, 15, 'v_k finite')
sage: SD.set_edge_label(14, 15, 'v_k m.c.r.')
sage: posn = {1:[ 3,-3],  2:[0,2],  3:[0, 13],  4:[3,9],  5:[3,3],  6:[16, 13], 7:[6,1],  8:[6,6
sage: SD.plot(pos=posn, vertex_size=400, vertex_colors={'#FFFFFF':range(1,19)}, edge_labels=True
```

Note: There is a function, called test_refine, that has the same signature as _refine. It calls _refine, then checks to make sure the output is sane. To use this, simply add 'test' to the two places this algorithm calls the function (states 1 and 2).

EXAMPLES: The following example is due to Chris Godsil:

```
sage: HS = graphs.HoffmanSingletonGraph()
sage: clqs = (HS.complement()).cliques()
sage: alqs = [Set(c) for c in clqs if len(c) == 15]
sage: Y = Graph([alqs, lambda s,t: len(s.intersection(t))==0], implementation='networkx')
sage: Y0,Y1 = Y.connected_components_subgraphs()
sage: Y0.is_isomorphic(Y1)
True
sage: Y0.is_isomorphic(HS)
True
```

```
sage: from sage.groups.perm_gps.partn_ref.refinement_graphs import search_tree
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: from sage.graphs.base.dense_graph import DenseGraph
sage: from sage.groups.perm_gps.permgroup import PermutationGroup
sage: from sage.graphs.graph_isom import perm_group_elt
```

```
sage: G = graphs.DodecahedralGraph()
sage: GD = DenseGraph(20)
sage: GS = SparseGraph(20)
sage: for i,j,_ in G.edge_iterator():
...     GD.add_arc(i,j); GD.add_arc(j,i)
...     GS.add_arc(i,j); GS.add_arc(j,i)
sage: Pi=[range(20)]
sage: a,b = search_tree(G, Pi)
sage: asp,bsp = search_tree(GS, Pi)
sage: ade,bde = search_tree(GD, Pi)
sage: bsg = Graph(implementation='networkx')
sage: bdg = Graph(implementation='networkx')
sage: for i in range(20):
...     for j in range(20):
...         if bsp.has_arc(i,j):
...             bsg.add_edge(i,j)
...         if bde.has_arc(i,j):
...             bdg.add_edge(i,j)
sage: print a, b.graph6_string()
[[0, 19, 3, 2, 6, 5, 4, 17, 18, 11, 10, 9, 13, 12, 16, 15, 14, 7, 8, 1], [0, 1, 8, 9, 13, 14, 7,
sage: a == asp
True
sage: a == ade
True
sage: b == bsg
True
```

```
sage: b == bdg
True
sage: c = search_tree(G, Pi, lab=False)
sage: print c
[[0, 19, 3, 2, 6, 5, 4, 17, 18, 11, 10, 9, 13, 12, 16, 15, 14, 7, 8, 1], [0, 1, 8, 9, 13, 14, 7,
sage: DodecAut = PermutationGroup([perm_group_elt(aa) for aa in a])
sage: DodecAut.character_table()
[                      1                       1                       1                       1
[                      1                      -1                       1                       1
[                      3                      -1                       0                      -1
[                      3                      -1                       0                      -1  ze
[                      3                       1                       0                      -1
[                      3                       1                       0                      -1 -ze
[                      4                       0                       1                       0
[                      4                       0                       1                       0
[                      5                       1                      -1                       1
[                      5                      -1                      -1                       1

sage: DodecAut2 = PermutationGroup([perm_group_elt(cc) for cc in c])
sage: DodecAut2.character_table()
[                      1                       1                       1                       1
[                      1                      -1                       1                       1
[                      3                      -1                       0                      -1
[                      3                      -1                       0                      -1  ze
[                      3                       1                       0                      -1
[                      3                       1                       0                      -1 -ze
[                      4                       0                       1                       0
[                      4                       0                       1                       0
[                      5                       1                      -1                       1
[                      5                      -1                      -1                       1


sage: G = graphs.PetersenGraph()
sage: Pi=[range(10)]
sage: a,b = search_tree(G, Pi)
sage: print a, b.graph6_string()
[[0, 1, 2, 7, 5, 4, 6, 3, 9, 8], [0, 1, 6, 8, 5, 4, 2, 9, 3, 7], [0, 4, 3, 8, 5, 1, 9, 2, 6, 7],
sage: c = search_tree(G, Pi, lab=False)
sage: PAut = PermutationGroup([perm_group_elt(aa) for aa in a])
sage: PAut.character_table()
[ 1  1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 4  2  0 -1  1  0 -1]
[ 5  1  1  1 -1 -1  0]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
sage: PAut = PermutationGroup([perm_group_elt(cc) for cc in c])
sage: PAut.character_table()
[ 1  1  1  1  1  1  1]
[ 1 -1  1 -1  1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 4  2  0 -1  1  0 -1]
[ 5  1  1  1 -1 -1  0]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
```

```
sage: G = graphs.CubeGraph(3)
sage: Pi = []
sage: for i in range(8):
...     b = Integer(i).binary()
...     Pi.append(b.zfill(3))
...
sage: Pi = [Pi]
sage: a,b = search_tree(G, Pi)
sage: print a, b.graph6_string()
[[0, 2, 1, 3, 4, 6, 5, 7], [0, 1, 4, 5, 2, 3, 6, 7], [1, 0, 3, 2, 5, 4, 7, 6]] GIQ\T_
sage: c = search_tree(G, Pi, lab=False)


sage: PermutationGroup([perm_group_elt(aa) for aa in a]).order()
48
sage: PermutationGroup([perm_group_elt(cc) for cc in c]).order()
48
sage: DodecAut.order()
120
sage: PAut.order()
120


sage: D = graphs.DodecahedralGraph()
sage: a,b,c = search_tree(D, [range(20)], certify=True)
sage: from sage.plot.plot import GraphicsArray
sage: from sage.graphs.graph_fast import spring_layout_fast
sage: position_D = spring_layout_fast(D)
sage: position_b = {}
sage: for vert in position_D:
...     position_b[c[vert]] = position_D[vert]
sage: graphics_array([D.plot(pos=position_D), b.plot(pos=position_b)]).show()
sage: c
{0: 0, 1: 19, 2: 16, 3: 15, 4: 9, 5: 1, 6: 10, 7: 8, 8: 14, 9: 12, 10: 17, 11: 11, 12: 5, 13: 6,
```

BENCHMARKS: The following examples are given to check modifications to the algorithm for optimization.

```
sage: G = Graph({0:[]})
sage: Pi = [[0]]
sage: a,b = search_tree(G, Pi)
sage: print a, b.graph6_string()
[] @
sage: a,b = search_tree(G, Pi, dig=True)
sage: print a, b.graph6_string()
[] @
sage: search_tree(G, Pi, lab=False)
[]


sage: from sage.graphs.graph_isom import all_labeled_graphs, all_ordered_partitions


sage: graph2 = all_labeled_graphs(2)
sage: part2 = all_ordered_partitions(range(2))
sage: for G in graph2:
...     for Pi in part2:
...         a,b = search_tree(G, Pi)
...         c,d = search_tree(G, Pi, dig=True)
...         e = search_tree(G, Pi, lab=False)
...         a = str(a); b = b.graph6_string(); c = str(c); d = d.graph6_string(); e = str(e)
```

```
...         print a.ljust(15), b.ljust(5), c.ljust(15), d.ljust(5), e.ljust(15)
[]              A?    []              A?    []
[]              A?    []              A?    []
[[1, 0]]        A?    [[1, 0]]        A?    [[1, 0]]
[[1, 0]]        A?    [[1, 0]]        A?    [[1, 0]]
[]              A_    []              A_    []
[]              A_    []              A_    []
[[1, 0]]        A_    [[1, 0]]        A_    [[1, 0]]
[[1, 0]]        A_    [[1, 0]]        A_    [[1, 0]]

sage: graph3 = all_labeled_graphs(3)
sage: part3 = all_ordered_partitions(range(3))
sage: for G in graph3:
...    for Pi in part3:
...        a,b = search_tree(G, Pi)
...        c,d = search_tree(G, Pi, dig=True)
...        e = search_tree(G, Pi, lab=False)
...        a = str(a); b = b.graph6_string(); c = str(c); d = d.graph6_string(); e = str(e)
...        print a.ljust(15), b.ljust(5), c.ljust(15), d.ljust(5), e.ljust(15)
[]              B?    []              B?    []
[]              B?    []              B?    []
[[0, 2, 1]]     B?    [[0, 2, 1]]     B?    [[0, 2, 1]]
[[0, 2, 1]]     B?    [[0, 2, 1]]     B?    [[0, 2, 1]]
[]              B?    []              B?    []
[]              B?    []              B?    []
[[2, 1, 0]]     B?    [[2, 1, 0]]     B?    [[2, 1, 0]]
[[2, 1, 0]]     B?    [[2, 1, 0]]     B?    [[2, 1, 0]]
[]              B?    []              B?    []
[]              B?    []              B?    []
[[1, 0, 2]]     B?    [[1, 0, 2]]     B?    [[1, 0, 2]]
[[1, 0, 2]]     B?    [[1, 0, 2]]     B?    [[1, 0, 2]]
[[1, 0, 2]]     B?    [[1, 0, 2]]     B?    [[1, 0, 2]]
[[2, 1, 0]]     B?    [[2, 1, 0]]     B?    [[2, 1, 0]]
[[1, 0, 2]]     B?    [[1, 0, 2]]     B?    [[1, 0, 2]]
[[0, 2, 1]]     B?    [[0, 2, 1]]     B?    [[0, 2, 1]]
[[2, 1, 0]]     B?    [[2, 1, 0]]     B?    [[2, 1, 0]]
[[0, 2, 1]]     B?    [[0, 2, 1]]     B?    [[0, 2, 1]]
[[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]]
[[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]]
[[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]]
[[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]]
[[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]]
[[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]] B?    [[0, 2, 1], [1, 0, 2]]
[]              BG    []              BG    []
[]              BG    []              BG    []
[[0, 2, 1]]     BG    [[0, 2, 1]]     BG    [[0, 2, 1]]
[[0, 2, 1]]     BG    [[0, 2, 1]]     BG    [[0, 2, 1]]
[]              BO    []              BO    []
[]              B_    []              B_    []
[]              BO    []              BO    []
[]              BO    []              BO    []
[]              BO    []              BO    []
[]              B_    []              B_    []
[]              BO    []              BO    []
[]              BO    []              BO    []
[]              BG    []              BG    []
[]              BG    []              BG    []
```

```
[]              BG    []              BG    []
[[0, 2, 1]]     B_    [[0, 2, 1]]     B_    [[0, 2, 1]]
[]              BG    []              BG    []
[[0, 2, 1]]     B_    [[0, 2, 1]]     B_    [[0, 2, 1]]
[[0, 2, 1]]     BG    [[0, 2, 1]]     BG    [[0, 2, 1]]
[[0, 2, 1]]     BG    [[0, 2, 1]]     BG    [[0, 2, 1]]
[[0, 2, 1]]     BG    [[0, 2, 1]]     BG    [[0, 2, 1]]
[[0, 2, 1]]     BG    [[0, 2, 1]]     BG    [[0, 2, 1]]
[[0, 2, 1]]     BG    [[0, 2, 1]]     BG    [[0, 2, 1]]
[]              BO    []              BO    []
[]              B_    []              B_    []
[]              BO    []              BO    []
[]              BO    []              BO    []
[]              BG    []              BG    []
[]              BG    []              BG    []
[[2, 1, 0]]     BG    [[2, 1, 0]]     BG    [[2, 1, 0]]
[[2, 1, 0]]     BG    [[2, 1, 0]]     BG    [[2, 1, 0]]
[]              B_    []              B_    []
[]              BO    []              BO    []
[]              BO    []              BO    []
[]              BO    []              BO    []
[]              BG    []              BG    []
[[2, 1, 0]]     B_    [[2, 1, 0]]     B_    [[2, 1, 0]]
[]              BG    []              BG    []
[]              BG    []              BG    []
[[2, 1, 0]]     B_    [[2, 1, 0]]     B_    [[2, 1, 0]]
[]              BG    []              BG    []
[[2, 1, 0]]     BG    [[2, 1, 0]]     BG    [[2, 1, 0]]
[[2, 1, 0]]     BG    [[2, 1, 0]]     BG    [[2, 1, 0]]
[[2, 1, 0]]     BG    [[2, 1, 0]]     BG    [[2, 1, 0]]
[[2, 1, 0]]     BG    [[2, 1, 0]]     BG    [[2, 1, 0]]
[[2, 1, 0]]     BG    [[2, 1, 0]]     BG    [[2, 1, 0]]
[]              BW    []              BW    []
[]              Bg    []              Bg    []
[]              BW    []              BW    []
[]              BW    []              BW    []
[]              BW    []              BW    []
[]              Bg    []              Bg    []
[]              BW    []              BW    []
[]              BW    []              BW    []
[]              Bo    []              Bo    []
[]              Bo    []              Bo    []
[[1, 0, 2]]     Bo    [[1, 0, 2]]     Bo    [[1, 0, 2]]
[[1, 0, 2]]     Bo    [[1, 0, 2]]     Bo    [[1, 0, 2]]
[[1, 0, 2]]     BW    [[1, 0, 2]]     BW    [[1, 0, 2]]
[]              Bg    []              Bg    []
[[1, 0, 2]]     BW    [[1, 0, 2]]     BW    [[1, 0, 2]]
[]              Bg    []              Bg    []
[]              Bg    []              Bg    []
[]              Bg    []              Bg    []
[[1, 0, 2]]     BW    [[1, 0, 2]]     BW    [[1, 0, 2]]
[[1, 0, 2]]     BW    [[1, 0, 2]]     BW    [[1, 0, 2]]
[[1, 0, 2]]     BW    [[1, 0, 2]]     BW    [[1, 0, 2]]
[[1, 0, 2]]     BW    [[1, 0, 2]]     BW    [[1, 0, 2]]
[[1, 0, 2]]     BW    [[1, 0, 2]]     BW    [[1, 0, 2]]
[[1, 0, 2]]     BW    [[1, 0, 2]]     BW    [[1, 0, 2]]
```

```
[]              B_    []              B_    []
[]              BO    []              BO    []
[]              BO    []              BO    []
[]              BO    []              BO    []
[]              B_    []              B_    []
[]              BO    []              BO    []
[]              BO    []              BO    []
[]              BO    []              BO    []
[]              BG    []              BG    []
[]              BG    []              BG    []
[[1, 0, 2]]     BG    [[1, 0, 2]]     BG    [[1, 0, 2]]
[[1, 0, 2]]     BG    [[1, 0, 2]]     BG    [[1, 0, 2]]
[[1, 0, 2]]     B_    [[1, 0, 2]]     B_    [[1, 0, 2]]
[]              BG    []              BG    []
[[1, 0, 2]]     B_    [[1, 0, 2]]     B_    [[1, 0, 2]]
[]              BG    []              BG    []
[]              BG    []              BG    []
[]              BG    []              BG    []
[[1, 0, 2]]     BG    [[1, 0, 2]]     BG    [[1, 0, 2]]
[[1, 0, 2]]     BG    [[1, 0, 2]]     BG    [[1, 0, 2]]
[[1, 0, 2]]     BG    [[1, 0, 2]]     BG    [[1, 0, 2]]
[[1, 0, 2]]     BG    [[1, 0, 2]]     BG    [[1, 0, 2]]
[[1, 0, 2]]     BG    [[1, 0, 2]]     BG    [[1, 0, 2]]
[[1, 0, 2]]     BG    [[1, 0, 2]]     BG    [[1, 0, 2]]
[]              Bg    []              Bg    []
[]              BW    []              BW    []
[]              BW    []              BW    []
[]              BW    []              BW    []
[]              Bo    []              Bo    []
[]              Bo    []              Bo    []
[[2, 1, 0]]     Bo    [[2, 1, 0]]     Bo    [[2, 1, 0]]
[[2, 1, 0]]     Bo    [[2, 1, 0]]     Bo    [[2, 1, 0]]
[]              BW    []              BW    []
[]              Bg    []              Bg    []
[]              BW    []              BW    []
[]              BW    []              BW    []
[]              Bg    []              Bg    []
[[2, 1, 0]]     BW    [[2, 1, 0]]     BW    [[2, 1, 0]]
[]              Bg    []              Bg    []
[]              Bg    []              Bg    []
[[2, 1, 0]]     BW    [[2, 1, 0]]     BW    [[2, 1, 0]]
[]              Bg    []              Bg    []
[[2, 1, 0]]     BW    [[2, 1, 0]]     BW    [[2, 1, 0]]
[[2, 1, 0]]     BW    [[2, 1, 0]]     BW    [[2, 1, 0]]
[[2, 1, 0]]     BW    [[2, 1, 0]]     BW    [[2, 1, 0]]
[[2, 1, 0]]     BW    [[2, 1, 0]]     BW    [[2, 1, 0]]
[[2, 1, 0]]     BW    [[2, 1, 0]]     BW    [[2, 1, 0]]
[[2, 1, 0]]     BW    [[2, 1, 0]]     BW    [[2, 1, 0]]
[]              Bo    []              Bo    []
[]              Bo    []              Bo    []
[[0, 2, 1]]     Bo    [[0, 2, 1]]     Bo    [[0, 2, 1]]
[[0, 2, 1]]     Bo    [[0, 2, 1]]     Bo    [[0, 2, 1]]
[]              Bg    []              Bg    []
[]              BW    []              BW    []
[]              BW    []              BW    []
[]              BW    []              BW    []
[]              Bg    []              Bg    []
[]              BW    []              BW    []
```

```
[]              BW    []              BW    []
[]              BW    []              BW    []
[]              Bg    []              Bg    []
[]              Bg    []              Bg    []
[]              Bg    []              Bg    []
[[0, 2, 1]]     BW    [[0, 2, 1]]     BW    [[0, 2, 1]]
[]              Bg    []              Bg    []
[[0, 2, 1]]     BW    [[0, 2, 1]]     BW    [[0, 2, 1]]
[[0, 2, 1]]     BW    [[0, 2, 1]]     BW    [[0, 2, 1]]
[[0, 2, 1]]     BW    [[0, 2, 1]]     BW    [[0, 2, 1]]
[[0, 2, 1]]     BW    [[0, 2, 1]]     BW    [[0, 2, 1]]
[[0, 2, 1]]     BW    [[0, 2, 1]]     BW    [[0, 2, 1]]
[[0, 2, 1]]     BW    [[0, 2, 1]]     BW    [[0, 2, 1]]
[[0, 2, 1]]     BW    [[0, 2, 1]]     BW    [[0, 2, 1]]
[]              Bw    []              Bw    []
[]              Bw    []              Bw    []
[[0, 2, 1]]     Bw    [[0, 2, 1]]     Bw    [[0, 2, 1]]
[[0, 2, 1]]     Bw    [[0, 2, 1]]     Bw    [[0, 2, 1]]
[]              Bw    []              Bw    []
[]              Bw    []              Bw    []
[[2, 1, 0]]     Bw    [[2, 1, 0]]     Bw    [[2, 1, 0]]
[[2, 1, 0]]     Bw    [[2, 1, 0]]     Bw    [[2, 1, 0]]
[]              Bw    []              Bw    []
[]              Bw    []              Bw    []
[[1, 0, 2]]     Bw    [[1, 0, 2]]     Bw    [[1, 0, 2]]
[[1, 0, 2]]     Bw    [[1, 0, 2]]     Bw    [[1, 0, 2]]
[[1, 0, 2]]     Bw    [[1, 0, 2]]     Bw    [[1, 0, 2]]
[[2, 1, 0]]     Bw    [[2, 1, 0]]     Bw    [[2, 1, 0]]
[[1, 0, 2]]     Bw    [[1, 0, 2]]     Bw    [[1, 0, 2]]
[[0, 2, 1]]     Bw    [[0, 2, 1]]     Bw    [[0, 2, 1]]
[[2, 1, 0]]     Bw    [[2, 1, 0]]     Bw    [[2, 1, 0]]
[[0, 2, 1]]     Bw    [[0, 2, 1]]     Bw    [[0, 2, 1]]
[[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]]
[[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]]
[[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]]
[[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]]
[[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]]
[[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]] Bw    [[0, 2, 1], [1, 0, 2]]

sage: C = graphs.CubeGraph(1)
sage: gens = search_tree(C, [C.vertices()], lab=False)
sage: PermutationGroup([perm_group_elt(aa) for aa in gens]).order()
2
sage: C = graphs.CubeGraph(2)
sage: gens = search_tree(C, [C.vertices()], lab=False)
sage: PermutationGroup([perm_group_elt(aa) for aa in gens]).order()
8
sage: C = graphs.CubeGraph(3)
sage: gens = search_tree(C, [C.vertices()], lab=False)
sage: PermutationGroup([perm_group_elt(aa) for aa in gens]).order()
48
sage: C = graphs.CubeGraph(4)
sage: gens = search_tree(C, [C.vertices()], lab=False)
sage: PermutationGroup([perm_group_elt(aa) for aa in gens]).order()
384
sage: C = graphs.CubeGraph(5)
sage: gens = search_tree(C, [C.vertices()], lab=False)
```

```
sage: PermutationGroup([perm_group_elt(aa) for aa in gens]).order()
3840
sage: C = graphs.CubeGraph(6)
sage: gens = search_tree(C, [C.vertices()], lab=False)
sage: PermutationGroup([perm_group_elt(aa) for aa in gens]).order()
46080
```

One can also turn off the indicator function (note- this will take longer)

```
sage: D1 = DiGraph({0:[2],2:[0],1:[1]}, loops=True)
sage: D2 = DiGraph({1:[2],2:[1],0:[0]}, loops=True)
sage: a,b = search_tree(D1, [D1.vertices()], use_indicator_function=False)
sage: c,d = search_tree(D2, [D2.vertices()], use_indicator_function=False)
sage: b==d
True
```

Previously a bug, now the output is correct:

```
sage: G = Graph('^?????????????????????{??N??@w??FaGa?PCO@CP?AGa?_QO?Q@G?CcA??cc????Bo????{????F_
sage: perm = {3:15, 15:3}
sage: H = G.relabel(perm, inplace=False)
sage: G.canonical_label() == H.canonical_label()
True
```

Another former bug:

```
sage: Graph('Fll^G').canonical_label()
Graph on 7 vertices
```

```
sage: g = Graph(21)
sage: g.automorphism_group(return_group=False, order=True)
51090942171709440000
```

verify_partition_refinement()

Verify that the refinement is correct.

EXAMPLE:

```
sage: from sage.graphs.graph_isom import PartitionStack
sage: from sage.graphs.base.sparse_graph import SparseGraph
sage: G = SparseGraph(10)
sage: for i,j,_ in graphs.PetersenGraph().edge_iterator():
...     G.add_arc(i,j)
...     G.add_arc(j,i)
sage: P = PartitionStack(10)
sage: P.set_k(1)
sage: P.split_vertex(0)
sage: P.refine(G, [0], 10, 0, 1)
sage: P
(0,2,3,6,7,8,9,1,4,5)
(0|2,3,6,7,8,9|1,4,5)
sage: P.set_k(2)
sage: P.split_vertex(1)
```

Note that this line implicitly tests the function verify_partition_refinement:

```
sage: P.refine(G, [7], 10, 0, 1, test=True)
sage: P
(0,3,7,8,9,2,6,1,4,5)
(0|3,7,8,9,2,6|1,4,5)
(0|3,7,8,9|2,6|1|4,5)
```

7.4 Graph Database Module

INFO:

This module implements classes (GraphDatabase, GraphQuery, GenericGraphQuery) for interfacing with the sqlite database graphs.db.

The GraphDatabase class interfaces with the sqlite database graphs.db. It is an immutable database that inherits from SQLDatabase (see sage.databases.database.py).

The database contains all unlabeled graphs with 7 or fewer nodes. This class will also interface with the optional database package containing all unlabeled graphs with 8 or fewer nodes. The database(s) consists of five tables, and has the structure given by the function graph_info. (For a full description including column data types, create a GraphDatabase instance and call the method get_skeleton).

AUTHORS:

- Emily A. Kirkman (2008-09-20): first version of interactive queries, cleaned up code and generalized many elements to sage.databases.database.py
- Emily A. Kirkman (2007-07-23): inherits GenericSQLDatabase, also added classes: GraphQuery and GenericGraphQuery
- Emily A. Kirkman (2007-05-11): initial sqlite version
- Emily A. Kirkman (2007-02-13): initial version (non-sqlite)

REFERENCES:

- Data provided by Jason Grout (Brigham Young University). [Online] Available: http://math.byu.edu/~grout/graphs/

class GenericGraphQuery(*query_string, database=None, param_tuple=None*)

class GraphDatabase()

interactive_query(*display_cols, **kwds*)

TODO: This function could use improvement. Add full options of typical GraphQuery (i.e.: have it accept list input); and update options in interact to make it less annoying to put in operators.

Generates an interact shell (in the notebook only) that allows the user to manipulate query parameters and see the updated results.

EXAMPLE:

```
sage: D = GraphDatabase()
sage: D.interactive_query(display_cols=['graph6','num_vertices','degree_sequence'],num_edges
<html>...</html>
```

query(*query_dict=None, display_cols=None, **kwds*)

Creates a GraphQuery on this database. For full class details, type GraphQuery? and press shift+enter.

EXAMPLE:

```
sage: D = GraphDatabase()
sage: q = D.query(display_cols=['graph6','num_vertices','degree_sequence'],num_edges=['<=',5
sage: q.show()
Graph6                Num Vertices          Degree Sequence
-----------------------------------------------------------
@                     1                     [0]
A?                    2                     [0, 0]
B?                    3                     [0, 0, 0]
C?                    4                     [0, 0, 0, 0]
D??                   5                     [0, 0, 0, 0, 0]
E???                  6                     [0, 0, 0, 0, 0, 0]
F????                 7                     [0, 0, 0, 0, 0, 0, 0]
A_                    2                     [1, 1]
BG                    3                     [0, 1, 1]
C@                    4                     [0, 0, 1, 1]
D?C                   5                     [0, 0, 0, 1, 1]
E??G                  6                     [0, 0, 0, 0, 1, 1]
F???G                 7                     [0, 0, 0, 0, 0, 1, 1]
BW                    3                     [1, 1, 2]
CB                    4                     [0, 1, 1, 2]
CK                    4                     [1, 1, 1, 1]
D?K                   5                     [0, 0, 1, 1, 2]
D@O                   5                     [0, 1, 1, 1, 1]
E??W                  6                     [0, 0, 0, 1, 1, 2]
E?C_                  6                     [0, 0, 1, 1, 1, 1]
F???W                 7                     [0, 0, 0, 0, 1, 1, 2]
F??G_                 7                     [0, 0, 0, 1, 1, 1, 1]
Bw                    3                     [2, 2, 2]
CF                    4                     [1, 1, 1, 3]
CJ                    4                     [0, 2, 2, 2]
CL                    4                     [1, 1, 2, 2]
D?[                   5                     [0, 1, 1, 1, 3]
D@K                   5                     [0, 0, 2, 2, 2]
D_K                   5                     [1, 1, 1, 1, 2]
D@S                   5                     [0, 1, 1, 2, 2]
E??w                  6                     [0, 0, 1, 1, 1, 3]
E?CW                  6                     [0, 0, 0, 2, 2, 2]
EG?W                  6                     [0, 1, 1, 1, 1, 2]
E?Cg                  6                     [0, 0, 1, 1, 2, 2]
E@Q?                  6                     [1, 1, 1, 1, 1, 1]
F???w                 7                     [0, 0, 0, 1, 1, 1, 3]
F??GW                 7                     [0, 0, 0, 0, 2, 2, 2]
F@??W                 7                     [0, 0, 1, 1, 1, 1, 2]
F??Gg                 7                     [0, 0, 0, 1, 1, 2, 2]
F?Ca?                 7                     [0, 1, 1, 1, 1, 1, 1]
CN                    4                     [1, 2, 2, 3]
C]                    4                     [2, 2, 2, 2]
D?{                   5                     [1, 1, 1, 1, 4]
D@[                   5                     [0, 1, 2, 2, 3]
D@s                   5                     [1, 1, 1, 2, 3]
DBg                   5                     [1, 1, 2, 2, 2]
DBW                   5                     [0, 2, 2, 2, 2]
D`K                   5                     [1, 1, 2, 2, 2]
E?@w                  6                     [0, 1, 1, 1, 1, 4]
E?Cw                  6                     [0, 0, 1, 2, 2, 3]
E?Dg                  6                     [0, 1, 1, 1, 2, 3]
E_?w                  6                     [1, 1, 1, 1, 1, 3]
E?LO                  6                     [0, 1, 1, 2, 2, 2]
```

```
E?N?                6                [1, 1, 1, 1, 2, 2]
E?Ko                6                [0, 0, 2, 2, 2, 2]
EGCW                6                [0, 1, 1, 2, 2, 2]
E_Cg                6                [1, 1, 1, 1, 2, 2]
F??@w               7                [0, 0, 1, 1, 1, 1, 4]
F??Gw               7                [0, 0, 0, 1, 2, 2, 3]
F??Hg               7                [0, 0, 1, 1, 1, 2, 3]
FG??w               7                [0, 1, 1, 1, 1, 1, 3]
F??XO               7                [0, 0, 1, 1, 2, 2, 2]
F??Z?               7                [0, 1, 1, 1, 1, 2, 2]
F??Wo               7                [0, 0, 0, 2, 2, 2, 2]
F@?GW               7                [0, 0, 1, 1, 2, 2, 2]
FK??W               7                [1, 1, 1, 1, 1, 1, 2]
FG?Gg               7                [0, 1, 1, 1, 1, 2, 2]
C^                  4                [2, 2, 3, 3]
D@{                 5                [1, 1, 2, 2, 4]
DB[                 5                [0, 2, 2, 3, 3]
DIk                 5                [1, 2, 2, 2, 3]
DBk                 5                [1, 1, 2, 3, 3]
DK[                 5                [1, 2, 2, 2, 3]
DLo                 5                [2, 2, 2, 2, 2]
E?Bw                6                [1, 1, 1, 1, 1, 5]
E?Dw                6                [0, 1, 1, 2, 2, 4]
E?Fg                6                [1, 1, 1, 1, 2, 4]
E?Kw                6                [0, 0, 2, 2, 3, 3]
E@HW                6                [0, 1, 2, 2, 2, 3]
E@FG                6                [1, 1, 1, 2, 2, 3]
E?LW                6                [0, 1, 1, 2, 3, 3]
E?NG                6                [1, 1, 1, 1, 3, 3]
E@N?                6                [1, 1, 2, 2, 2, 2]
E@YO                6                [1, 1, 2, 2, 2, 2]
E@QW                6                [1, 1, 1, 2, 2, 3]
E@Ow                6                [0, 1, 2, 2, 2, 3]
E_Cw                6                [1, 1, 1, 2, 2, 3]
E@T_                6                [0, 2, 2, 2, 2, 2]
E_Ko                6                [1, 1, 2, 2, 2, 2]
F??Bw               7                [0, 1, 1, 1, 1, 1, 5]
F??Hw               7                [0, 0, 1, 1, 2, 2, 4]
F??Jg               7                [0, 1, 1, 1, 1, 2, 4]
F_?@w               7                [1, 1, 1, 1, 1, 1, 4]
F??Ww               7                [0, 0, 0, 2, 2, 3, 3]
F?CPW               7                [0, 0, 1, 2, 2, 2, 3]
F?CJG               7                [0, 1, 1, 1, 2, 2, 3]
F??^?               7                [1, 1, 1, 1, 1, 2, 3]
F??XW               7                [0, 0, 1, 1, 2, 3, 3]
F??ZG               7                [0, 1, 1, 1, 1, 3, 3]
F?CZ?               7                [0, 1, 1, 2, 2, 2, 2]
F_?Hg               7                [1, 1, 1, 1, 1, 2, 3]
F?CqO               7                [0, 1, 1, 2, 2, 2, 2]
F?CaW               7                [0, 1, 1, 1, 2, 2, 3]
F?LCG               7                [1, 1, 1, 1, 2, 2, 2]
F?C_w               7                [0, 0, 1, 2, 2, 2, 3]
FG?Gw               7                [0, 1, 1, 1, 2, 2, 3]
F?Ch_               7                [0, 0, 2, 2, 2, 2, 2]
FG?Wo               7                [0, 1, 1, 2, 2, 2, 2]
F_?XO               7                [1, 1, 1, 1, 2, 2, 2]
FK?GW               7                [1, 1, 1, 1, 2, 2, 2]
```

class GraphQuery(*graph_db=None, query_dict=None, display_cols=None, **kwds*)

get_graphs_list()

Returns a list of Sage Graph objects that satisfy the query.

EXAMPLES:

```
sage: Q = GraphQuery(display_cols=['graph6','num_vertices','degree_sequence'],num_edges=['<=
sage: L = Q.get_graphs_list()
sage: L[0]
Graph on 2 vertices
sage: len(L)
35
```

number_of()

Returns the number of graphs in the database that satisfy the query.

EXAMPLES:

```
sage: Q = GraphQuery(display_cols=['graph6','num_vertices','degree_sequence'],num_edges=['<=
sage: Q.number_of()
35
```

query_iterator()

Returns an iterator over the results list of the GraphQuery.

EXAMPLE:

```
sage: Q = GraphQuery(display_cols=['graph6'],num_vertices=7, diameter=5)
sage: for g in Q:
...     print g.graph6_string()
F@?]O
F@OKg
F?`po
F?gqg
FIAHo
F@R@o
FA_pW
FGC{o
FEOhW

sage: Q = GraphQuery(display_cols=['graph6'],num_vertices=7, diameter=5)
sage: it = iter(Q)
sage: while True:
...     try: print it.next().graph6_string()
...     except StopIteration: break
F@?]O
F@OKg
F?`po
F?gqg
FIAHo
F@R@o
FA_pW
FGC{o
FEOhW
```

show(*max_field_size=20, with_picture=False*)

Displays the results of a query in table format.

INPUT:

- `max_field_size` - width of fields in command prompt version
- `with_picture` - whether or not to display results with a picture of the graph (available only in the notebook)

EXAMPLES:

```
sage: G = GraphDatabase()
sage: Q = GraphQuery(G, display_cols=['graph6','num_vertices','aut_grp_size'], num_vertices=
sage: Q.show()
Graph6                Num Vertices          Aut Grp Size
------------------------------------------------------------
C@                    4                     4
C^                    4                     4

sage: R = GraphQuery(G, display_cols=['graph6','num_vertices','degree_sequence'], num_vertic
sage: R.show()
Graph6                Num Vertices          Degree Sequence
------------------------------------------------------------
C?                    4                     [0, 0, 0, 0]
C@                    4                     [0, 0, 1, 1]
CB                    4                     [0, 1, 1, 2]
CK                    4                     [1, 1, 1, 1]
CF                    4                     [1, 1, 1, 3]
CJ                    4                     [0, 2, 2, 2]
CL                    4                     [1, 1, 2, 2]
CN                    4                     [1, 2, 2, 3]
C]                    4                     [2, 2, 2, 2]
C^                    4                     [2, 2, 3, 3]
C~                    4                     [3, 3, 3, 3]
```

Show the pictures (in notebook mode only):

```
sage: S = GraphQuery(G, display_cols=['graph6','aut_grp_size'], num_vertices=4)
sage: S.show(with_picture=True)
...
NotImplementedError: Cannot display plot on command line.
```

Note that pictures can be turned off:

```
sage: S.show(with_picture=False)
Graph6                Aut Grp Size
---------------------------------------
C?                    24
C@                    4
CB                    2
CK                    8
CF                    6
CJ                    6
CL                    2
CN                    2
C]                    8
C^                    4
C~                    24
```

Show your own query (note that the output is not reformatted for generic queries):

```
sage: (GenericGraphQuery('select degree_sequence from degrees where max_degree=2 and min_deg
degree_sequence
--------------------
211
222
2211
2222
21111
22211
```

```
22211
22222
221111
221111
222211
222211
222211
222222
222222
2111111
2221111
2221111
2221111
2222211
2222211
2222211
2222211
2222222
2222222
```

data_to_degseq(*data, graph6=None*)

Takes the database integer data type (one digit per vertex representing its degree, sorted high to low) and converts it to degree sequence list. The graph6 identifier is required for all graphs with no edges, so that the correct number of zeros will be returned.

EXAMPLE:

```
sage: from sage.graphs.graph_database import data_to_degseq
sage: data_to_degseq(3221)
[1, 2, 2, 3]
sage: data_to_degseq(0,'D??')
[0, 0, 0, 0, 0]
```

degseq_to_data(*degree_sequence*)

Takes a degree sequence list (of Integers) and converts to a sorted (max-min) integer data type, as used for faster access in the underlying database.

EXAMPLE:

```
sage: from sage.graphs.graph_database import degseq_to_data
sage: degseq_to_data([2,2,3,1])
3221
```

graph6_to_plot(*graph6*)

Constructs a graph from a graph6 string and returns a Graphics object with arguments preset for show function.

EXAMPLE:

```
sage: from sage.graphs.graph_database import graph6_to_plot
sage: type(graph6_to_plot('D??'))
<class 'sage.plot.plot.Graphics'>
```

graph_db_info(*tablename=None*)

Returns a dictionary of allowed table and column names.

INPUT:

- `tablename` - restricts the output to a single table

EXAMPLE:

```
sage: graph_db_info().keys()
['graph_data', 'degrees', 'spectrum', 'misc', 'aut_grp']

sage: graph_db_info(tablename='graph_data')
['complement_graph6',
 'eulerian',
 'graph6',
 'lovasz_number',
 'num_cycles',
 'num_edges',
 'num_hamiltonian_cycles',
 'num_vertices',
 'perfect',
 'planar']
```

subgraphs_to_query (*subgraphs, db*)

Constructs and returns a GraphQuery object respecting the special input required for the induced_subgraphs parameter. This input can be an individual graph6 string (in which case it is evaluated without the use of this method) or a list of strings. In the latter case, the list should be of one of the following two formats: 1. ['one_of',String,...,String] Will search for graphs containing a subgraph isomorphic to any of the graph6 strings in the list. 2. ['all_of',String,...,String] Will search for graphs containing a subgraph isomorphic to each of the graph6 strings in the list.

This is a helper method called by the GraphQuery constructor to handle this special format. This method should not be used on its own because it doesn't set any display columns in the query string, causing a failure to fetch the data when run.

EXAMPLE:

```
sage: from sage.graphs.graph_database import subgraphs_to_query
sage: gd = GraphDatabase()
sage: q = subgraphs_to_query(['all_of','A?','B?','C?'],gd)
sage: q.get_query_string()
'SELECT , ,, ,,  FROM misc WHERE ( ( misc.induced_subgraphs regexp ? ) AND ( misc.induced_subgra
```

7.5 A module for dealing with lists of graphs.

AUTHORS:

- Robert L. Miller (2007-02-10): initial version
- Emily A. Kirkman (2007-02-13): added show functions (to_graphics_array and show_graphs)

from_graph6 (*data*)

Returns a list of Sage Graphs, given a list of graph6 data.

INPUT:

- `data` - can be a string, a list of strings, or a file stream.

EXAMPLE:

```
sage: l = ['N@@?N@UGAGG?gGlKCMO','XsGGWOW?CC?C@HQKHqOjYKC_uHWGX?P?~TqIKA`OA@SAOEcEA??']
sage: graphs_list.from_graph6(l)
[Graph on 15 vertices, Graph on 25 vertices]
```

from_sparse6(*data*)

Returns a list of Sage Graphs, given a list of sparse6 data.

INPUT:

- `data` - can be a string, a list of strings, or a file stream.

EXAMPLE:

```
sage: l = [':P_`cBaC_ACd`C_@BC`ABDHaEH_@BF_@CHIK_@BCEHKL_BIKM_BFGHI',':f`??KO?B_OOSCGE_?OWONDBO?
sage: graphs_list.from_sparse6(l)
[Looped multi-graph on 17 vertices, Looped multi-graph on 39 vertices]
```

from_whatever(*data*)

Returns a list of Sage Graphs, given a list of whatever kind of data.

INPUT:

- `data` - can be a string, a list of strings, or a file stream, or whatever.

EXAMPLE:

```
sage: l = ['N@@?N@UGAGG?gGlKCMO',':P_`cBaC_ACd`C_@BC`ABDHaEH_@BF_@CHIK_@BCEHKL_BIKM_BFGHI']
sage: graphs_list.from_whatever(l)
[Graph on 15 vertices, Looped multi-graph on 17 vertices]
```

show_graphs(*list, **kwds*)

Shows a maximum of 20 graphs from list in a sage graphics array. If more than 20 graphs are given in the list argument, then it will display one graphics array after another with each containing at most 20 graphs.

Note that if to save the image output from the notebook, you must save each graphics array individually. (There will be a small space between graphics arrays).

INPUT:

- `list` - a list of Sage graphs

GRAPH PLOTTING: Defaults to circular layout for graphs. This allows for a nicer display in a small area and takes much less time to compute than the spring- layout algorithm for many graphs.

EXAMPLES: Create a list of graphs:

```
sage: glist = []
sage: glist.append(graphs.CompleteGraph(6))
sage: glist.append(graphs.CompleteBipartiteGraph(4,5))
sage: glist.append(graphs.BarbellGraph(7,4))
sage: glist.append(graphs.CycleGraph(15))
sage: glist.append(graphs.DiamondGraph())
sage: glist.append(graphs.HouseGraph())
sage: glist.append(graphs.HouseXGraph())
sage: glist.append(graphs.KrackhardtKiteGraph())
sage: glist.append(graphs.LadderGraph(5))
sage: glist.append(graphs.LollipopGraph(5,6))
sage: glist.append(graphs.PathGraph(15))
sage: glist.append(graphs.PetersenGraph())
sage: glist.append(graphs.StarGraph(17))
sage: glist.append(graphs.WheelGraph(9))
```

Check that length is = 20:

```
sage: len(glist)
14
```

Show the graphs in a graphics array:

```
sage: graphs_list.show_graphs(glist)
```

Here's an example where more than one graphics array is used:

```
sage: gq = GraphQuery(display_cols=['graph6'],num_vertices=5)
sage: g = gq.get_graphs_list()
sage: len(g)
34
sage: graphs_list.show_graphs(g)
```

See the .plot() or .show() documentation for an individual graph for options, all of which are available from to_graphics_arrays

```
sage: glist = []
sage: for _ in range(10):
...       glist.append(graphs.RandomLobster(41, .3, .4))
sage: graphs_list.show_graphs(glist, layout='spring', vertex_size=20)
```

to_graph6(*list, file=None, output_list=False*)

Converts a list of Sage graphs to a single string of graph6 graphs. If file is specified, then the string will be written quietly to the file. If output_list is True, then a list of strings will be returned, one string per graph.

INPUT:

- `list` - a Python list of Sage Graphs
- `file` - (optional) a file stream to write to (must be in 'w' mode)
- `output_list` - False - output is a string True - output is a list of strings (ignored if file gets specified)

EXAMPLE:

```
sage: l = [graphs.DodecahedralGraph(), graphs.PetersenGraph()]
sage: graphs_list.to_graph6(l)
'ShCHGD@?K?_@?@?C_GGG@??cG?G?GK_?C\nIheA@GUAo\n'
```

to_graphics_arrays(*list, **kwds*)

Returns a list of Sage graphics arrays containing the graphs in list. The maximum number of graphs per array is 20 (5 rows of 4). Use this function if there are too many graphs for the show_graphs function. The graphics arrays will contain 20 graphs each except potentially the last graphics array in the list.

INPUT:

- `list` - a list of Sage graphs

GRAPH PLOTTING: Defaults to circular layout for graphs. This allows for a nicer display in a small area and takes much less time to compute than the spring- layout algorithm for many graphs.

EXAMPLES:

```
sage: glist = []
sage: for i in range(999):
...     glist.append(graphs.RandomGNP(6,.45))
...
sage: garray = graphs_list.to_graphics_arrays(glist)
```

Display the first graphics array in the list.

```
sage: garray[0].show()
```

Display the last graphics array in the list.

```
sage: garray[len(garray)-1].show()
```

See the .plot() or .show() documentation for an individual graph for options, all of which are available from to_graphics_arrays

```
sage: glist = []
sage: for _ in range(10):
...       glist.append(graphs.RandomLobster(41, .3, .4))
sage: w = graphs_list.to_graphics_arrays(glist, layout='spring', vertex_size=20)
sage: len(w)
1
sage: w[0]
```

to_sparse6 (*list, file=None, output_list=False*)

Converts a list of Sage graphs to a single string of sparse6 graphs. If file is specified, then the string will be written quietly to the file. If output_list is True, then a list of strings will be returned, one string per graph.

INPUT:

- `list` - a Python list of Sage Graphs
- `file` - (optional) a file stream to write to (must be in 'w' mode)
- `output_list` - False - output is a string True - output is a list of strings (ignored if file gets specified)

EXAMPLE:

```
sage: l = [graphs.DodecahedralGraph(), graphs.PetersenGraph()]
sage: graphs_list.to_sparse6(l)
':S_`abcaDe`Fg_HijhKfLdMkNcOjP_BQ\n:I`ES@obGkqegW~\n'
```

CHAPTER

EIGHT

CONSTANTS

8.1 Mathematical constants

The following standard mathematical constants are defined in Sage, along with support for coercing them into GAP, GP/PARI, KASH, Maxima, Mathematica, Maple, Octave, and Singular:

```
sage: pi
pi
sage: e              # base of the natural logarithm
e
sage: NaN            # Not a number
NaN
sage: golden_ratio
golden_ratio
sage: log2           # natural logarithm of the real number 2
log2
sage: euler_gamma    # Euler's gamma constant
euler_gamma
sage: catalan        # the Catalon constant
catalan
sage: khinchin       # Khinchin's constant
khinchin
sage: twinprime
twinprime
sage: merten
merten
sage: brun
brun
```

Support for coercion into the various systems means that if, e.g., you want to create π in Maxima and Singular, you don't have to figure out the special notation for each system. You just type the following:

```
sage: maxima(pi)
%pi
sage: singular(pi)
pi
sage: gap(pi)
pi
sage: gp(pi)
3.1415926535897932384626433832795028842     # 32-bit
3.1415926535897932384626433832795028842   # 64-bit
sage: pari(pi)
3.14159265358979
```

```
sage: kash(pi)                      # optional
3.14159265358979323846264338328
sage: mathematica(pi)               # optional
Pi
sage: maple(pi)                     # optional
Pi
sage: octave(pi)                    # optional
3.14159
```

Arithmetic operations with constants also yield constants, which can be coerced into other systems or evaluated.

```
sage: a = pi + e*4/5; a
pi + 4/5*e
sage: maxima(a)
%pi+4*%e/5
sage: RealField(15)(a)            # 15 *bits* of precision
5.316
sage: gp(a)
5.3162181163570294267508733603616328824    # 64-bit
sage: print mathematica(a)                     # optional
 4 E
 --- + Pi
  5
```

EXAMPLES: Decimal expansions of constants

We can obtain floating point approximations to each of these constants by coercing into the real field with given precision. For example, to 200 decimal places we have the following:

```
sage: R = RealField(200); R
Real Field with 200 bits of precision


sage: R(pi)
3.1415926535897932384626433832795028841971693993751058209749


sage: R(e)
2.7182818284590452353602874713526624977572470936999595749670


sage: R(NaN)
NaN


sage: R(golden_ratio)
1.6180339887498948482045868343656381177203091798057628621354


sage: R(log2)
0.69314718055994530941723212145817656807550013436025525412068


sage: R(euler_gamma)
0.57721566490153286060651209008240243104215933593992359880577


sage: R(catalan)
0.91596559417721901505460351493238411077414937428167213426650
```

```
sage: R(khinchin)
2.685452001065306445309714835481795693820382293994462953051 2
```

EXAMPLES: Arithmetic with constants

```
sage: f = I*(e+1); f
I*e + I
sage: f^2
(I*e + I)^2
sage: _.expand()
-2*e - e^2 - 1
```

```
sage: pp = pi+pi; pp
2*pi
sage: R(pp)
6.283185307179586476925286766559005768394338798750211641949 9
```

```
sage: s = (1 + e^pi); s
e^pi + 1
sage: R(s)
24.140692632779269005729086367948547380266106242600211993445
sage: R(s-1)
23.140692632779269005729086367948547380266106242600211993445
```

```
sage: l = (1-log2)/(1+log2); l
-(log2 - 1)/(log2 + 1)
sage: R(l)
0.181232218299282499487613818646503114233306097747760134880 6
```

```
sage: pim = maxima(pi)
sage: maxima.eval('fpprec : 100')
'100'
sage: pim.bfloat()
3.141592653589793238462643383279502884197169399375105820974944592307816406286208998628034825342117068
```

AUTHORS:

- Alex Clemesha (2006-01-15)
- William Stein
- Alex Clemesha, William Stein (2006-02-20): added new constants; removed todos
- Didier Deshommes (2007-03-27): added constants from RQDF (deprecated)

TESTS:

Coercing the sum of a bunch of the constants to many different floating point rings:

```
sage: a = pi + e + golden_ratio + log2 + euler_gamma + catalan + khinchin + twinprime + merten; a
pi + euler_gamma + catalan + golden_ratio + log2 + khinchin + twinprime + merten + e
sage: parent(a)
Symbolic Ring
sage: RR(a)
13.2713479401972
```

```
sage: RealField(212)(a)
13.271347940197249310098819199575813940871106820003074817832971189555
sage: RealField(230)(a)
13.271347940197249310098819199575813940871106820003074817832971189555
sage: CC(a)
13.2713479401972
sage: CDF(a)
13.2713479402
sage: ComplexField(230)(a)
13.271347940197249310098819199575813940871106820003074817832971189555
sage: RDF(a)
13.2713479402
```

class Brun (*name='brun'*)

Brun's constant is the sum of reciprocals of odd twin primes.

It is not known to very high precision; calculating the number using twin primes up to 10^{16} (Sebah 2002) gives the number 1.9021605831040.

EXAMPLES:

```
sage: float(brun)
...
NotImplementedError: brun is only available up to 41 bits
sage: R = RealField(41); R
Real Field with 41 bits of precision
sage: R(brun)
1.90216058310
```

class Catalan (*name='catalan'*)

A number appaering in combinatorics defined as the Dirichlet beta function evaluated at the number 2.

EXAMPLES:

```
sage: catalan^2 + merten
merten + catalan^2
```

class Constant (*name, conversions=None, latex=None, mathml='', domain='complex'*)

domain()

Returns the domain of this constant. This is either positive, real, or complex, and is used by Pynac to make inferences about expressions containing this constant.

EXAMPLES:

```
sage: p = pi.pyobject(); p
pi
sage: type(_)
<class 'sage.symbolic.constants.Pi'>
sage: p.domain()
'positive'
```

expression()

Returns an expression for this constant.

EXAMPLES:

```
sage: a = pi.pyobject()
sage: pi2 = a.expression()
sage: pi2
```

```
pi
sage: pi2 + 2
pi + 2
sage: pi - pi2
0
```

name()

Returns the name of this constant.

EXAMPLES:

```
sage: from sage.symbolic.constants import Constant
sage: c = Constant('c')
sage: c.name()
'c'
```

class E(*name='e'*)

expression()

Note: For e, we don't return a wrapper around a Pynac constant. Instead, we return exp(1) so that Pynac can perform appropiate.

EXAMPLES:

```
sage: e + 2
e + 2
sage: e.operator()
exp
sage: e.operands()
[1]
```

class EulerGamma(*name='euler_gamma'*)

The limiting difference between the harmonic series and the natural logarithm.

EXAMPLES:

```
sage: R = RealField()
sage: R(euler_gamma)
0.577215664901533
sage: R = RealField(200); R
Real Field with 200 bits of precision
sage: R(euler_gamma)
0.57721566490153286060651209008240243104215933593992359880577
sage: eg = euler_gamma + euler_gamma; eg
2*euler_gamma
sage: R(eg)
1.15443132980306572121302418016480486208431867187984719761115
```

class GoldenRatio(*name='golden_ratio'*)

The number (1+sqrt(5))/2

EXAMPLES:

```
sage: gr = golden_ratio
sage: RR(gr)
1.61803398874989
sage: R = RealField(200)
sage: R(gr)
1.61803398874989484820458683436563811772030917980576286213544
```

```
sage: grm = maxima(golden_ratio);grm
(sqrt(5)+1)/2
sage: grm + grm
sqrt(5)+1
sage: float(grm + grm)
3.2360679774997898
```

minpoly(*bits=None, degree=None, epsilon=0*)

EXAMPLES:

```
sage: golden_ratio.minpoly()
x^2 - x - 1
```

class **I_class**(*name='I'*)

expression(*constant=False*)

Returns an Expression for I. If *constnat* is True, then it returns a wrapper around a Pynac constant. If *constant* is False, then it returns a wrapper around a NumberFieldElement.

EXAMPLES:

```
sage: from sage.symbolic.constants import I_class
sage: a = I_class()
sage: I_constant = a.expression(constant=True)
sage: type(I_constant.pyobject())
<class 'sage.symbolic.constants.I_class'>
sage: I_nf = a.expression()
sage: type(I_nf.pyobject())
<type 'sage.rings.number_field.number_field_element_quadratic.NumberFieldElement_quadratic'>
```

class **Khinchin**(*name='khinchin'*)

The geometric mean of the continued fraction expansion of any (almost any) real number.

EXAMPLES:

```
sage: float(khinchin)
2.6854520010653062
sage: m = mathematica(khinchin); m              # optional
Khinchin
sage: m.N(200)                                  # optional
2.6854520010653064453097148354817956938203822939944629530511523455572188595371520028011411749318
2.6854520010653064453097148354817956938203822939944629530511523455572188595371520028011411749318
```

class **LimitedPrecisionConstant**(*name, value, **kwds*)

class **Log2**(*name='log2'*)

The natural logarithm of the real number 2.

EXAMPLES:

```
sage: log2
log2
sage: float(log2)
0.69314718055994529
sage: RR(log2)
0.693147180559945
sage: R = RealField(200); R
Real Field with 200 bits of precision
sage: R(log2)
0.69314718055994530941723212145817656807550013436025525412068
```

```
sage: l = (1-log2)/(1+log2); l
-(log2 - 1)/(log2 + 1)
sage: R(l)
0.18123221829928249948761381864650311423330609774776013488056
sage: maxima(log2)
log(2)
sage: maxima(log2).float()
.6931471805599453
sage: gp(log2)
0.69314718055994530941723212145817656807   # 64-bit
0.69314718055994530941723212             # 32-bit
```

class Merten (*name='merten'*)

The Merten constant is related to the Twin Primes constant and appears in Merten's second theorem.

EXAMPLES:

```
sage: float(merten)
0.26149721284764277
sage: R=RealField(200);R
Real Field with 200 bits of precision
sage: R(merten)
0.26149721284764278375542683860869585905156664826119920619206
```

class NotANumber (*name='NaN'*)

Not a Number

class Pi (*name='pi'*)

class TwinPrime (*name='twinprime'*)

The Twin Primes constant is defined as $\prod 1 - 1/(p-1)^2$ for primes $p > 2$.

EXAMPLES:

```
sage: float(twinprime)
0.66016181584686962
sage: R=RealField(200);R
Real Field with 200 bits of precision
sage: R(twinprime)
0.66016181584686957392781211001455577843262336028473341331945
```

unpickle_Constant (*class_name, name, conversions, latex, mathml, domain*)

EXAMPLES:

```
sage: from sage.symbolic.constants import unpickle_Constant
sage: a = unpickle_Constant('Constant', 'a', {}, 'aa', '', 'positive')
sage: a.domain()
'positive'
sage: latex(a)
aa
```

Note that if the name already appears in the `constants_name_table`, then that will be returned instead of constructing a new object:

```
sage: pi = unpickle_Constant('Pi', 'pi', None, None, None, None)
sage: pi._maxima_init_()
'%pi'
```

CHAPTER

NINE

FUNCTIONS

9.1 Logarithmic functions

class Function_dilog()

class Function_exp()

class Function_log()

class Function_polylog()

ln(*x*)

The natural logarithm of x.

INPUT:

- x - positive real number

OUTPUT:

- ln(x) - real number

EXAMPLES:

```
sage: ln(e^2)
2
sage: ln(2)
log(2)
sage: ln(2.0)
0.693147180559945
```

log(*x*, *base=None*)

Return the logarithm of x to the given base.

Calls the log method of the object x when computing the logarithm, thus allowing use of logarithm on any object containing a log method. In other words, log works on more than just real numbers.

TODO: Add p-adic log example.

EXAMPLES:

```
sage: log(e^2)
2
sage: log(1024, 2); RDF(log(1024, 2))
10
10.0
sage: log(10, 4); RDF(log(10, 4))
```

```
log(10)/log(4)
1.66096404744

sage: log(10, 2)
log(10)/log(2)
sage: n(log(10, 2))
3.32192809488736
sage: log(10, e)
log(10)
sage: n(log(10, e))
2.30258509299405
```

The log function also works in finite fields as long as the base is generator of the multiplicative group:

```
sage: F = GF(13); g = F.multiplicative_generator(); g
2
sage: a = F(8)
sage: log(a,g); g^log(a,g)
3
8
sage: log(a,3)
...
ValueError: base (=3) for discrete log must generate multiplicative group
```

9.2 Hyperbolic Functions

class Function_arccosh()

class Function_arccoth()

class Function_arccsch()

class Function_arcsech()

class Function_arcsinh()

class Function_arctanh()

class Function_cosh()

class Function_coth()

class Function_csch()

class Function_sech()

class Function_sinh()

class Function_tanh()

class HyperbolicFunction()

9.3 Transcendental Functions

class DickmanRhoComputer()

Dickman's function is the continuous function satisfying the differential equation

$$x\rho'(x) + \rho(x-1) = 0$$

with initial conditions $\rho(x) = 1$ for $0 \leq x \leq 1$. It is useful in estimating the frequency of smooth numbers as asymptotically

$$\Psi(a, a^{1/s}) \sim a\rho(s)$$

where $\Psi(a, b)$ is the number of b-smooth numbers less than a.

ALGORITHM:

Dickmans's function is analytic on the interval $[n, n+1]$ for each integer n. To evalute at $n + t, 0 \leq t < 1$, a power series is recursively computed about $n + 1/2$ using the differential equation stated above. As high precision arithmetic may be needed for intermediate results the computed series are cached for later use.

Simple explicit formulas are used for the intervals [0,1] and [1,2].

EXAMPLES:

```
sage: dickman_rho(2)
0.306852819440055
sage: dickman_rho(10)
2.77017183772596e-11
sage: dickman_rho(10.000000000000000000000000000000000000000)
2.7701718377259589887581212006343423263 43e-11
sage: plot(log(dickman_rho(x)), (x, 0, 15))
```

AUTHORS:

- Robert Bradshaw (2008-09)

REFERENCES:

- G. Marsaglia, A. Zaman, J. Marsaglia. "Numerical Solutions to some Classical Differential-Difference Equations." Mathematics of Computation, Vol. 53, No. 187 (1989).

approximate (*x*)

Approximate using de Bruijn's formula

$$\rho(x) \sim \frac{exp(-x\xi + Ei(\xi))}{\sqrt{2\pi x}\xi}$$

which is asymptotically equal to Dickman's function, and is much faster to compute.

REFERENCES:

- N. De Bruijn, "The Asymptotic behavior of a function occuring in the theory of primes." J. Indian Math Soc. v 15. (1951)

EXAMPLES:

```
sage: dickman_rho.approximate(10)
2.41739196365564e-11
sage: dickman_rho(10)
2.77017183772596e-11
sage: dickman_rho.approximate(1000)
4.32938809066403e-3464
```

power_series (*n, abs_prec*)

This function returns the power series about $n+1/2$ used to evaluate Dickman's function. It is scaled such that the interval $[n, n+1]$ corresponds to x in $[-1, 1]$.

INPUT:

- `n` - the lower endpoint of the interval for which this power series holds
- `abs_prec` - the absolute precision of the resulting power series

EXAMPLES:

```
sage: f = dickman_rho.power_series(2, 20); f
-9.9376e-8*x^11 + 3.7722e-7*x^10 - 1.4684e-6*x^9 + 5.8783e-6*x^8 - 0.000024259*x^7 + 0.00010
sage: f(-1), f(0), f(1)
(0.30685, 0.13032, 0.048608)
sage: dickman_rho(2), dickman_rho(2.5), dickman_rho(3)
(0.306852819440055, 0.130319561832251, 0.0486083882911316)
```

Ei (*z*)

Return the value of the complex exponential integral Ei(z) at a complex number z.

> **Warning:** Calculations are done to double precision, and the output is a complex double element, no matter how big the precision of the input is.

EXAMPLES:

```
sage: Ei(10)
2492.22897624
sage: Ei(I)
0.337403922901 + 2.51687939716*I
sage: Ei(3+I)
7.823134676 + 6.09751978399*I
```

The branch cut for this function is along the positive real axis:

```
sage: Ei(3 + 0.1*I)
9.91152770287 + 0.668898200718*I
sage: Ei(3 - 0.1*I)
9.91152770287 + 5.61428710646*I
```

ALGORITHM: Uses scipy's special.exp1 function.

Li (*x, eps_rel=None, err_bound=False*)

Return value of the function Li(x) as a real double field element.

This is the function

$$\int_2^x dt/\log(t).$$

The function Li(x) is an approximation for the number of primes up to x. In fact, the famous Riemann Hypothesis is equivalent to the statement that for $x \geq 2.01$ we have

$$|\pi(x) - Li(x)| \leq \sqrt{x}\log(x).$$

For "small" x, $Li(x)$ is always slightly bigger than $\pi(x)$. However it is a theorem that there are (very large, e.g., around 10^{316}) values of x so that $\pi(x) > Li(x)$. See "A new bound for the smallest x with $\pi(x) > li(x)$", Bays and Hudson, Mathematics of Computation, 69 (2000) 1285-1296.

ALGORITHM: Computed numerically using GSL.

INPUT:

- x - a real number = 2.

OUTPUT:

- x - a real double

EXAMPLES:

```
sage: Li(2)
0.0
sage: Li(5)
2.58942452992
sage: Li(1000)
176.56449421
sage: Li(10^5)
9628.76383727
sage: prime_pi(10^5)
9592
sage: Li(1)
...
ValueError: Li only defined for x at least 2.
```

```
sage: for n in range(1,7):
...     print '%-10s%-10s%-20s'%(10^n, prime_pi(10^n), Li(10^n))
10        4         5.12043572467
100       25        29.080977804
1000      168       176.56449421
10000     1229      1245.09205212
100000    9592      9628.76383727
1000000   78498     78626.5039957
```

exponential_integral_1 (*x, n=0*)

Returns the exponential integral $E_1(x)$. If the optional argument n is given, computes list of the first n values of the exponential integral $E_1(xm)$.

The exponential integral $E_1(x)$ is

$$E_1(x) = \int_x^\infty e^{-t}/t dt$$

INPUT:

- `x` - a positive real number
- `n` - (default: 0) a nonnegative integer; if nonzero, then return a list of values E_1(x*m) for m = 1,2,3,...,n. This is useful, e.g., when computing derivatives of L-functions.

OUTPUT:

- `float` - if n is 0 (the default) or
- `list` - list of floats if n 0

EXAMPLES:

```
sage: exponential_integral_1(2)
0.048900510708061118
sage: w = exponential_integral_1(2,4); w
[0.048900510708061118, 0.003779352409848905, 0.00036008245216265542, 3.7665622843921715e-05] # 3
[0.048900510708061118, 0.003779352409848905, 0.00036008245216265542, 3.7665622843921498e-05] # 64
```

IMPLEMENTATION: We use the PARI C-library functions eint1 and veceint1.

REFERENCE:

- See page 262, Prop 5.6.12, of Cohen's book "A Course in Computational Algebraic Number Theory".

REMARKS: When called with the optional argument n, the PARI C-library is fast for values of n up to some bound, then very very slow. For example, if x=5, then the computation takes less than a second for n=800000, and takes "forever" for n=900000.

gamma_inc(*s*, *t*)
Incomplete Gamma function Gamma(s,t).

EXAMPLES:

```
sage: gamma_inc(CDF(0,1), 3)
0.00320857499337 + 0.0124061858119*I
sage: gamma_inc(3, 3)
0.846380162253687
sage: gamma_inc(RDF(1), 3)
0.0497870683678639
```

incomplete_gamma(*s*, *t*)
Incomplete Gamma function Gamma(s,t).

EXAMPLES:

```
sage: gamma_inc(CDF(0,1), 3)
0.00320857499337 + 0.0124061858119*I
sage: gamma_inc(3, 3)
0.846380162253687
sage: gamma_inc(RDF(1), 3)
0.0497870683678639
```

zeta(*s*)
Riemann zeta function at s with s a real or complex number.

INPUT:

- s - real or complex number

If s is a real number the computation is done using the MPFR library. When the input is not real, the computation is done using the PARI C library.

EXAMPLES:

```
sage: zeta(2)
1.64493406684823
sage: RR = RealField(200)
sage: zeta(RR(2))
1.6449340668482264364724151666460251892189499012067984377356
sage: zeta(I)
zeta(I)
sage: zeta(I).n()
0.00330022368532410 - 0.418155449141322*I
```

zeta_symmetric(*s*)
Completed function $\xi(s)$ that satisfies $\xi(s) = \xi(1-s)$ and has zeros at the same points as the Riemann zeta function.

INPUT:

- s - real or complex number

If s is a real number the computation is done using the MPFR library. When the input is not real, the computation is done using the PARI C library.

More precisely,

$$xi(s) = \gamma(s/2+1) * (s-1) * \pi^{-s/2} * \zeta(s).$$

EXAMPLES:

```
sage: zeta_symmetric(0.7)
0.497580414651127
sage: zeta_symmetric(1-0.7)
0.497580414651127
sage: RR = RealField(200)
sage: zeta_symmetric(RR(0.7))
0.49758041465112690357779107525638385212657443284080589766062
sage: C.<i> = ComplexField()
sage: zeta_symmetric(0.5 + i*14.0)
0.000201294444235258 + 1.49077798716757e-19*I
sage: zeta_symmetric(0.5 + i*14.1)
0.0000489893483255687 + 4.40457132572236e-20*I
sage: zeta_symmetric(0.5 + i*14.2)
-0.0000868931282620101 + 7.11507675693612e-20*I
```

REFERENCE:

•I copied the definition of xi from http://www.math.ubc.ca/~pugh/RiemannZeta/RiemannZetaLong.html

9.4 Piecewise-defined Functions.

Sage implements a very simple class of piecewise-defined functions. Functions may be any type of symbolic expression. Infinite intervals are not supported. The endpoints of each interval must line up.

TODO:

- Implement max/min location and values,
- Need: parent object - ring of piecewise functions
- This class should derive from an element-type class, and should define `_add_`, `_mul_`, etc. That will automatically take care of left multiplication and proper coercion. The coercion mentioned below for scalar mult on right is bad, since it only allows ints and rationals. The right way is to use an element class and only define `_mul_`, and have a parent, so anything gets coerced properly.

AUTHORS:

- David Joyner (2006-04): initial version
- David Joyner (2006-09): added __eq__, extend_by_zero_to, unextend, convolution, trapezoid, trapezoid_integral_approximation, riemann_sum, riemann_sum_integral_approximation, tangent_line fixed bugs in __mul__, __add__
- David Joyner (2007-03): adding Hann filter for FS, added general FS filter methods for computing and plotting, added options to plotting of FS (eg, specifying rgb values arenow allowed). Fixed bug in documentation reported by Pablo De Napoli.
- David Joyner (2007-09): bug fixes due to behaviour of SymbolicArithmetic
- David Joyner (2008-04): fixed docstring bugs reported by J Morrow; added support for laplace transform of functions with infinite support.
- David Joyner (2008-07): fixed a left multiplication bug reported by C. Boncelet (by defining __rmul__ = __mul__).
- Paul Butler (2009-01): added indefinite integration and default_variable

TESTS:

```
sage: R.<x> = QQ[]
sage: f = Piecewise([[(0,1),1*x^0]])
sage: 2*f
Piecewise defined function with 1 parts, [[(0, 1), 2]]
```

Piecewise(*list_of_pairs, var=None*)

Returns a piecewise function from a list of (interval, function) pairs.

`list_of_pairs` is a list of pairs (I, fcn), where fcn is a Sage function (such as a polynomial over RR, or functions using the lambda notation), and I is an interval such as I = (1,3). Two consecutive intervals must share a common endpoint.

If the optional `var` is specified, then any symbolic expressions in the list will be converted to symbolic functions using `fcn.function(var)`. (This says which variable is considered to be "piecewise".)

We assume that these definitions are consistent (ie, no checking is done).

EXAMPLES:

```
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f(1)
-1
sage: f(3)
2
sage: f = Piecewise([[(0,1),x], [(1,2),x^2]], x); f
Piecewise defined function with 2 parts, [[(0, 1), x |--> x], [(1, 2), x |--> x^2]]
sage: f(0.9)
0.900000000000000
sage: f(1.1)
1.21000000000000
```

class **PiecewisePolynomial**(*list_of_pairs, var=None*)

Returns a piecewise function from a list of (interval, function) pairs.

EXAMPLES:

```
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f(1)
-1
sage: f(3)
2
```

base_ring()

Returns the base-ring (ie, QQ[x]) - useful when this class is extended.

convolution(*other*)

Returns the convolution function, $f * g(t) = \int_{-\infty}^{\infty} f(u)g(t-u)du$, for compactly supported f, g.

EXAMPLES:

```
sage: x = PolynomialRing(QQ,'x').gen()
sage: f = Piecewise([[(0,1),1*x^0]])  ## example 0
sage: g = f.convolution(f)
sage: h = f.convolution(g)
sage: P = f.plot(); Q = g.plot(rgbcolor=(1,1,0)); R = h.plot(rgbcolor=(0,1,1));
sage: # Type show(P+Q+R) to view
```

```
sage: f = Piecewise([[(0,1),1*x^0],[(1,2),2*x^0],[(2,3),1*x^0]])  ## example 1
sage: g = f.convolution(f)
sage: h = f.convolution(g)
sage: P = f.plot(); Q = g.plot(rgbcolor=(1,1,0)); R = h.plot(rgbcolor=(0,1,1));
sage: # Type show(P+Q+R) to view
sage: f = Piecewise([[(-1,1),1]])                              ## example 2
sage: g = Piecewise([[(0,3),x]])
sage: f.convolution(g)
Piecewise defined function with 3 parts, [[(-1, 1), 0], [(1, 2), -3/2*x], [(2, 4), -3/2*x]]
sage: g = Piecewise([[(0,3),1*x^0],[(3,4),2*x^0]])
sage: f.convolution(g)
Piecewise defined function with 5 parts, [[(-1, 1), x + 1], [(1, 2), 3], [(2, 3), x], [(3, 4
```

cosine_series_coefficient(*n, L*)

Returns the n-th cosine series coefficient of $\cos(n\pi x/L)$, a_n.

INPUT:

- `self` - the function f(x), defined over 0 x L (no checking is done to insure this)
- `n` - an integer n=0
- `L` - (the period)/2

OUTPUT: $a_n = \frac{2}{L}\int_{-L}^{L} f(x)\cos(n\pi x/L)dx$ such that

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(\frac{n\pi x}{L}),\ \ 0 < x < L.$$

EXAMPLES:

```
sage: f(x) = x
sage: f = Piecewise([[(0,1),f]])
sage: f.cosine_series_coefficient(2,1)
0
sage: f.cosine_series_coefficient(3,1)
-4/9/pi^2
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f.cosine_series_coefficient(2,pi)
0
sage: f.cosine_series_coefficient(3,pi)
2/pi
sage: f.cosine_series_coefficient(111,pi)
2/37/pi
sage: f1 = lambda x: x*(pi-x)
sage: f = Piecewise([[(0,pi),f1]])
sage: f.cosine_series_coefficient(0,pi)
1/3*pi^2
```

critical_points()

Return the critical points of this piecewise function.

> **Warning:** Uses maxima, which prints the warning to use results with caution. Only works for piecewise functions whose parts are polynomials with real critical not occurring on the interval endpoints.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f1 = x^0
```

```
sage: f2 = 10*x - x^2
sage: f3 = 3*x^4 - 156*x^3 + 3036*x^2 - 26208*x
sage: f = Piecewise([[(0,3),f1],[(3,10),f2],[(10,20),f3]])
sage: expected = [5, 12, 13, 14]
sage: all(abs(e-a) < 0.001 for e,a in zip(expected, f.critical_points()))
True
```

default_variable()

Return the default variable. The default variable is defined as the first variable in the first piece uses a variable. If no pieces have a variable (each piece is a constant value), x is returned.

The result is cached.

AUTHOR: Paul Butler

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 5*x
sage: p = Piecewise([[(0,1),f1],[(1,4),f2]])
sage: p.default_variable()
x
```

derivative()

Returns the derivative (as computed by maxima) Piecewise(I,‘(d/dx)(selfl_I)‘), as I runs over the intervals belonging to self. self must be piecewise polynomial.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.derivative()
Piecewise defined function with 2 parts, [[(0, 1), x |--> 0], [(1, 2), x |--> -1]]
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f.derivative()
Piecewise defined function with 2 parts, [[(0, 1/2*pi), x |--> 0], [(1/2*pi, pi), x |--> 0]]

sage: f = Piecewise([[(0,1), (x * 2)]], x)
sage: f.derivative()
Piecewise defined function with 1 parts, [[(0, 1), x |--> 2]]
```

domain()

Returns the domain of the function.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = exp(x)
sage: f4(x) = sin(2*x)
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: f.domain()
(0, 10)
```

end_points()

Returns a list of all interval endpoints for this function.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
```

```
sage: f3(x) = x^2-5
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3]])
sage: f.end_points()
[0, 1, 2, 3]
```

extend_by_zero_to (*xmin=-1000, xmax=1000*)

This function simply returns the piecewise defined function which is extended by 0 so it is defined on all of (xmin,xmax). This is needed to add two piecewise functions in a reasonable way.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1 - x
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.extend_by_zero_to(-1, 3)
Piecewise defined function with 4 parts, [[(-1, 0), 0], [(0, 1), x |--> 1], [(1, 2), x |-->
```

fourier_series_cosine_coefficient (*n, L*)

Returns the n-th Fourier series coefficient of $\cos(n\pi x/L)$, a_n.

INPUT:

- `self` - the function f(x), defined over -L x L
- `n` - an integer n=0
- `L` - (the period)/2

OUTPUT: $a_n = \frac{1}{L}\int_{-L}^{L} f(x)\cos(n\pi x/L)dx$

EXAMPLES:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_cosine_coefficient(2,1)
pi^(-2)
sage: f(x) = x^2
sage: f = Piecewise([[(-pi,pi),f]])
sage: f.fourier_series_cosine_coefficient(2,pi)
1
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_cosine_coefficient(5,pi)
-3/5/pi
```

fourier_series_partial_sum (*N, L*)

Returns the partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N}[a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

as a string.

EXAMPLE:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_partial_sum(3,1)
-4*cos(pi*x)/pi^2 + cos(2*pi*x)/pi^2 + 1/3
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_partial_sum(3,pi)
-3*sin(2*x)/pi + 3*sin(x)/pi - 3*cos(x)/pi - 1/4
```

fourier_series_partial_sum_cesaro(*N, L*)

Returns the Cesaro partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} (1 - n/N) * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

as a string. This is a "smoother" partial sum - the Gibbs phenomenon is mollified.

EXAMPLE:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_partial_sum_cesaro(3,1)
-8/3*cos(pi*x)/pi^2 + 1/3*cos(2*pi*x)/pi^2 + 1/3
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_partial_sum_cesaro(3,pi)
-sin(2*x)/pi + 2*sin(x)/pi - 2*cos(x)/pi - 1/4
```

fourier_series_partial_sum_filtered(*N, L, F*)

Returns the "filtered" partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} F_n * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

as a string, where $F = [F_1, F_2, ..., F_N]$ is a list of length N consisting of real numbers. This can be used to plot FS solutions to the heat and wave PDEs.

EXAMPLE:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_partial_sum_filtered(3,1,[1,1,1])
-4*cos(pi*x)/pi^2 + cos(2*pi*x)/pi^2 + 1/3
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_partial_sum_filtered(3,pi,[1,1,1])
-3*sin(2*x)/pi + 3*sin(x)/pi - 3*cos(x)/pi - 1/4
```

fourier_series_partial_sum_hann(*N, L*)

Returns the Hann-filtered partial sum (named after von Hann, not Hamming)

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} H_N(n) * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

as a string, where $H_N(x) = (1+\cos(\pi x/N))/2$. This is a "smoother" partial sum - the Gibbs phenomenon is mollified.

EXAMPLE:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_partial_sum_hann(3,1)
-3*cos(pi*x)/pi^2 + 1/4*cos(2*pi*x)/pi^2 + 1/3
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_partial_sum_hann(3,pi)
-3/4*sin(2*x)/pi + 9/4*sin(x)/pi - 9/4*cos(x)/pi - 1/4
```

fourier_series_sine_coefficient(*n, L*)

Returns the n-th Fourier series coefficient of $\sin(n\pi x/L)$, b_n.

INPUT:

- `self` - the function f(x), defined over -L x L
- `n` - an integer n0
- `L` - (the period)/2

OUTPUT: $b_n = \frac{1}{L}\int_{-L}^{L} f(x)\sin(n\pi x/L)dx$

EXAMPLES:

```
sage: f(x) = x^2
sage: f = Piecewise([[(-1,1),f]])
sage: f.fourier_series_sine_coefficient(2,1)  # L=1, n=2
0
```

fourier_series_value(*x, L*)

Returns the value of the Fourier series coefficient of self at x,

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{\infty}[a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})], \quad -L < x < L.$$

This method applies to piecewise non-polynomial functions as well.

INPUT:

- `self` - the function f(x), defined over -L x L
- `x` - a real number
- `L` - (the period)/2

OUTPUT: $(f^*(x+) + f^*(x-)/2$, where f^* denotes the function f extended to $\mathbf{R}$ with period $2L$ (Dirichlet's Theorem for Fourier series).

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = exp(x)
sage: f4(x) = sin(2*x)
sage: f = Piecewise([[(-10,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: f.fourier_series_value(101,10)
1/2
sage: f.fourier_series_value(100,10)
1
sage: f.fourier_series_value(10,10)
1/2*sin(20) + 1/2
sage: f.fourier_series_value(20,10)
1
sage: f.fourier_series_value(30,10)
1/2*sin(20) + 1/2
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,0),lambda x:0],[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f.fourier_series_value(-1,pi)
0
sage: f.fourier_series_value(20,pi)
-1
sage: f.fourier_series_value(pi/2,pi)
1/2
```

functions()

Returns the list of functions (the "pieces").

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = exp(x)
sage: f4(x) = sin(2*x)
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: f.functions()
[x |--> 1, x |--> -x + 1, x |--> e^x, x |--> sin(2*x)]
```

integral (*x=None, a=None, b=None, definite=False*)

By default, returns the indefinite integral of the function. If definite=True is given, returns the definite integral.

AUTHOR:

- Paul Butler

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.integral(definite=True)
1/2

sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f.integral(definite=True)
1/2*pi

sage: f1(x) = 2
sage: f2(x) = 3 - x
sage: f = Piecewise([[(-2, 0), f1], [(0, 3), f2]])
sage: f.integral()
Piecewise defined function with 2 parts, [[(-2, 0), x |--> 2*x + 4], [(0, 3), x |--> -1/2*x^

sage: f1(y) = -1
sage: f2(y) = y + 3
sage: f3(y) = -y - 1
sage: f4(y) = y^2 - 1
sage: f5(y) = 3
sage: f = Piecewise([[(-4,-3),f1],[(-3,-2),f2],[(-2,0),f3],[(0,2),f4],[(2,3),f5]])
sage: F = f.integral(y)
sage: F
Piecewise defined function with 5 parts, [[(-4, -3), y |--> -y - 4], [(-3, -2), y |--> 1/2*y
```

Ensure results are consistant with FTC:

```
sage: F(-3) - F(-4)
-1
sage: F(-1) - F(-3)
1
sage: F(2) - F(0)
2/3
sage: f.integral(y, 0, 2)
2/3
sage: F(3) - F(-4)
19/6
sage: f.integral(y, -4, 3)
19/6
sage: f.integral(definite=True)
```

```
19/6

sage: f1(y) = (y+3)^2
sage: f2(y) = y+3
sage: f3(y) = 3
sage: f = Piecewise([[(-infinity, -3), f1], [(-3, 0), f2], [(0, infinity), f3]])
sage: f.integral()
Piecewise defined function with 3 parts, [[(-Infinity, -3), y |--> 1/3*y^3 + 3*y^2 + 9*y + 9

sage: f1(x) = e^(-abs(x))
sage: f = Piecewise([[(-infinity, infinity), f1]])
sage: f.integral(definite=True)
2
sage: f.integral()
Piecewise defined function with 1 parts, [[(-Infinity, +Infinity), x |--> -integrate(e^(-abs
```

intervals()

A piecewise non-polynomial example.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = exp(x)
sage: f4(x) = sin(2*x)
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: f.intervals()
[(0, 1), (1, 2), (2, 3), (3, 10)]
```

laplace(*x='x', s='t'*)

Returns the Laplace transform of self with respect to the variable var.

INPUT:

- x - variable of self
- s - variable of Laplace transform.

We assume that a piecewise function is 0 outside of its domain and that the left-most endpoint of the domain is 0.

EXAMPLES:

```
sage: x, s, w = var('x, s, w')
sage: f = Piecewise([[(0,1),1],[(1,2), 1-x]])
sage: f.laplace(x, s)
(s + 1)*e^(-2*s)/s^2 - e^(-s)/s + 1/s - e^(-s)/s^2
sage: f.laplace(x, w)
(w + 1)*e^(-2*w)/w^2 - e^(-w)/w + 1/w - e^(-w)/w^2

sage: y, t = var('y, t')
sage: f = Piecewise([[(1,2), 1-y]])
sage: f.laplace(y, t)
(t + 1)*e^(-2*t)/t^2 - e^(-t)/t^2

sage: s = var('s')
sage: t = var('t')
sage: f1(t) = -t
sage: f2(t) = 2
sage: f = Piecewise([[(0,1),f1],[(1,infinity),f2]])
sage: f.laplace(t,s)
(s + 1)*e^(-s)/s^2 + 2*e^(-s)/s - 1/s^2
```

length()

Retuns the number of pieces of this function.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1 - x
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.length()
2
```

list()

plot(**args, **kwds*)

Returns the plot of self.

Keyword arguments are passed onto the plot command for each piece of the function. E.g., the plot_points keyword affects each segment of the plot.

EXAMPLES:

```
sage: f1(x) = 1
sage: f2(x) = 1-x
sage: f3(x) = exp(x)
sage: f4(x) = sin(2*x)
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: P = f.plot(rgbcolor=(0.7,0.1,0), plot_points=40)
sage: P
```

Remember: to view this, type show(P) or P.save("path/myplot.png") and then open it in a graphics viewer such as GIMP.

plot_fourier_series_partial_sum(*N, L, xmin, xmax, **kwds*)

Plots the partial sum

$$f(x) \sim \frac{a_0}{2} + sum_{n=1}^{N}[a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

over xmin x xmin.

EXAMPLE:

```
sage: f1(x) = -2
sage: f2(x) = 1
sage: f3(x) = -1
sage: f4(x) = 2
sage: f = Piecewise([[(-pi,-pi/2),f1],[(-pi/2,0),f2],[(0,pi/2),f3],[(pi/2,pi),f4]])
sage: P = f.plot_fourier_series_partial_sum(3,pi,-5,5)     # long time
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: P = f.plot_fourier_series_partial_sum(15,pi,-5,5)    # long time
```

Remember, to view this type show(P) or P.save("path/myplot.png") and then open it in a graphics viewer such as GIMP.

plot_fourier_series_partial_sum_cesaro(*N, L, xmin, xmax, **kwds*)

Plots the partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N}(1 - n/N) * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

over xmin x xmin. This is a "smoother" partial sum - the Gibbs phenomenon is mollified.

EXAMPLE:

```
sage: f1(x) = -2
sage: f2(x) = 1
sage: f3(x) = -1
sage: f4(x) = 2
sage: f = Piecewise([[(-pi,-pi/2),f1],[(-pi/2,0),f2],[(0,pi/2),f3],[(pi/2,pi),f4]])
sage: P = f.plot_fourier_series_partial_sum_cesaro(3,pi,-5,5)     # long time
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: P = f.plot_fourier_series_partial_sum_cesaro(15,pi,-5,5)    # long time
```

Remember, to view this type show(P) or P.save("path/myplot.png") and then open it in a graphics viewer such as GIMP.

plot_fourier_series_partial_sum_filtered(*N, L, F, xmin, xmax, **kwds*)

Plots the partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} F_n * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

over xmin x xmin, where $F = [F_1, F_2, ..., F_N]$ is a list of length N consisting of real numbers. This can be used to plot FS solutions to the heat and wave PDEs.

EXAMPLE:

```
sage: f1(x) = -2
sage: f2(x) = 1
sage: f3(x) = -1
sage: f4(x) = 2
sage: f = Piecewise([[(-pi,-pi/2),f1],[(-pi/2,0),f2],[(0,pi/2),f3],[(pi/2,pi),f4]])
sage: P = f.plot_fourier_series_partial_sum_filtered(3,pi,[1]*3,-5,5)     # long time
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,-pi/2),f1],[(-pi/2,0),f2],[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: P = f.plot_fourier_series_partial_sum_filtered(15,pi,[1]*15,-5,5)    # long time
```

Remember, to view this type show(P) or P.save("path/myplot.png") and then open it in a graphics viewer such as GIMP.

plot_fourier_series_partial_sum_hann(*N, L, xmin, xmax, **kwds*)

Plots the partial sum

$$f(x) \sim \frac{a_0}{2} + \sum_{n=1}^{N} H_N(n) * [a_n \cos(\frac{n\pi x}{L}) + b_n \sin(\frac{n\pi x}{L})],$$

over xmin x xmin, where H_N(x) = (0.5)+(0.5)*cos(x*pi/N) is the N-th Hann filter.

EXAMPLE:

```
sage: f1(x) = -2
sage: f2(x) = 1
sage: f3(x) = -1
sage: f4(x) = 2
sage: f = Piecewise([[(-pi,-pi/2),f1],[(-pi/2,0),f2],[(0,pi/2),f3],[(pi/2,pi),f4]])
sage: P = f.plot_fourier_series_partial_sum_hann(3,pi,-5,5)     # long time
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(-pi,pi/2),f1],[(pi/2,pi),f2]])
sage: P = f.plot_fourier_series_partial_sum_hann(15,pi,-5,5)    # long time
```

Remember, to view this type show(P) or P.save("path/myplot.png") and then open it in a graphics viewer such as GIMP.

riemann_sum(*N, mode=None*)

Returns the piecewise line function defined by the Riemann sums in numerical integration based on a subdivision into N subintervals. Set mode="midpoint" for the height of the rectangles to be determined by the midpoint of the subinterval; set mode="right" for the height of the rectangles to be determined by the right-hand endpoint of the subinterval; the default is mode="left" (the height of the rectangles to be determined by the leftt-hand endpoint of the subinterval).

EXAMPLES:

```
sage: f1(x) = x^2
sage: f2(x) = 5-x^2
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.riemann_sum(6,mode="midpoint")
Piecewise defined function with 6 parts, [[(0, 1/3), 1/36], [(1/3, 2/3), 1/4], [(2/3, 1), 25

sage: f = Piecewise([[(-1,1),(1-x^2).function(x)]])
sage: rsf = f.riemann_sum(7)
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: Q = rsf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
sage: L = add([line([[a,0],[a,f(x=a)]],rgbcolor=(0.7,0.6,0.6)) for (a,b),f in rsf.list()])
sage: P + Q + L

sage: f = Piecewise([[(-1,1),(1/2+x-x^3)]], x) ## example 3
sage: rsf = f.riemann_sum(8)
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: Q = rsf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
sage: L = add([line([[a,0],[a,f(x=a)]],rgbcolor=(0.7,0.6,0.6)) for (a,b),f in rsf.list()])
sage: P + Q + L
```

riemann_sum_integral_approximation(*N, mode=None*)

Returns the piecewise line function defined by the Riemann sums in numerical integration based on a subdivision into N subintervals.

Set mode="midpoint" for the height of the rectangles to be determined by the midpoint of the subinterval; set mode="right" for the height of the rectangles to be determined by the right-hand endpoint of the subinterval; the default is mode="left" (the height of the rectangles to be determined by the leftt-hand endpoint of the subinterval).

EXAMPLES:

```
sage: f1(x) = x^2                    ## example 1
sage: f2(x) = 5-x^2
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.riemann_sum_integral_approximation(6)
17/6
sage: f.riemann_sum_integral_approximation(6,mode="right")
19/6
sage: f.riemann_sum_integral_approximation(6,mode="midpoint")
3
sage: f.integral(definite=True)
3
```

sine_series_coefficient(*n, L*)

Returns the n-th sine series coefficient of $\sin(n\pi x/L)$, b_n.

INPUT:

- `self` - the function f(x), defined over 0 x L (no checking is done to insure this)
- `n` - an integer n0
- `L` - (the period)/2

OUTPUT:

$b_n = \frac{2}{L}\int_{-L}^{L} f(x)\sin(n\pi x/L)dx$ such that

$$f(x) \sim \sum_{n=1}^{\infty} b_n \sin(\frac{n\pi x}{L}),\ \ 0 < x < L.$$

EXAMPLES:

```
sage: f(x) = 1
sage: f = Piecewise([[(0,1),f]])
sage: f.sine_series_coefficient(2,1)
0
sage: f.sine_series_coefficient(3,1)
4/3/pi
```

tangent_line(*pt*)

Computes the linear function defining the tangent line of the piecewise function self.

EXAMPLES:

```
sage: f1(x) = x^2
sage: f2(x) = 5-x^3+x
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: tf = f.tangent_line(0.9) ## tangent line at x=0.9
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: Q = tf.plot(rgbcolor=(0.7,0.2,0.2), plot_points=40)
sage: P + Q
```

trapezoid(*N*)

Returns the piecewise line function defined by the trapezoid rule for numerical integration based on a subdivision into N subintervals.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f1 = x^2
sage: f2 = 5-x^2
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: f.trapezoid(4)
Piecewise defined function with 4 parts, [[(0, 1/2), 1/2*x], [(1/2, 1), 9/2*x - 2], [(1, 3/2

sage: R.<x> = QQ[]
sage: f = Piecewise([[(-1,1),1-x^2]])
sage: tf = f.trapezoid(4)
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: Q = tf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
sage: L = add([line([[a,0],[a,f(a)]],rgbcolor=(0.7,0.6,0.6)) for (a,b),f in tf.list()])
sage: P+Q+L

sage: R.<x> = QQ[]
sage: f = Piecewise([[(-1,1),1/2+x-x^3]]) ## example 3
sage: tf = f.trapezoid(6)
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: Q = tf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
sage: L = add([line([[a,0],[a,f(a)]],rgbcolor=(0.7,0.6,0.6)) for (a,b),f in tf.list()])
sage: P+Q+L
```

trapezoid_integral_approximation(*N*)

Returns the approximation given by the trapezoid rule for numerical integration based on a subdivision into N subintervals.

EXAMPLES:

```
sage: f1(x) = x^2                          ## example 1
sage: f2(x) = 1-(1-x)^2
sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])
sage: P = f.plot(rgbcolor=(0.7,0.1,0.5), plot_points=40)
sage: tf = f.trapezoid(6)
sage: Q = tf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
sage: ta = f.trapezoid_integral_approximation(6)
sage: t = text('trapezoid approximation = %s'%ta, (1.5, 0.25))
sage: a = f.integral(definite=True)
sage: tt = text('area under curve = %s'%a, (1.5, -0.5))
sage: P + Q + t + tt

sage: f = Piecewise([[(0,1),f1],[(1,2),f2]])  ## example 2
sage: tf = f.trapezoid(4)
sage: ta = f.trapezoid_integral_approximation(4)
sage: Q = tf.plot(rgbcolor=(0.7,0.6,0.6), plot_points=40)
sage: t = text('trapezoid approximation = %s'%ta, (1.5, 0.25))
sage: a = f.integral(definite=True)
sage: tt = text('area under curve = %s'%a, (1.5, -0.5))
sage: P+Q+t+tt
```

unextend()

This removes any parts in the front or back of the function which is zero (the inverse to extend_by_zero_to).

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = Piecewise([[(-3,-1),1+2+x],[(-1,1),1-x^2]])
sage: e = f.extend_by_zero_to(-10,10); e
Piecewise defined function with 4 parts, [[(-10, -3), 0], [(-3, -1), x + 3], [(-1, 1), -x^2
sage: d = e.unextend(); d
Piecewise defined function with 2 parts, [[(-3, -1), x + 3], [(-1, 1), -x^2 + 1]]
sage: d==f
True
```

which_function(*x0*)

Returns the function piece used to evaluate self at x0.

EXAMPLES:

```
sage: f1(z) = z
sage: f2(x) = 1-x
sage: f3(y) = exp(y)
sage: f4(t) = sin(2*t)
sage: f = Piecewise([[(0,1),f1],[(1,2),f2],[(2,3),f3],[(3,10),f4]])
sage: f.which_function(3/2)
x |--> -x + 1
```

piecewise(*list_of_pairs, var=None*)

Returns a piecewise function from a list of (interval, function) pairs.

`list_of_pairs` is a list of pairs (I, fcn), where fcn is a Sage function (such as a polynomial over RR, or functions using the lambda notation), and I is an interval such as I = (1,3). Two consecutive intervals must share a common endpoint.

If the optional `var` is specified, then any symbolic expressions in the list will be converted to symbolic functions using `fcn.function(var)`. (This says which variable is considered to be "piecewise".)

We assume that these definitions are consistent (ie, no checking is done).

EXAMPLES:

```
sage: f1(x) = -1
sage: f2(x) = 2
sage: f = Piecewise([[(0,pi/2),f1],[(pi/2,pi),f2]])
sage: f(1)
-1
sage: f(3)
2
sage: f = Piecewise([[(0,1),x], [(1,2),x^2]], x); f
Piecewise defined function with 2 parts, [[(0, 1), x |--> x], [(1, 2), x |--> x^2]]
sage: f(0.9)
0.900000000000000
sage: f(1.1)
1.21000000000000
```

9.5 Orthogonal Polynomials

This module wraps some of the orthogonal/special functions in the Maxima package "orthopoly". This package was written by Barton Willis of the University of Nebraska at Kearney. It is released under the terms of the General Public License (GPL). Send Maxima-related bug reports and comments on this module to willisb@unk.edu. In your report, please include Maxima and specfun version information.

- The Chebyshev polynomial of the first kind arises as a solution to the differential equation

  $$(1-x^2)\,y'' - x\,y' + n^2\,y = 0$$

  and those of the second kind as a solution to

  $$(1-x^2)\,y'' - 3x\,y' + n(n+2)\,y = 0.$$

  The Chebyshev polynomials of the first kind are defined by the recurrence relation

  $$T_0(x) = 1\,T_1(x) = x\,T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x).$$

  The Chebyshev polynomials of the second kind are defined by the recurrence relation

  $$U_0(x) = 1\,U_1(x) = 2x\,U_{n+1}(x) = 2xU_n(x) - U_{n-1}(x).$$

  For integers m, n, they satisfy the orthogonality relations

  $$\int_{-1}^{1} T_n(x)T_m(x)\,\frac{dx}{\sqrt{1-x^2}} = \begin{cases} 0 & : n \neq m \\ \pi & : n = m = 0 \\ \pi/2 & : n = m \neq 0 \end{cases}$$

  and

  $$\int_{-1}^{1} U_n(x)U_m(x)\sqrt{1-x^2}\,dx = \frac{\pi}{2}\delta_{m,n}.$$

  They are named after Pafnuty Chebyshev (alternative transliterations: Tchebyshef or Tschebyscheff).

- The Hermite polynomials are defined either by

  $$H_n(x) = (-1)^n e^{x^2/2}\frac{d^n}{dx^n}e^{-x^2/2}$$

(the "probabilists' Hermite polynomials"), or by

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}$$

(the "physicists' Hermite polynomials"). Sage (via Maxima) implements the latter flavor. These satisfy the orthogonality relation

$$\int_{-\infty}^{\infty} H_n(x) H_m(x)\, e^{-x^2}\, dx = n! 2^n \sqrt{\pi} \delta_{nm}$$

They are named in honor of Charles Hermite.

- Each *Legendre polynomial* $P_n(x)$ is an n-th degree polynomial. It may be expressed using Rodrigues' formula:

$$P_n(x) = (2^n n!)^{-1} \frac{d^n}{dx^n} \left[(x^2-1)^n\right].$$

These are solutions to Legendre's differential equation:

$$\frac{d}{dx}\left[(1-x^2)\frac{d}{dx}P(x)\right] + n(n+1)P(x) = 0.$$

and satisfy the orthogonality relation

$$\int_{-1}^{1} P_m(x) P_n(x)\, dx = \frac{2}{2n+1}\delta_{mn}$$

The *Legendre function of the second kind* $Q_n(x)$ is another (linearly independent) solution to the Legendre differential equation. It is not an ""orthogonal polynomial"" however.

The associated Legendre functions of the first kind $P_\ell^m(x)$ can be given in terms of the "usual" Legendre polynomials by

$$\begin{aligned} P_\ell^m(x) &= (-1)^m (1-x^2)^{m/2} (d^m)/(dx^m) P_\ell(x) \\ &= \tfrac{(-1)^m}{2^\ell \ell!} (1-x^2)^{m/2} \tfrac{d^{(\ell+m)}}{dx^{(\ell+m)}} (x^2-1)^\ell. \end{aligned}$$

Assuming $0 \le m \le \ell$, they satisfy the orthogonality relation:

$$\int_{-1}^{1} P_k^{(m)} P_\ell^{(m)} dx = \frac{2(\ell+m)!}{(2\ell+1)(\ell-m)!}\, \delta_{k,\ell},$$

where $\delta_{k,\ell}$ is the Kronecker delta.

The associated Legendre functions of the second kind $Q_\ell^m(x)$ can be given in terms of the "usual" Legendre polynomials by

$$Q_\ell^m(x) = (-1)^m (1-x^2)^{m/2} (d^m)/(dx^m) Q_\ell(x).$$

They are named after Adrien-Marie Legendre.

- Laguerre polynomials may be defined by the Rodrigues formula

$$L_n(x) = \frac{e^x}{n!}\frac{d^n}{dx^n}\left(e^{-x}x^n\right).$$

They are solutions of Laguerre's equation:

$$x\, y'' + (1-x)\, y' + n\, y = 0$$

and satisfy the orthogonality relation

$$\int_0^\infty L_m(x)L_n(x)e^{-x}\,dx = \delta_{mn}.$$

The generalized Laguerre polynomials may be defined by the Rodrigues formula:

$$L_n^{(\alpha)}(x) = \frac{x^{-\alpha}e^x}{n!}\frac{d^n}{dx^n}\left(e^{-x}x^{n+\alpha}\right).$$

(These are also sometimes called the associated Laguerre polynomials.) The simple Laguerre polynomials are recovered from the generalized polynomials by setting $\alpha = 0$.

They are named after Edmond Laguerre.

- Jacobi polynomials are a class of orthogonal polynomials. They are obtained from hypergeometric series in cases where the series is in fact finite:

$$P_n^{(\alpha,\beta)}(z) = \frac{(\alpha+1)_n}{n!}\,{}_2F_1\left(-n, 1+\alpha+\beta+n; \alpha+1; \frac{1-z}{2}\right),$$

where $()_n$ is Pochhammer's symbol (for the rising factorial), (Abramowitz and Stegun p561.) and thus have the explicit expression

$$P_n^{(\alpha,\beta)}(z) = \frac{\Gamma(\alpha+n+1)}{n!\Gamma(\alpha+\beta+n+1)}\sum_{m=0}^{n}\binom{n}{m}\frac{\Gamma(\alpha+\beta+n+m+1)}{\Gamma(\alpha+m+1)}\left(\frac{z-1}{2}\right)^m.$$

They are named after Carl Jacobi.

- Ultraspherical or Gegenbauer polynomials are given in terms of the Jacobi polynomials $P_n^(\alpha,\beta))(x)$ with $\alpha = \beta = a - 1/2$ by

$$C_n^(a))(x) = \frac{\Gamma(a+1/2)}{\Gamma(2a)}\frac{\Gamma(n+2a)}{\Gamma(n+a+1/2)}P_n^{(a-1/2,a-1/2)}(x).$$

They satisfy the orthogonality relation

$$\int_(-1)^1 (1-x^2)^{a-1/2}C_m^{(a)}(x)C_n^{(a)}(x)\,dx = \delta_{mn}2^(1-2a)\pi\frac{\Gamma(n+2a)}{(n+a)\Gamma^2(a)\Gamma(n+1)},$$

for $a > -1/2$. They are obtained from hypergeometric series in cases where the series is in fact finite:

$$C_n^{(a)}(z) = \frac{(2a)^{\underline{n}}}{n!}\,{}_2F_1\left(-n, 2a+n; a+\frac{1}{2}; \frac{1-z}{2}\right)$$

where $\underline{n}$ is the falling factorial. (See Abramowitz and Stegun p561)

They are named for Leopold Gegenbauer (1849-1903).

For completeness, the Pochhammer symbol, introduced by Leo August Pochhammer, $(x)_n$, is used in the theory of special functions to represent the "rising factorial" or "upper factorial"

$$(x)_n = x(x+1)(x+2)\cdots(x+n-1) = \frac{(x+n-1)!}{(x-1)!}.$$

On the other hand, the "falling factorial" or "lower factorial" is

$$x^{\underline{n}} = \frac{x!}{(x-n)!},$$

in the notation of Ronald L. Graham, Donald E. Knuth and Oren Patashnik in their book Concrete Mathematics.

Note: The first call of any of these will usually cost a bit extra (it loads "specfun", but I'm not sure if that is the real reason). The next call is usually faster but not always.

TODO: Implement associated Legendre polynomials and Zernike polynomials. (Neither is in Maxima.) http://en.wikipedia.org/wiki/Associated_Legendre_polynomials http://en.wikipedia.org/wiki/Zernike_polynomials

REFERENCES:

- Abramowitz and Stegun: Handbook of Mathematical Functions, http://www.math.sfu.ca/ cbm/aands/
- http://en.wikipedia.org/wiki/Chebyshev_polynomials
- http://en.wikipedia.org/wiki/Legendre_polynomials
- http://en.wikipedia.org/wiki/Hermite_polynomials
- http://mathworld.wolfram.com/GegenbauerPolynomial.html
- http://en.wikipedia.org/wiki/Jacobi_polynomials
- http://en.wikipedia.org/wiki/Laguerre_polynomia
- http://en.wikipedia.org/wiki/Associated_Legendre_polynomials

AUTHORS:

- David Joyner (2006-06)

chebyshev_T(*n, x*)

Returns the Chebyshev function of the first kind for integers $n > -1$.

REFERENCE:

•AS 22.5.31 page 778 and AS 6.1.22 page 256.

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: chebyshev_T(2,x)
2*x^2 - 1
```

chebyshev_U(*n, x*)

Returns the Chebyshev function of the second kind for integers $n > -1$.

REFERENCE:

•AS, 22.8.3 page 783 and AS 6.1.22 page 256.

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: chebyshev_U(2,x)
4*x^2 - 1
```

gegenbauer(*n, a, x*)

Returns the ultraspherical (or Gegenbauer) polynomial for integers $n > -1$.

Computed using Maxima.

REFERENCE:

•AS 22.5.27

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: ultraspherical(2,3/2,x)
15/2*x^2 - 3/2
sage: ultraspherical(2,1/2,x)
3/2*x^2 - 1/2
sage: ultraspherical(1,1,x)
2*x
sage: t = PolynomialRing(RationalField(),"t").gen()
sage: gegenbauer(3,2,t)
32*t^3 - 12*t
```

gen_laguerre(*n, a, x*)

Returns the generalized Laguerre polynomial for integers $n > -1$. Typically, a = 1/2 or a = -1/2.

REFERENCE:

- table on page 789 in AS.

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: gen_laguerre(2,1,x)
1/2*x^2 - 3*x + 3
sage: gen_laguerre(2,1/2,x)
1/2*x^2 - 5/2*x + 15/8
sage: gen_laguerre(2,-1/2,x)
1/2*x^2 - 3/2*x + 3/8
sage: gen_laguerre(2,0,x)
1/2*x^2 - 2*x + 1
sage: gen_laguerre(3,0,x)
-1/6*x^3 + 3/2*x^2 - 3*x + 1
```

gen_legendre_P(*n, m, x*)

Returns the generalized (or associated) Legendre function of the first kind for integers $n > -1, m > -1$.

The awkward code for when m is odd and 1 results from the fact that Maxima is happy with, for example, $(1 - t^2)^3/2$, but Sage is not. For these cases the function is computed from the (m-1)-case using one of the recursions satisfied by the Legendre functions.

REFERENCE:

- Gradshteyn and Ryzhik 8.706 page 1000.

EXAMPLES:

```
sage: P.<t> = QQ[]
sage: gen_legendre_P(2, 0, t)
3/2*t^2 - 1/2
sage: gen_legendre_P(2, 0, t) == legendre_P(2, t)
True
sage: gen_legendre_P(3, 1, t)
-3/2*sqrt(-t^2 + 1)*(5*t^2 - 1)
sage: gen_legendre_P(4, 3, t)
(105*t^3 - 105*t)*sqrt(-t^2 + 1)
sage: gen_legendre_P(7, 3, I).expand()
-16695*sqrt(2)
sage: gen_legendre_P(4, 1, 2.5)
-583.562373654533*I
```

gen_legendre_Q(*n, m, x*)

Returns the generalized (or associated) Legendre function of the second kind for integers $n > -1$, $m > -1$.

Maxima restricts m = n. Hence the cases m n are computed using the same recursion used for gen_legendre_P(n,m,x) when m is odd and 1.

EXAMPLES:

```
sage: P.<t> = QQ[]
sage: gen_legendre_Q(2,0,t)
3/4*t^2*log((-t - 1)/(t - 1)) - 3/2*t - 1/4*log((-t - 1)/(t - 1))
sage: gen_legendre_Q(2,0,t) - legendre_Q(2, t)
0
sage: gen_legendre_Q(3,1,0.5)
2.49185259170895
sage: gen_legendre_Q(0, 1, x)
-1/sqrt(-x^2 + 1)
sage: gen_legendre_Q(2, 4, x).factor()
48*x/((x - 1)^2*(x + 1)^2)
```

hermite(*n, x*)

Returns the Hermite polynomial for integers $n > -1$.

REFERENCE:

- AS 22.5.40 and 22.5.41, page 779.

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: hermite(2,x)
4*x^2 - 2
sage: hermite(3,x)
8*x^3 - 12*x
sage: hermite(3,2)
40
sage: S.<y> = PolynomialRing(RR)
sage: hermite(3,y)
8.00000000000000*y^3 - 12.0000000000000*y
sage: R.<x,y> = QQ[]
sage: hermite(3,y^2)
8*y^6 - 12*y^2
sage: w = var('w')
sage: hermite(3,2*w)
8*(8*w^2 - 3)*w
```

jacobi_P(*n, a, b, x*)

Returns the Jacobi polynomial $P_n^{(a,b)}(x)$ for integers $n > -1$ and a and b symbolic or $a > -1$ and $b > -1$. The Jacobi polynomials are actually defined for all a and b. However, the Jacobi polynomial weight $(1-x)^a(1+x)^b$ isn't integrable for $a \leq -1$ or $b \leq -1$.

REFERENCE:

- table on page 789 in AS.

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: jacobi_P(2,0,0,x)
3/2*x^2 - 1/2
sage: jacobi_P(2,1,2,1.2)        # random output of low order bits
5.009999999999998
```

laguerre (*n, x*)

Returns the Laguerre polynomial for integers $n > -1$.

REFERENCE:

- AS 22.5.16, page 778 and AS page 789.

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: laguerre(2,x)
1/2*x^2 - 2*x + 1
sage: laguerre(3,x)
-1/6*x^3 + 3/2*x^2 - 3*x + 1
sage: laguerre(2,2)
-1
```

legendre_P (*n, x*)

Returns the Legendre polynomial of the first kind for integers $n > -1$.

REFERENCE:

- AS 22.5.35 page 779.

EXAMPLES:

```
sage: P.<t> = QQ[]
sage: legendre_P(2,t)
3/2*t^2 - 1/2
sage: legendre_P(3, 1.1)
1.67750000000000
sage: legendre_P(3, 1 + I)
7/2*I - 13/2
sage: legendre_P(3, MatrixSpace(ZZ, 2)([1, 2, -4, 7]))
[-179  242]
[-484  547]
sage: legendre_P(3, GF(11)(5))
8
```

legendre_Q (*n, x*)

Returns the Legendre function of the second kind for integers $n > -1$.

Computed using Maxima.

EXAMPLES:

```
sage: P.<t> = QQ[]
sage: legendre_Q(2, t)
3/4*t^2*log((-t - 1)/(t - 1)) - 3/2*t - 1/4*log((-t - 1)/(t - 1))
sage: legendre_Q(3, 0.5)
-0.198654771479482
sage: legendre_Q(4, 2)
443/16*I*pi + 443/16*log(3) - 365/12
sage: legendre_Q(4, 2.0)
NaN
```

ultraspherical (*n, a, x*)

Returns the ultraspherical (or Gegenbauer) polynomial for integers $n > -1$.

Computed using Maxima.

REFERENCE:

•AS 22.5.27

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: ultraspherical(2,3/2,x)
15/2*x^2 - 3/2
sage: ultraspherical(2,1/2,x)
3/2*x^2 - 1/2
sage: ultraspherical(1,1,x)
2*x
sage: t = PolynomialRing(RationalField(),"t").gen()
sage: gegenbauer(3,2,t)
32*t^3 - 12*t
```

9.6 Special Functions

AUTHORS:

- David Joyner (2006-13-06): initial version
- David Joyner (2006-30-10): bug fixes to pari wrappers of Bessel functions, hypergeometric_U
- William Stein (2008-02): Impose some sanity checks.
- David Joyner (2008-04-23): addition of elliptic integrals

This module provides easy access to many of Maxima and PARI's special functions.

Maxima's special functions package (which includes spherical harmonic functions, spherical Bessel functions (of the 1st and 2nd kind), and spherical Hankel functions (of the 1st and 2nd kind)) was written by Barton Willis of the University of Nebraska at Kearney. It is released under the terms of the General Public License (GPL).

Support for elliptic functions and integrals was written by Raymond Toy. It is placed under the terms of the General Public License (GPL) that governs the distribution of Maxima.

The (usual) Bessel functions and Airy functions are part of the standard Maxima package. Some Bessel functions also are implemented in Pari. (Caution: The Pari versions are sometimes different than the Maxima version.) For example, the K-Bessel function $K_\nu(z)$ can be computed using either Maxima or Pari, depending on an optional variable you pass to bessel_K.

Next, we summarize some of the properties of the functions implemented here.

- Bessel functions, first defined by the Swiss mathematician Daniel Bernoulli and named after Friedrich Bessel, are canonical solutions y(x) of Bessel's differential equation:

$$x^2\frac{d^2y}{dx^2} + x\frac{dy}{dx} + (x^2 - \alpha^2)y = 0,$$

  for an arbitrary real number α (the order).

- Another important formulation of the two linearly independent solutions to Bessel's equation are the Hankel functions $H_\alpha^{(1)}(x)$ and $H_\alpha^{(2)}(x)$, defined by:

$$H_\alpha^{(1)}(x) = J_\alpha(x) + iY_\alpha(x)$$

$$H_\alpha^{(2)}(x) = J_\alpha(x) - iY_\alpha(x)$$

where i is the imaginary unit (and J_* and Y_* are the usual J- and Y-Bessel functions). These linear combinations are also known as Bessel functions of the third kind; they are two linearly independent solutions of Bessel's differential equation. They are named for Hermann Hankel.

- Airy function The function $Ai(x)$ and the related function $Bi(x)$, which is also called an Airy function, are solutions to the differential equation

$$y'' - xy = 0,$$

known as the Airy equation. They belong to the class of 'Bessel functions of fractional order'. The initial conditions $Ai(0) = (\Gamma(2/3)3^{2/3})^{-1}$, $Ai'(0) = -(\Gamma(1/3)3^{1/3})^{-1}$ define $Ai(x)$. The initial conditions $Bi(0) = 3^{1/2}Ai(0)$, $Bi'(0) = -3^{1/2}Ai'(0)$ define $Bi(x)$.

They are named after the British astronomer George Biddell Airy.

- Spherical harmonics: Laplace's equation in spherical coordinates is:

$$\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial f}{\partial r}\right) + \frac{1}{r^2}\sin\theta\frac{\partial}{\partial\theta}\left(\sin\theta\frac{\partial f}{\partial\theta}\right) + \frac{1}{r^2\sin^2\theta}\frac{\partial^2 f}{\partial\varphi^2} = 0.$$

Note that the spherical coordinates θ and φ are defined here as follows: θ is the colatitude or polar angle, ranging from $0 \le \theta \le \pi$ and φ the azimuth or longitude, ranging from $0 \le \varphi < 2\pi$.

The general solution which remains finite towards infinity is a linear combination of functions of the form

$$r^{-1-\ell}\cos(m\varphi)P_\ell^m(\cos\theta)$$

and

$$r^{-1-\ell}\sin(m\varphi)P_\ell^m(\cos\theta)$$

where P_ℓ^m are the associated Legendre polynomials, and with integer parameters $\ell \ge 0$ and m from 0 to ℓ. Put in another way, the solutions with integer parameters $\ell \ge 0$ and $-\ell \le m \le \ell$, can be written as linear combinations of:

$$U_{\ell,m}(r,\theta,\varphi) = r^{-1-\ell}Y_\ell^m(\theta,\varphi)$$

where the functions Y are the spherical harmonic functions with parameters ℓ, m, which can be written as:

$$Y_\ell^m(\theta,\varphi) = \sqrt{\frac{(2\ell+1)}{4\pi}\frac{(\ell-m)!}{(\ell+m)!}}\cdot e^{im\varphi}\cdot P_\ell^m(\cos\theta).$$

The spherical harmonics obey the normalisation condition

$$\int_{\theta=0}^{\pi}\int_{\varphi=0}^{2\pi} Y_\ell^m Y_{\ell'}^{m'*}\,d\Omega = \delta_{\ell\ell'}\delta_{mm'} \qquad d\Omega = \sin\theta\,d\varphi\,d\theta.$$

- When solving for separable solutions of Laplace's equation in spherical coordinates, the radial equation has the form:

$$x^2\frac{d^2y}{dx^2} + 2x\frac{dy}{dx} + [x^2 - n(n+1)]y = 0.$$

The spherical Bessel functions j_n and y_n, are two linearly independent solutions to this equation. They are related to the ordinary Bessel functions J_n and Y_n by:

$$j_n(x) = \sqrt{\frac{\pi}{2x}}J_{n+1/2}(x),$$

$$y_n(x) = \sqrt{\frac{\pi}{2x}}Y_{n+1/2}(x) = (-1)^{n+1}\sqrt{\frac{\pi}{2x}}J_{-n-1/2}(x).$$

- For $x > 0$, the confluent hypergeometric function $y = U(a, b, x)$ is defined to be the solution to Kummer's differential equation

$$xy'' + (b - x)y' - ay = 0,$$

which satisfies $U(a, b, x) \sim x^{-a}$, as $x \to \infty$. (There is a linearly independent solution, called Kummer's function $M(a, b, x)$, which is not implemented.)

- Jacobi elliptic functions can be thought of as generalizations of both ordinary and hyperbolic trig functions. There are twelve Jacobian elliptic functions. Each of the twelve corresponds to an arrow drawn from one corner of a rectangle to another.

Each of the corners of the rectangle are labeled, by convention, s, c, d and n. The rectangle is understood to be lying on the complex plane, so that s is at the origin, c is on the real axis, and n is on the imaginary axis. The twelve Jacobian elliptic functions are then pq(x), where p and q are one of the letters s,c,d,n.

The Jacobian elliptic functions are then the unique doubly-periodic, meromorphic functions satisfying the following three properties:

1. There is a simple zero at the corner p, and a simple pole at the corner q.
2. The step from p to q is equal to half the period of the function pq(x); that is, the function pq(x) is periodic in the direction pq, with the period being twice the distance from p to q. Also, pq(x) is also periodic in the other two directions as well, with a period such that the distance from p to one of the other corners is a quarter period.
3. If the function pq(x) is expanded in terms of x at one of the corners, the leading term in the expansion has a coefficient of 1. In other words, the leading term of the expansion of pq(x) at the corner p is x; the leading term of the expansion at the corner q is 1/x, and the leading term of an expansion at the other two corners is 1.

We can write

$$pq(x) = \frac{pr(x)}{qr(x)}$$

where p, q, and r are any of the letters s, c, d, n, with the understanding that $ss = cc = dd = nn = 1$.

Let

$$u = \int_0^\phi \frac{d\theta}{\sqrt{1 - m\sin^2\theta}}$$

Then the *Jacobi elliptic function* $sn(u)$ is given by

$$sn\ u = \sin\phi$$

and $cn(u)$ is given by

$$cn\ u = \cos\phi$$

and

$$dn\ u = \sqrt{1 - m\sin^2\phi}.$$

To emphasize the dependence on m, one can write $sn(u, m)$ for example (and similarly for cn and dn). This is the notation used below.

For a given k with $0 < k < 1$ they therefore are solutions to the following nonlinear ordinary differential equations:

– $\mathrm{sn}\,(x;k)$ solves the differential equations

$$\frac{\mathrm{d}^2y}{\mathrm{d}x^2} + (1+k^2)y - 2k^2y^3 = 0,$$

and

$$\left(\frac{\mathrm{d}y}{\mathrm{d}x}\right)^2 = (1-y^2)(1-k^2y^2).$$

– $\mathrm{cn}\,(x;k)$ solves the differential equations

$$\frac{\mathrm{d}^2y}{\mathrm{d}x^2} + (1-2k^2)y + 2k^2y^3 = 0,$$

and $\left(\frac{\mathrm{d}y}{\mathrm{d}x}\right)^2 = (1-y^2)(1-k^2+k^2y^2)$.

– $\mathrm{dn}\,(x;k)$ solves the differential equations

$$\frac{\mathrm{d}^2y}{\mathrm{d}x^2} - (2-k^2)y + 2y^3 = 0,$$

and $\left(\frac{\mathrm{d}y}{\mathrm{d}x}\right)^2 = y^2(1-k^2-y^2)$.

If $K(m)$ denotes the complete elliptic integral of the first kind (denoted `elliptic_kc`), the elliptic functions $sn(x,m)$ and $cn(x,m)$ have real periods $4K(m)$, whereas $dn(x,m)$ has a period $2K(m)$. The limit $m \to 0$ gives $K(0) = \pi/2$ and trigonometric functions: $sn(x,0) = \sin x$, $cn(x,0) = \cos x$, $dn(x,0) = 1$. The limit $m \to 1$ gives $K(1) \to \infty$ and hyperbolic functions: $sn(x,1) = \tanh x$, $cn(x,1) = \mathrm{sech} x$, $dn(x,1) = \mathrm{sech} x$.

– The incomplete elliptic integrals (of the first kind, etc.) are:

$$\int_0^\phi \frac{1}{\sqrt{1-m\sin(x)^2}}\,dx,$$
$$\int_0^\phi \sqrt{1-m\sin(x)^2}\,dx,$$
$$\int_0^\phi \frac{\sqrt{1-mt^2}}{\sqrt{(1-t^2)}}\,dx,$$
$$\int_0^\phi \frac{1}{\sqrt{1-m\sin(x)^2}\sqrt{1-n\sin(x)^2}}\,dx,$$

and the complete ones are obtained by taking $\phi = \pi/2$.

REFERENCES:

- Abramowitz and Stegun: Handbook of Mathematical Functions, http://www.math.sfu.ca/~cbm/aands/
- http://en.wikipedia.org/wiki/Bessel_function
- http://en.wikipedia.org/wiki/Airy_function
- http://en.wikipedia.org/wiki/Spherical_harmonics
- http://en.wikipedia.org/wiki/Helmholtz_equation
- http://en.wikipedia.org/wiki/Jacobi's_elliptic_functions
- A. Khare, U. Sukhatme, 'Cyclic Identities Involving Jacobi Elliptic Functions', Math ArXiv, math-ph/0201004

- Online Encyclopedia of Special Function http://algo.inria.fr/esf/index.html

TODO: Resolve weird bug in commented out code in hypergeometric_U below.

AUTHORS:

- David Joyner and William Stein

Added 16-02-2008 (wdj): optional calls to scipy and replace all '#random' by '...' (both at the request of William Stein)

Warning: SciPy's versions are poorly documented and seem less accurate than the Maxima and Pari versions.

class Bessel(*nu, typ='J', algorithm='pari', prec=53*)

A class implementing the I, J, K, and Y Bessel functions.

EXAMPLES:

```
sage: g = Bessel(2); g
J_{2}
sage: print g
J-Bessel function of order 2
sage: g.plot(0,10)
```

order()

plot(*a, b*)

prec()

system()

type()

class EllipticE()

class EllipticEC()

class EllipticEU()

class EllipticF()

class EllipticKC()

class EllipticPi()

class MaximaFunction(*name, nargs=2, conversions={}*)

airy_ai(*x*)

The function $Ai(x)$ and the related function $Bi(x)$, which is also called an *Airy function*, are solutions to the differential equation

$$y'' - xy = 0,$$

known as the *Airy equation*. The initial conditions $Ai(0) = (\Gamma(2/3)3^{2/3})^{-1}$, $Ai'(0) = -(\Gamma(1/3)3^{1/3})^{-1}$ define $Ai(x)$. The initial conditions $Bi(0) = 3^{1/2}Ai(0)$, $Bi'(0) = -3^{1/2}Ai'(0)$ define $Bi(x)$.

They are named after the British astronomer George Biddell Airy. They belong to the class of "Bessel functions of fractional order".

EXAMPLES:

```
sage: airy_ai(1.0)        # last few digits are random
0.135292416312881400
sage: airy_bi(1.0)        # last few digits are random
1.20742359495287099
```

REFERENCE:

- Abramowitz and Stegun: Handbook of Mathematical Functions, http://www.math.sfu.ca/~cbm/aands/
- http://en.wikipedia.org/wiki/Airy_function

airy_bi (x)

The function $Ai(x)$ and the related function $Bi(x)$, which is also called an *Airy function*, are solutions to the differential equation

$$y'' - xy = 0,$$

known as the *Airy equation*. The initial conditions $Ai(0) = (\Gamma(2/3)3^{2/3})^{-1}$, $Ai'(0) = -(\Gamma(1/3)3^{1/3})^{-1}$ define $Ai(x)$. The initial conditions $Bi(0) = 3^{1/2}Ai(0)$, $Bi'(0) = -3^{1/2}Ai'(0)$ define $Bi(x)$.

They are named after the British astronomer George Biddell Airy. They belong to the class of "Bessel functions of fractional order".

EXAMPLES:

```
sage: airy_ai(1)         # last few digits are random
0.135292416312881400
sage: airy_bi(1)         # last few digits are random
1.20742359495287099
```

REFERENCE:

- Abramowitz and Stegun: Handbook of Mathematical Functions, http://www.math.sfu.ca/~cbm/aands/
- http://en.wikipedia.org/wiki/Airy_function

bessel_I (*nu, z, algorithm='pari', prec=53*)

Implements the "I-Bessel function", or "modified Bessel function, 1st kind", with index (or "order") nu and argument z.

INPUT:

- nu - a real (or complex, for pari) number
- z - a real (positive) algorithm - "pari" or "maxima" or "scipy" prec - real precision (for Pari only)

DEFINITION:

Maxima:

```
 inf
====   - nu - 2 k   nu + 2 k
\     2           z
 >    -------------------
/     k! Gamma(nu + k + 1)
====
k = 0
```

Pari:

```
 inf
====   - 2 k   2 k
\     2       z     Gamma(nu + 1)
 >    -----------------------
/       k! Gamma(nu + k + 1)
====
k = 0
```

Sometimes `bessel_I(nu,z)` is denoted `I_nu(z)` in the literature.

Warning: In Maxima (the manual says) i0 is deprecated but `bessel_i(0,*)` is broken. (Was fixed in recent CVS patch though.)

EXAMPLES:

```
sage: bessel_I(1,1,"pari",500)
0.565159103992485027207696027609863307328899621621092009480294489479255640964371134092664997766
sage: bessel_I(1,1)
0.565159103992485
sage: bessel_I(2,1.1,"maxima")
0.16708949925104...
sage: bessel_I(0,1.1,"maxima")
1.32616018371265...
sage: bessel_I(0,1,"maxima")
1.26606587775200...
sage: bessel_I(1,1,"scipy")
0.565159103992...
```

bessel_J(*nu, z, algorithm='pari', prec=53*)

Return value of the "J-Bessel function", or "Bessel function, 1st kind", with index (or "order") nu and argument z.

```
Defn:
Maxima:
                  inf
                 ====           - nu - 2 k  nu + 2 k
                 \      (-1)^k 2           z
                  >     ------------------------
                 /         k! Gamma(nu + k + 1)
                 ====
                 k = 0

Pari:

                  inf
                 ====           - 2k    2k
                 \      (-1)^k 2      z    Gamma(nu + 1)
                  >     ---------------------------
                 /          k! Gamma(nu + k + 1)
                 ====
                 k = 0
```

Sometimes bessel_J(nu,z) is denoted J_nu(z) in the literature.

Warning: Inaccurate for small values of z.

EXAMPLES:

```
sage: bessel_J(2,1.1)
0.136564153956658
sage: bessel_J(0,1.1)
0.719622018527511
sage: bessel_J(0,1)
0.765197686557967
sage: bessel_J(0,0)
1.00000000000000
```

```
sage: bessel_J(0.1,0.1)
0.777264368097005
```

We check consistency of PARI and Maxima:

```
sage: n(bessel_J(3,10,"maxima"))
0.0583793793051...
sage: n(bessel_J(3,10,"pari"))
0.0583793793051868
sage: bessel_J(3,10,"scipy")
0.0583793793052...
```

bessel_K(*nu, z, algorithm='pari', prec=53*)

Implements the "K-Bessel function", or "modified Bessel function, 2nd kind", with index (or "order") nu and argument z. Defn:

```
 pi*(bessel_I(-nu, z) - bessel_I(nu, z))
-----------------------------------------
             2*sin(pi*nu)
```

if nu is not an integer and by taking a limit otherwise.

Sometimes bessel_K(nu,z) is denoted K_nu(z) in the literature. In Pari, nu can be complex and z must be real and positive.

EXAMPLES:

```
sage: bessel_K(3,2,"scipy")
0.64738539094...
sage: bessel_K(3,2)
0.64738539094...
sage: bessel_K(1,1)
0.60190723019...
sage: bessel_K(1,1,"pari",10)
0.60
sage: bessel_K(1,1,"pari",100)
0.60190723019723457473754000154
```

bessel_Y(*nu, z, algorithm='maxima', prec=53*)

Implements the "Y-Bessel function", or "Bessel function of the 2nd kind", with index (or "order") nu and argument z.

Note: Currently only prec=53 is supported.

Defn:

```
 cos(pi n)*bessel_J(nu, z) - bessel_J(-nu, z)
-----------------------------------------------
                  sin(nu*pi)
```

if nu is not an integer and by taking a limit otherwise.

Sometimes bessel_Y(n,z) is denoted Y_n(z) in the literature.

This is computed using Maxima by default.

EXAMPLES:

```
sage: bessel_Y(2,1.1,"scipy")
-1.4314714939...
sage: bessel_Y(2,1.1)
-1.4314714939590...
sage: bessel_Y(3.001,2.1)
-1.0299574976424...
```

Note: Adding '0'+ inside sage_eval as a temporary bug work-around.

error_fcn(*t*)
The complementary error function $\frac{2}{\sqrt{\pi}} \int_t^\infty e^{-x^2} dx$ (t belongs to RR).

exp_int(*t*)
The exponential integral $\int_x^\infty e^{-x}/x dx$ (t belongs to RR).

hypergeometric_U(*alpha, beta, x, algorithm='pari', prec=53*)
Default is a wrap of Pari's hyperu(alpha,beta,x) function. Optionally, algorithm = "scipy" can be used.

The confluent hypergeometric function $y = U(a, b, x)$ is defined to be the solution to Kummer's differential equation

$$xy'' + (b - x)y' - ay = 0.$$

This satisfies $U(a, b, x) \sim x^{-a}$, as $x \to \infty$, and is sometimes denoted `x^{-a}2_F_0(a,1+a-b,-1/x)`. This is not the same as Kummer's M-hypergeometric function, denoted sometimes as `_1F_1(alpha,beta,x)`, though it satisfies the same DE that U does.

Warning: In the literature, both are called "Kummer confluent hypergeometric" functions.

EXAMPLES:

```
sage: hypergeometric_U(1,1,1,"scipy")
0.596347362323...
sage: hypergeometric_U(1,1,1)
0.59634736232319...
sage: hypergeometric_U(1,1,1,"pari",70)
0.59634736232319407434...
```

inverse_jacobi(*sym, x, m*)
Here sym = "pq", where p,q in c,d,n,s. This returns the value of the inverse Jacobi function $pq^{-1}(x, m)$. There are a total of 12 functions described by this.

EXAMPLES:

```
sage: jacobi("sn",1/2,1/2)
jacobi_sn(1/2, 1/2)
sage: float(jacobi("sn",1/2,1/2))
0.4707504736556572
sage: float(inverse_jacobi("sn",0.47,1/2))
0.49909823132221959
sage: float(inverse_jacobi("sn",0.4707504,0.5))
0.49999991146655481
sage: P = plot(inverse_jacobi('sn', x, 0.5), 0, 1, plot_points=20)
```

Now to view this, just type show(P).

jacobi(*sym, x, m*)
Here sym = "pq", where p,q in c,d,n,s. This returns the value of the Jacobi function pq(x,m), as described in the documentation for Sage's "special" module. There are a total of 12 functions described by this.

EXAMPLES:

```
sage: jacobi("sn",1,1)
tanh(1)
sage: jacobi("cd",1,1/2)
jacobi_cd(1, 1/2)
sage: RDF(jacobi("cd",1,1/2))
0.724009721659
sage: RDF(jacobi("cn",1,1/2)); RDF(jacobi("dn",1,1/2)); RDF(jacobi("cn",1,1/2)/jacobi("dn",1,1/2
0.595976567672
0.823161001632
0.724009721659
sage: jsn = jacobi("sn",x,1)
sage: P = plot(jsn,0,1)
```

To view this, type P.show().

lngamma(*t*)
The principal branch of the logarithm of the Gamma function of t.

meval(*x*)

spherical_bessel_J(*n, var, algorithm='maxima'*)
Returns the spherical Bessel function of the first kind for integers n -1.

Reference: AS 10.1.8 page 437 and AS 10.1.15 page 439.

EXAMPLES:

```
sage: spherical_bessel_J(2,x)
((3/x^2 - 1)*sin(x) - 3*cos(x)/x)/x
```

spherical_bessel_Y(*n, var, algorithm='maxima'*)
Returns the spherical Bessel function of the second kind for integers n -1.

Reference: AS 10.1.9 page 437 and AS 10.1.15 page 439.

EXAMPLES:

```
sage: x = PolynomialRing(QQ, 'x').gen()
sage: spherical_bessel_Y(2,x)
-((3/x^2 - 1)*cos(x) + 3*sin(x)/x)/x
```

spherical_hankel1(*n, var*)
Returns the spherical Hankel function of the first kind for integers $n > -1$, written as a string. Reference: AS 10.1.36 page 439.

EXAMPLES:

```
sage: spherical_hankel1(2, x)
(I*x^2 - 3*x - 3*I)*e^(I*x)/x^3
```

spherical_hankel2(*n, x*)
Returns the spherical Hankel function of the second kind for integers n -1, written as a string. Reference: AS 10.1.17 page 439.

EXAMPLES:

```
sage: spherical_hankel2(2, x)
(-I*x^2 - 3*x + 3*I)*e^(-I*x)/x^3
```

Here I = sqrt(-1).

spherical_harmonic(*m, n, x, y*)

Returns the spherical Harmonic function of the second kind for integers $n > -1$, $|m| \leq n$. Reference: Merzbacher 9.64.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: spherical_harmonic(3,2,x,y)
1/8*sqrt(7)*sqrt(30)*e^(2*I*y)*sin(x)^2*cos(x)/sqrt(pi)
sage: spherical_harmonic(3,2,1,2)
1/8*sqrt(7)*sqrt(30)*e^(4*I)*sin(1)^2*cos(1)/sqrt(pi)
```

CHAPTER

TEN

BASIC STRUCTURES

10.1 Abstract base class for Sage objects

class SageObject()

category()

db()

Dumps self into the Sage database. Use db(name) by itself to reload.

The database directory is `$HOME/.sage/db`

dump()

Same as self.save(filename, compress)

dumps()

Dump self to a string s, which can later be reconstituted as self using loads(s).

rename()

Change self so it prints as x, where x is a string.

Note: This is *only* supported for Python classes that derive from SageObject.

EXAMPLES:

```
sage: x = PolynomialRing(QQ,'x').gen()
sage: g = x^3 + x - 5
sage: g
x^3 + x - 5
sage: g.rename('a polynomial')
sage: g
a polynomial
sage: g + x
x^3 + 2*x - 5
sage: h = g^100
sage: str(h)[:20]
'x^300 + 100*x^298 - '
sage: h.rename('x^300 + ...')
sage: h
x^300 + ...
```

Real numbers are not Python classes, so rename is not supported:

```
sage: a = 3.14
sage: type(a)
<type 'sage.rings.real_mpfr.RealLiteral'>
sage: a.rename('pi')
```

```
...
NotImplementedError: object does not support renaming: 3.14000000000000
```

Note: The reason C-extension types are not supported by default is if they were then every single one would have to carry around an extra attribute, which would be slower and waste a lot of memory.

To support them for a specific class, add a `cdef public __custom_name` attribute.

reset_name()

save()

Save self to the given filename.

EXAMPLES:

```
sage: f = x^3 + 5
sage: f.save(SAGE_TMP + '/file')
sage: load(SAGE_TMP + '/file.sobj')
x^3 + 5
```

version()

The version of Sage.

Call this to save the version of Sage in this object. If you then save and load this object it will know in what version of Sage it was created.

This only works on Python classes that derive from SageObject.

dumps()

Dump obj to a string s. To recover obj, use `loads(s)`.

See Also:

`dumps()`

EXAMPLES:

```
sage: a = 2/3
sage: s = dumps(a)
sage: print len(s)
49
sage: loads(s)
2/3
```

load()

Load Sage object from the file with name filename, which will have an .sobj extension added if it doesn't have one.

Note: There is also a special Sage command (that is not available in Python) called load that you use by typing

```
sage: load filename.sage           # not tested
```

The documentation below is not for that command. The documentation for load is almost identical to that for attach. Type attach? for help on attach.

This also loads a ".sobj" file over a network by specifying the full URL. (Setting "verbose = False" suppresses the loading progress indicator.)

EXAMPLE:

```
sage: u = 'http://sage.math.washington.edu/home/was/db/test.sobj'
sage: s = load(u)                                                  # optional - internet
Attempting to load remote file: http://sage.math.washington.edu/home/was/db/test.sobj
Loading: [.]
sage: s                                                            # optional - internet
'hello SAGE'
```

loads()

Recover an object x that has been dumped to a string s using `s = dumps(x)`.

See Also:

`dumps()`

EXAMPLES:

```
sage: a = matrix(2, [1,2,3,-4/3])
sage: s = dumps(a)
sage: loads(s)
[   1    2]
[   3 -4/3]
```

If compress is True (the default), it will try to decompress the data with zlib and with bz2 (in turn); if neither succeeds, it will assume the data is actually uncompressed. If compress=False is explicitly specified, then no decompression is attempted.

```
sage: v = [1..10]
sage: loads(dumps(v, compress=False)) == v
True
sage: loads(dumps(v, compress=False), compress=True) == v
True
sage: loads(dumps(v, compress=True), compress=False)
...
UnpicklingError: invalid load key, 'x'.
```

picklejar()

Create pickled sobj of obj in dir, with name the absolute value of the hash of the pickle of obj. This is used in conjection with sage.structure.sage_object.unpickle_all.

To use this to test the whole Sage library right now, set the environment variable SAGE_PICKLE_JAR, which will make it so dumps will by default call picklejar with the default dir. Once you do that and doctest Sage, you'll find that the SAGE_ROOT/tmp/ contains a bunch of pickled objects along with corresponding txt descriptions of them. Use the sage.structure.sage_object.unpickle_all to see if they unpickle later.

INPUTS:

- `obj` - a pickleable object
- `dir` - a string or None; if None defaults to `SAGE_ROOT/tmp/pickle_jar-version`

EXAMPLES:

```
sage: dir = tmp_dir()
sage: sage.structure.sage_object.picklejar(1,dir)
sage: len(os.listdir(dir))
2
```

register_unpickle_override()

Python pickles include the module and class name of classes. This means that rearranging the Sage source can invalidate old pickles. To keep the old pickles working, you can call register_unpickle_override with an old module name and class name, and the Python callable (function, class with __call__ method, etc.) to use for unpickling. (If this callable is a value in some module, you can specify the module name and class name, for the benefit of explain_pickle(..., in_current_sage=True).)

EXAMPLES:

```
sage: from sage.structure.sage_object import unpickle_override, register_unpickle_override
sage: unpickle_global('sage.rings.integer', 'Integer')
<type 'sage.rings.integer.Integer'>
```

Now we horribly break the pickling system:

sage: register_unpickle_override('sage.rings.integer', 'Integer', Rational, call_name=('sage.rings.rational', 'Rational')) sage: unpickle_global('sage.rings.integer', 'Integer') <type 'sage.rings.rational.Rational'>

And we reach into the internals and put it back:

sage: del unpickle_override[('sage.rings.integer', 'Integer')] sage: unpickle_global('sage.rings.integer', 'Integer') <type 'sage.rings.integer.Integer'>

save()

Save obj to the file with name filename, which will have an .sobj extension added if it doesn't have one. This will *replace* the contents of filename.

EXAMPLES:

```
sage: a = matrix(2, [1,2,3,-5/2])
sage: save(a, 'test.sobj')
sage: load('test')
[   1    2]
[   3 -5/2]
sage: E = EllipticCurve([-1,0])
sage: P = plot(E)
sage: save(P, 'test')
sage: save(P, filename="sage.png", xmin=-2)
sage: print load('test.sobj')
Graphics object consisting of 2 graphics primitives
sage: save("A python string", './test')
sage: load('./test.sobj')
'A python string'
sage: load('./test')
'A python string'
```

unpickle_all()

Unpickle all sobj's in the given directory, reporting failures as they occur. Also printed the number of successes and failure.

INPUT:

- `dir` - string; a directory or name of a .tar.bz2 file that decompresses to give a directory full of pickles.

EXAMPLES:

```
sage: dir = tmp_dir()
sage: sage.structure.sage_object.picklejar('hello', dir)
sage: sage.structure.sage_object.unpickle_all(dir)
Successfully unpickled 1 objects.
Failed to unpickle 0 objects.
```

We unpickle the standard pickle jar. This doctest tests that all "standard pickles" unpickle. Every so often the standard pickle jar should be updated by running the doctest suite with the environment variable SAGE_PICKLE_JAR set, then copying the files from SAGE_ROOT/tmp/pickle_jar* into the standard pickle jar.

```
sage: std = os.environ['SAGE_DATA'] + '/extcode/pickle_jar/pickle_jar.tar.bz2'
sage: sage.structure.sage_object.unpickle_all(std)
doctest:...: DeprecationWarning: RQDF is deprecated; use RealField(212) instead.
Successfully unpickled 572 objects.
Failed to unpickle 0 objects.
```

unpickle_global()

Given a module name and a name within that module (typically a class name), retrieve the corresponding object. This normally just looks up the name in the module, but it can be overridden by register_unpickle_override. This is used in the Sage unpickling mechanism, so if the Sage source code organization changes, register_unpickle_override can allow old pickles to continue to work.

EXAMPLES:

```
sage: from sage.structure.sage_object import unpickle_override, register_unpickle_override
sage: unpickle_global('sage.rings.integer', 'Integer')
<type 'sage.rings.integer.Integer'>
```

Now we horribly break the pickling system:

sage: register_unpickle_override('sage.rings.integer', 'Integer', Rational, call_name=('sage.rings.rational', 'Rational')) sage: unpickle_global('sage.rings.integer', 'Integer') <type 'sage.rings.rational.Rational'>

And we reach into the internals and put it back:

sage: del unpickle_override[('sage.rings.integer', 'Integer')] sage: unpickle_global('sage.rings.integer', 'Integer') <type 'sage.rings.integer.Integer'>

10.2 Base class for parent objects with generators.

Note: This class is being deprecated, see `sage.structure.parent.Parent` and `sage.structure.category_object.CategoryObject` for the new model.

Many parent objects in Sage are equipped with generators, which are special elements of the object. For example, the polynomial ring $\mathbf{Z}[x, y, z]$ is generated by x, y, and z. In Sage the i^{th} generator of an object `X` is obtained using the notation `X.gen(i)`. From the Sage interactive prompt, the shorthand notation `X.i` is also allowed.

REQUIRED: A class that derives from ParentWithGens *must* define the ngens() and gen(i) methods.

OPTIONAL: It is also good if they define gens() to return all gens, but this is not necessary.

The `gens` function returns a tuple of all generators, the `ngens` function returns the number of generators.

The `_assign_names` functions is for internal use only, and is called when objects are created to set the generator names. It can only be called once.

The following examples illustrate these functions in the context of multivariate polynomial rings and free modules.

EXAMPLES:

```
sage: R = PolynomialRing(ZZ, 3, 'x')
sage: R.ngens()
3
sage: R.gen(0)
x0
sage: R.gens()
(x0, x1, x2)
sage: R.variable_names()
('x0', 'x1', 'x2')
```

This example illustrates generators for a free module over $\mathbf{Z}$.

```
sage: M = FreeModule(ZZ, 4)
sage: M
Ambient free module of rank 4 over the principal ideal domain Integer Ring
sage: M.ngens()
4
sage: M.gen(0)
(1, 0, 0, 0)
sage: M.gens()
((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))
```

class ParentWithAdditiveAbelianGens()

generator_orders()

class ParentWithGens()

gen()

gens()
Return a tuple whose entries are the generators for this object, in order.

hom()
Return the unique homomorphism from self to codomain that sends `self.gens()` to the entries of `im_gens`. Raises a TypeError if there is no such homomorphism.

INPUT:

- `im_gens` - the images in the codomain of the generators of this object under the homomorphism
- `codomain` - the codomain of the homomorphism
- `check` - whether to verify that the images of generators extend to define a map (using only canonical coercisions).

OUTPUT:

- a homomorphism self –> codomain

Note: As a shortcut, one can also give an object X instead of `im_gens`, in which case return the (if it exists) natural map to X.

EXAMPLE: Polynomial Ring We first illustrate construction of a few homomorphisms involving a polynomial ring.

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = R.hom([5], QQ)
sage: f(x^2 - 19)
6

sage: R.<x> = PolynomialRing(QQ)
sage: f = R.hom([5], GF(7))
...
TypeError: images do not define a valid homomorphism

sage: R.<x> = PolynomialRing(GF(7))
sage: f = R.hom([3], GF(49,'a'))
sage: f
Ring morphism:
  From: Univariate Polynomial Ring in x over Finite Field of size 7
  To:   Finite Field in a of size 7^2
  Defn: x |--> 3
sage: f(x+6)
2
```

```
sage: f(x^2+1)
3
```

EXAMPLE: Natural morphism

```
sage: f = ZZ.hom(GF(5))
sage: f(7)
2
sage: f
Ring Coercion morphism:
  From: Integer Ring
  To:   Finite Field of size 5
```

There might not be a natural morphism, in which case a TypeError exception is raised.

```
sage: QQ.hom(ZZ)
...
TypeError: Natural coercion morphism from Rational Field to Integer Ring not defined.
```

ngens()

class **ParentWithMultiplicativeAbelianGens**()

generator_orders()

is_ParentWithAdditiveAbelianGens()

Return True if x is a parent object with additive abelian generators, i.e., derives from `sage.structure.parent.ParentWithAdditiveAbelianGens` and False otherwise.

EXAMPLES:

```
sage: from sage.structure.parent_gens import is_ParentWithAdditiveAbelianGens
sage: is_ParentWithAdditiveAbelianGens(QQ)
False
sage: is_ParentWithAdditiveAbelianGens(QQ^3)
True
```

is_ParentWithGens()

Return True if x is a parent object with generators, i.e., derives from `sage.structure.parent.ParentWithGens` and False otherwise.

EXAMPLES:

```
sage: from sage.structure.parent_gens import is_ParentWithGens
sage: is_ParentWithGens(QQ['x'])
True
sage: is_ParentWithGens(CC)
True
sage: is_ParentWithGens(Primes())
False
```

is_ParentWithMultiplicativeAbelianGens()

Return True if x is a parent object with additive abelian generators, i.e., derives from `sage.structure.parent.ParentWithMultiplicativeAbelianGens` and False otherwise.

EXAMPLES:

```
sage: from sage.structure.parent_gens import is_ParentWithMultiplicativeAbelianGens
sage: is_ParentWithMultiplicativeAbelianGens(QQ)
False
```

```
sage: is_ParentWithMultiplicativeAbelianGens(DirichletGroup(11))
True
```

class localvars()

Context manager for safely temporarily changing the variables names of an object with generators.

Objects with named generators are globally unique in SAGE. Sometimes, though, it is very useful to be able to temporarily display the generators differently. The new Python `with` statement and the localvars context manager make this easy and safe (and fun!)

Suppose X is any object with generators. Write

```
with localvars(X, names[, latex_names] [,normalize=False]):
     some code
     ...
```

and the indented code will be run as if the names in X are changed to the new names. If you give normalize=True, then the names are assumed to be a tuple of the correct number of strings.

If you're writing Python library code, you currently have to put `from __future__ import with_statement` in your file in order to use the `with` statement. This restriction will disappear in Python 2.6.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: with localvars(R, 'z,w'):
...       print x^3 + y^3 - x*y
...
z^3 + w^3 - z*w
```

Note: I wrote this because it was needed to print elements of the quotient of a ring R by an ideal I using the print function for elements of R. See the code in `quotient_ring_element.pyx`.

AUTHOR:

- William Stein (2006-10-31)

normalize_names()

Return a tuple of strings of variable names of length ngens given the input names.

INPUT:

- `ngens` - integer
- `names`
  - tuple or list of strings, such as ('x', 'y')
  - a string prefix, such as 'alpha'
  - string of single character names, such as 'xyz'

EXAMPLES:

```
sage: from sage.structure.parent_gens import normalize_names as nn
sage: nn(1, 'a')
('a',)
sage: nn(2, 'zzz')
('zzz0', 'zzz1')
sage: nn(2, 'ab')
('a', 'b')
sage: nn(3, ('a', 'bb', 'ccc'))
('a', 'bb', 'ccc')
```

```
sage: nn(4, ['a1', 'a2', 'b1', 'b11'])
('a1', 'a2', 'b1', 'b11')
```

TESTS:

```
sage: nn(2, 'z1')
('z10', 'z11')
sage: PolynomialRing(QQ, 2, 'alpha0')
Multivariate Polynomial Ring in alpha00, alpha01 over Rational Field
```

10.3 Formal sums

AUTHORS:

- David Harvey (2006-09-20): changed FormalSum not to derive from "list" anymore, because that breaks new Element interface
- Nick Alexander (2006-12-06): added test cases.
- William Stein (2006, 2009): wrote the first version in 2006, documented it in 2009.

FUNCTIONS:

- **`FormalSums(ring)` – create the module of formal finite sums with** coefficients in the given ring.
- `FormalSum(list of pairs (coeff, number))` – create a formal sum

EXAMPLES:

```
sage: A = FormalSum([(1, 2/3)]); A
2/3
sage: B = FormalSum([(3, 1/5)]); B
3*1/5
sage: -B
-3*1/5
sage: A + B
3*1/5 + 2/3
sage: A - B
-3*1/5 + 2/3
sage: B*3
9*1/5
sage: 2*A
2*2/3
sage: list(2*A + A)
[(3, 2/3)]
```

TESTS:

```
sage: R = FormalSums(QQ)
sage: loads(dumps(R)) == R
True
sage: a = R(2/3) + R(-5/7); a
-5/7 + 2/3
sage: loads(dumps(a)) == a
True
```

class FormalSum(*x, parent=Abelian Group of all Formal Finite Sums over Integer Ring, check=True, reduce=True*)
A formal sum over a ring.

reduce()
EXAMPLES:

```
sage: a = FormalSum([(-2,3), (2,3)], reduce=False); a
-2*3 + 2*3
sage: a.reduce()
sage: a
0
```

FormalSums(*R=Integer Ring*)
Return the R-module of finite formal sums with coefficients in R.

INPUT: R – a ring (default: ZZ)

EXAMPLES:

```
sage: FormalSums()
Abelian Group of all Formal Finite Sums over Integer Ring
sage: FormalSums(ZZ[sqrt(2)])
Abelian Group of all Formal Finite Sums over Order in Number Field in sqrt2 with defining polync
sage: FormalSums(GF(9,'a'))
Abelian Group of all Formal Finite Sums over Finite Field in a of size 3^2
```

class FormalSums_generic(*base=Integer Ring*)

base_extend(*R*)
EXAMPLES:

```
sage: FormalSums(ZZ).base_extend(GF(7))
Abelian Group of all Formal Finite Sums over Finite Field of size 7
```

get_action_impl(*other, op, self_is_left*)
EXAMPLES:

```
sage: A = FormalSums(RR).get_action(RR); A      # indirect doctest
Right scalar multiplication by Real Field with 53 bits of precision on Abelian Group of all

sage: A = FormalSums(ZZ).get_action(QQ); A
Right scalar multiplication by Rational Field on Abelian Group of all Formal Finite Sums ove
with precomposition on left by Call morphism:
  From: Abelian Group of all Formal Finite Sums over Integer Ring
  To:   Abelian Group of all Formal Finite Sums over Rational Field
sage: A = FormalSums(QQ).get_action(ZZ); A
Right scalar multiplication by Integer Ring on Abelian Group of all Formal Finite Sums over
```

10.4 Factorizations

The `Factorization` class provides a structure for holding quite general lists of objects with integer multiplicities. These may hold the results of an arithmetic or algebraic factorization, where the objects may be primes or irreducible polynomials and the multiplicities are the (non-zero) exponents in the factorization. For other types of example, see below.

`Factorization` class objects contain a `list`, so can be printed nicely and be manipulated like a list of prime-exponent pairs, or easily turned into a plain list. For example, we factor the integer -45:

```
sage: F = factor(-45)
```

This returns an object of type `Factorization`:

```
sage: type(F)
<class 'sage.structure.factorization.Factorization'>
```

It prints in a nice factored form:

```
sage: F
-1 * 3^2 * 5
```

There is an underlying list representation, emph{which ignores the unit part} (!).

```
sage: list(F)
[(3, 2), (5, 1)]
```

A `Factorization` is not actually a list:

```
sage: isinstance(F, list)
False
```

However, we can access the `Factorization` F itself as if it were a list:

```
sage: F[0]
(3, 2)
sage: F[1]
(5, 1)
```

To get at the unit part, use the `Factorization.unit()` function:

```
sage: F.unit()
-1
```

All factorizations are immutable. Thus if you write a function that returns a cached version of a factorization, you do not have to return a copy.

```
sage: F = factor(-12); F
-1 * 2^2 * 3
sage: F[0] = (5,4)
...
TypeError: 'Factorization' object does not support item assignment
```

EXAMPLES:

This more complicated example involving polynomials also illustrates +that the unit part is not discarded from factorizations.

```
sage: x = QQ['x'].0
sage: f = -5*(x-2)*(x-3)
sage: f
-5*x^2 + 25*x - 30
sage: F = f.factor(); F
(-5) * (x - 3) * (x - 2)
sage: F.unit()
```

```
-5
sage: expand(F)
-5*x^2 + 25*x - 30
```

The underlying list is the list of pairs (p_i, e_i), where each p_i is a 'prime' and each e_i is an integer. The unit part is discarded by the list.

```
sage: list(F)
[(x - 3, 1), (x - 2, 1)]
sage: len(F)
2
sage: F[1]
(x - 2, 1)
```

In the ring $\mathbf{Z}[x]$, the integer -5 is not a unit, so the factorization has three factors:

```
sage: x = ZZ['x'].0
sage: f = -5*(x-2)*(x-3)
sage: f
-5*x^2 + 25*x - 30
sage: F = f.factor(); F
(-1) * 5 * (x - 3) * (x - 2)
sage: F.universe()
Univariate Polynomial Ring in x over Integer Ring
sage: F.unit()
-1
sage: list(F)
[(5, 1), (x - 3, 1), (x - 2, 1)]
sage: expand(F)
-5*x^2 + 25*x - 30
sage: len(F)
3
```

On the other hand, -1 is a unit in $\mathbf{Z}$, so it is included in the unit.

```
sage: x = ZZ['x'].0
sage: f = -1*(x-2)*(x-3)
sage: F = f.factor(); F
(-1) * (x - 3) * (x - 2)
sage: F.unit()
-1
sage: list(F)
[(x - 3, 1), (x - 2, 1)]
```

Factorizations can involve fairly abstract mathematical objects:

```
sage: F = ModularSymbols(11,4).factorization()
sage: F
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 6 for Gamma_0(11) of w
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 6 for Gamma_0(11) of w
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 6 for Gamma_0(11) of w
sage: type(F)
<class 'sage.structure.factorization.Factorization'>


sage: K.<a> = NumberField(x^2 + 3); K
```

```
Number Field in a with defining polynomial x^2 + 3
sage: f = K.factor(15); f
(Fractional ideal (1/2*a - 3/2))^2 * (Fractional ideal (5))
sage: f.universe()
Monoid of ideals of Number Field in a with defining polynomial x^2 + 3
sage: f.unit()
Fractional ideal (1)
sage: g=K.factor(9); g
(Fractional ideal (1/2*a - 3/2))^4
sage: f.lcm(g)
(Fractional ideal (1/2*a - 3/2))^4 * (Fractional ideal (5))
sage: f.gcd(g)
(Fractional ideal (1/2*a - 3/2))^2
sage: f.is_integral()
True
```

TESTS:

```
sage: F = factor(-20); F
-1 * 2^2 * 5
sage: G = loads(dumps(F)); G
-1 * 2^2 * 5
sage: G == F
True
sage: G is F
False
```

AUTHORS:

- William Stein (2006-01-22): added unit part as suggested by David Kohel.
- William Stein (2008-01-17): wrote much of the documentation and fixed a couple of bugs.
- Nick Alexander (2008-01-19): added support for non-commuting factors.
- John Cremona (2008-08-22): added division, lcm, gcd, is_integral and universe functions

class **Factorization**(*x, unit=None, cr=False, sort=True, simplify=True*)

A formal factorization of an object.

EXAMPLES:

```
sage: N = 2006
sage: F = N.factor(); F
2 * 17 * 59
sage: F.unit()
1
sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.unit()
-1
sage: loads(F.dumps()) == F
True
sage: F = Factorization([(x,1/3)])
...
TypeError: powers of factors must be integers
```

base_change(*U*)

Return the factorization self, with its factors (including the unit part) coerced into the universe U.

EXAMPLES:

```
sage: F = factor(2006)
sage: F.universe()
Integer Ring
sage: P.<x> = ZZ[]
sage: F.base_change(P).universe()
Univariate Polynomial Ring in x over Integer Ring
```

This method will return a TypeError if the coercion is not possible:

```
sage: g = x^2 - 1
sage: F = factor(g); F
(x - 1) * (x + 1)
sage: F.universe()
Univariate Polynomial Ring in x over Integer Ring
sage: F.base_change(ZZ)
...
TypeError: Impossible to coerce the factors of (x - 1) * (x + 1) into Integer Ring
```

expand()

Same as `value()`, so this returns the product of the factors, multiplied out.

EXAMPLES:

```
sage: x = polygen(QQ, 'x')
sage: F = factor(-x^5 + 1); F
(-1) * (x - 1) * (x^4 + x^3 + x^2 + x + 1)
sage: F.expand()
-x^5 + 1
```

gcd(*other*)

Return the gcd of two factorizations.

If the two factorizations have different universes, this method will attempt to find a common universe for the gcd. A TypeError is raised if this is impossible.

EXAMPLES:

```
sage: factor(-30).gcd(factor(-160))
2 * 5
sage: factor(gcd(-30,160))
2 * 5

sage: R.<x> = ZZ[]
sage: (factor(-20).gcd(factor(5*x+10))).universe()
Univariate Polynomial Ring in x over Integer Ring
```

is_commutative()

Return True if my factors commute.

EXAMPLES:

```
sage: F = factor(2006)
sage: F.is_commutative()
True
sage: K = QuadraticField(23, 'a')
sage: F = K.factor(13)
sage: F.is_commutative()
True
sage: R.<x,y,z> = FreeAlgebra(QQ, 3)
```

```
sage: F = Factorization([(z, 2)], 3)
sage: F.is_commutative()
False
sage: (F*F^-1).is_commutative()
False
```

is_integral()

Return True iff all exponents of this Factorization are non-negative

EXAMPLES:

```
sage: F = factor(-10); F
-1 * 2 * 5
sage: F.is_integral()
True

sage: F = factor(-10) / factor(16); F
-1 * 2^-3 * 5
sage: F.is_integral()
False
```

lcm(*other*)

Return the lcm of two factorizations.

If the two factorizations have different universes, this method will attempt to find a common universe for the lcm. A TypeError is raised if this is impossible.

EXAMPLES:

```
sage: factor(-10).lcm(factor(-16))
2^4 * 5
sage: factor(lcm(-10,16))
2^4 * 5

sage: R.<x> = ZZ[]
sage: (factor(-20).lcm(factor(5*x+10))).universe()
Univariate Polynomial Ring in x over Integer Ring
```

prod()

Same as `value()`.

EXAMPLES:

```
sage: F = factor(100)
sage: F.prod()
100
```

simplify()

Combine adjacent products that commute as much as possible.

TESTS:

```
sage: R.<x,y> = FreeAlgebra(ZZ, 2)
sage: F = Factorization([(x,3), (y, 2), (y,2)], simplify=False); F
x^3 * y^2 * y^2
sage: F.simplify(); F
x^3 * y^4
sage: F * Factorization([(y, -2)], 2)
(2) * x^3 * y^2
```

sort(*_cmp=None*)

Sort the factors in this factorization.

INPUT:

- _cmp - (default: None) comparison function

OUTPUT:

- changes this factorization to be sorted

If _cmp is None, we determine the comparison function as follows: If the prime in the first factor has a dimension method, then we sort based first on *dimension* then on the exponent. If there is no dimension method, we next attempt to sort based on a degree method, in which case, we sort based first on *degree*, then exponent to break ties when two factors have the same degree, and if those match break ties based on the actual prime itself. If there is no degree method, we sort based on dimension.

EXAMPLES: We create a factored polynomial:

```
sage: x = polygen(QQ,'x')
sage: F = factor(x^3 + 1); F
(x + 1) * (x^2 - x + 1)
```

Then we sort it but using the negated version of the standard Python cmp function:

```
sage: F.sort(_cmp = lambda x,y: -cmp(x,y))
sage: F
(x^2 - x + 1) * (x + 1)
```

unit()

Return the unit part of this factorization.

EXAMPLES: We create a polynomial over the real double field and factor it:

```
sage: x = polygen(RDF, 'x')
sage: F = factor(-2*x^2 - 1); F
(-2.0) * (1.0*x^2 + 0.5)
```

Note that the unit part of the factorization is -2.0.

```
sage: F.unit()
-2.0

sage: F = factor(-2006); F
-1 * 2 * 17 * 59
sage: F.unit()
-1
```

universe()

Return the parent structure of my factors.

Note: This used to be called `base_ring`, but the universe of a factorization need not be a ring.

EXAMPLES:

```
sage: F = factor(2006)
sage: F.universe()
Integer Ring

sage: R.<x,y,z> = FreeAlgebra(QQ, 3)
sage: F = Factorization([(z, 2)], 3)
sage: (F*F^-1).universe()
Free Algebra on 3 generators (x, y, z) over Rational Field

sage: F = ModularSymbols(11,4).factorization()
sage: F.universe()
```

value()

Return the product of the factors in the factorization, multiplied out.

EXAMPLES:

```
sage: F = factor(2006); F
2 * 17 * 59
sage: F.value()
2006

sage: R.<x,y> = FreeAlgebra(ZZ, 2)
sage: F = Factorization([(x,3), (y, 2), (x,1)]); F
x^3 * y^2 * x
sage: F.value()
x^3*y^2*x
```

10.5 Elements

AUTHORS:

- David Harvey (2006-10-16): changed CommutativeAlgebraElement to derive from CommutativeRingElement instead of AlgebraElement
- David Harvey (2006-10-29): implementation and documentation of new arithmetic architecture
- William Stein (2006-11): arithmetic architecture – pushing it through to completion.
- Gonzalo Tornaria (2007-06): recursive base extend for coercion – lots of tests

10.5.1 The Abstract Element Class Hierarchy

This is the abstract class hierarchy, i.e., these are all abstract base classes.

```
SageObject
    Element
        ModuleElement
            RingElement
                CommutativeRingElement
                    IntegralDomainElement
                        DedekindDomainElement
                            PrincipalIdealDomainElement
                                EuclideanDomainElement
                    FieldElement
                        FiniteFieldElement
                    CommutativeAlgebraElement
                AlgebraElement   (note -- can't derive from module, since no multiple inheritence)
                    CommutativeAlgebra ??? (should be removed from element.pxd)
                    Matrix
                InfinityElement
                    PlusInfinityElement
                    MinusInfinityElement
            AdditiveGroupElement
            Vector

        MonoidElement
            MultiplicativeGroupElement
```

10.5.2 How to Define a New Element Class

Elements typically define a method `_new_c`, e.g.,

```
cdef _new_c(self, defining data):
    cdef FreeModuleElement_generic_dense x
    x = PY_NEW(FreeModuleElement_generic_dense)
    x._parent = self._parent
    x._entries = v
```

that creates a new sibling very quickly from defining data with assumed properties.

Sage has a special system in place for handling arithmetic operations for all Element subclasses. There are various rules that must be followed by both arithmetic implementors and callers.

A quick summary for the impatient:

- To implement addition for any Element class, override def _add_().
- If you want to add x and y, whose parents you know are IDENTICAL, you may call _add_(x, y). This will be the fastest way to guarantee that the correct implementation gets called. Of course you can still always use "x + y".

Now in more detail. The aims of this system are to provide (1) an efficient calling protocol from both python and cython, (2) uniform coercion semantics across Sage, (3) ease of use, (4) readability of code.

We will take addition of RingElements as an example; all other operators and classes are similar. There are four relevant functions.

- **def RingElement.__add__**

  This function is called by python or pyrex when the binary "+" operator is encountered. It ASSUMES that at least one of its arguments is a RingElement; only a really twisted programmer would violate this condition. It has a fast pathway to deal with the most common case where the arguments have the same parent. Otherwise, it uses the coercion module to work out how to make them have the same parent. After any necessary coercions have been performed, it calls _add_ to dispatch to the correct underlying addition implementation.

  Note that although this function is declared as def, it doesn't have the usual overheads associated with python functions (either for the caller or for __add__ itself). This is because python has optimised calling protocols for such special functions.

- **def RingElement._add_**

  This is the function you should override to implement addition in a python subclass of RingElement.

  Warning: if you override this in a *Cython* class, it won't get called. You should override _add_ instead. It is especially important to keep this in mind whenever you move a class down from Python to Cython.

  The two arguments to this function are guaranteed to have the SAME PARENT. Its return value MUST have the SAME PARENT as its arguments.

  If you want to add two objects from python, and you know that their parents are the same object, you are encouraged to call this function directly, instead of using "x + y".

  The default implementation of this function is to call _add_, so if no-one has defined a python implementation, the correct pyrex implementation will get called.

- **cpdef RingElement._add_**

  This is the function you should override to implement addition in a pyrex subclass of RingElement.

The two arguments to this function are guaranteed to have the SAME PARENT. Its return value MUST have the SAME PARENT as its arguments.

The default implementation of this function is to raise a NotImplementedError, which will happen if no-one has supplied implementations of either _add_.

For speed, there are also **inplace** version of the arithmetic commands. DD NOT call them directly, they may mutate the object and will be called when and only when it has been determined that the old object will no longer be accessible from the calling function after this operation.

- **def RingElement._iadd_**

  This is the function you should override to inplace implement addition in a Python subclass of RingElement.

  The two arguments to this function are guaranteed to have the SAME PARENT. Its return value MUST have the SAME PARENT as its arguments.

  The default implementation of this function is to call _add_, so if no-one has defined a Python implementation, the correct Cython implementation will get called.

class AdditiveGroupElement()
Generic element of an additive group.

order()
Return additive order of element

class AlgebraElement()

class CoercionModel()
Most basic coersion scheme. If it doesn't already match, throw an error.

bin_op()

canonical_coercion()

class CommutativeAlgebra()

class CommutativeAlgebraElement()

class CommutativeRingElement()

divides()
Return True if self divides x.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(QQ)
sage: x.divides(x^2)
True
sage: x.divides(x^2+2)
False
sage: (x^2+2).divides(x)
False
sage: P.<x> = PolynomialRing(ZZ)
sage: x.divides(x^2)
True
sage: x.divides(x^2+2)
False
sage: (x^2+2).divides(x)
False
```

inverse_mod()
Return an inverse of self modulo the ideal I, if defined, i.e., if I and self together generate the unit ideal.

mod()

Return a representative for self modulo the ideal I (or the ideal generated by the elements of I if I is not an ideal.)

EXAMPLE: Integers Reduction of 5 modulo an ideal:

```
sage: n = 5
sage: n.mod(3*ZZ)
2
```

Reduction of 5 modulo the ideal generated by 3:

```
sage: n.mod(3)
2
```

Reduction of 5 modulo the ideal generated by 15 and 6, which is (3).

```
sage: n.mod([15,6])
2
```

EXAMPLE: Univiate polynomials

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = x^3 + x + 1
sage: f.mod(x + 1)
-1
```

When little is implemented about a given ring, then mod may return simply return f. For example, reduction is not implemented for $\mathbf{Z}[x]$ yet. (TODO!)

sage: R.<x> = PolynomialRing(ZZ) sage: f = x^3 + x + 1 sage: f.mod(x + 1) x^3 + x + 1

EXAMPLE: Multivariate polynomials We reduce a polynomial in two variables modulo a polynomial and an ideal:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: (x^2 + y^2 + z^2).mod(x+y+z)
2*y^2 + 2*y*z + 2*z^2
```

Notice above that x is eliminated. In the next example, both y and z are eliminated:

```
sage: (x^2 + y^2 + z^2).mod( (x - y, y - z) )
3*z^2
sage: f = (x^2 + y^2 + z^2)^2; f
x^4 + 2*x^2*y^2 + y^4 + 2*x^2*z^2 + 2*y^2*z^2 + z^4
sage: f.mod( (x - y, y - z) )
9*z^4
```

In this example y is eliminated:

```
sage: (x^2 + y^2 + z^2).mod( (x^3, y - z) )
x^2 + 2*z^2
```

class DedekindDomainElement()

class Element()

Generic element of a structure. All other types of elements (RingElement, ModuleElement, etc) derive from this type.

Subtypes must either call __init__() to set _parent, or may set _parent themselves if that would be more efficient.

base_extend()

base_ring()

Returns the base ring of this element's parent (if that makes sense).

category()

is_zero()
Return True if self equals self.parent()(0). The default implementation is to fall back to 'not self.__nonzero__'.

> **Warning:** Do not re-implement this method in your subclass but implement __nonzero__ instead.

n()
Return a numerical approximation of x with at least prec bits of precision.

EXAMPLES:

```
sage: (2/3).n()
0.666666666666667
sage: a = 2/3
sage: pi.n(digits=10)
3.141592654
sage: pi.n(prec=20)    # 20 bits
3.1416
```

parent()
Returns parent of this element; or, if the optional argument x is supplied, the result of coercing x into the parent of this element.

subs()
Substitutes given generators with given values while not touching other generators. This is a generic wrapper around __call__. The syntax is meant to be compatible with the corresponding method for symbolic expressions.

INPUT:

- in_dict - (optional) dictionary of inputs
- **kwds - named parameters

OUTPUT:

- new object if substitution is possible, otherwise self.

EXAMPLES:

```
sage: x, y = PolynomialRing(ZZ,2,'xy').gens()
sage: f = x^2 + y + x^2*y^2 + 5
sage: f((5,y))
25*y^2 + y + 30
sage: f.subs({x:5})
25*y^2 + y + 30
sage: f.subs(x=5)
25*y^2 + y + 30
sage: (1/f).subs(x=5)
1/(25*y^2 + y + 30)
sage: Integer(5).subs(x=4)
5
```

substitute()
This is an alias for self.subs().

INPUT:

- in_dict - (optional) dictionary of inputs
- **kwds - named parameters

OUTPUT:

- new object if substitution is possible, otherwise self.

EXAMPLES:

```
sage: x, y = PolynomialRing(ZZ,2,'xy').gens()
sage: f = x^2 + y + x^2*y^2 + 5
sage: f((5,y))
25*y^2 + y + 30
sage: f.substitute({x:5})
25*y^2 + y + 30
sage: f.substitute(x=5)
25*y^2 + y + 30
sage: (1/f).substitute(x=5)
1/(25*y^2 + y + 30)
sage: Integer(5).substitute(x=4)
5
```

class EuclideanDomainElement()

degree()

leading_coefficient()

quo_rem()

class FieldElement()

divides()

Check whether self divides other, for field elements.

Since this is a field, all values divide all other values, except that zero does not divide any non-zero values.

EXAMPLES:

```
sage: K.<rt3> = QQ[sqrt(3)]
sage: K(0).divides(rt3)
False
sage: rt3.divides(K(17))
True
sage: K(0).divides(K(0))
True
sage: rt3.divides(K(0))
True
```

is_unit()

Return True if self is a unit in its parent ring.

EXAMPLES:

```
sage: a = 2/3; a.is_unit()
True
```

On the other hand, 2 is not a unit, since its parent is ZZ.

```
sage: a = 2; a.is_unit()
False
sage: parent(a)
Integer Ring
```

However, a is a unit when viewed as an element of QQ:

```
sage: a = QQ(2); a.is_unit()
True
```

quo_rem()

class FiniteFieldElement()

additive_order()

Return the additive order of this finite field element.

EXAMPLES:

```
sage: k.<a> = FiniteField(2^12, 'a')
sage: b = a^3 + a + 1
sage: b.additive_order()
2
sage: k(0).additive_order()
1
```

charpoly()

Return the characteristic polynomial of self as a polynomial with given variable.

INPUT:

- var - string (default: 'x')
- algorithm - string (default: 'matrix')
  - 'matrix' - return the charpoly computed from the matrix of left multiplication by self
  - 'pari' – use pari's charpoly routine on polymods, which is not very good except in small cases

The result is not cached.

EXAMPLES:

```
sage: k.<a> = GF(19^2)
sage: parent(a)
Finite Field in a of size 19^2
sage: a.charpoly('X')
X^2 + 18*X + 2
sage: a^2 + 18*a + 2
0
sage: a.charpoly('X', algorithm='pari')
X^2 + 18*X + 2
```

frobenius()

Return the $(p^k)^{th}$ power of self, where p is the characteristic of the field.

INPUT:

- k - integer (default: 1, must fit in C int type)

Note that if k is negative, then this computes the appropriate root.

EXAMPLES:

```
sage: F.<a> = GF(29^2)
sage: z = a^2 + 5*a + 1
sage: z.pth_power()
19*a + 20
sage: z.pth_power(10)
10*a + 28
sage: z.pth_power(-10) == z
True
sage: F.<b> = GF(2^12)
sage: y = b^3 + b + 1
sage: y == (y.pth_power(-3))^(2^3)
True
sage: y.pth_power(2)
b^7 + b^6 + b^5 + b^4 + b^3 + b
```

matrix()

See _matrix_().

EXAMPLE:

```
sage: k.<a> = GF(2^16)
sage: e = a^2 + 1
sage: e.matrix() # random-ish error message
doctest:1: DeprecationWarning:The function matrix is replaced by _matrix_.
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
[1 0 1 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 1 0 1 0 0 0 0 0 0 0 0 0 0 1 1]
[0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 1]
[0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1]
[0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1]
```

minimal_polynomial()

Returns the minimal polynomial of this element (over the corresponding prime subfield).

EXAMPLES:

```
sage: k.<a> = FiniteField(3^4)
sage: parent(a)
Finite Field in a of size 3^4
sage: b=a**20;p=charpoly(b,"y");p
y^4 + 2*y^2 + 1
sage: factor(p)
(y^2 + 1)^2
sage: b.minimal_polynomial('y')
y^2 + 1
```

minpoly()

Returns the minimal polynomial of this element (over the corresponding prime subfield).

EXAMPLES:

```
sage: k.<a> = FiniteField(19^2)
sage: parent(a)
Finite Field in a of size 19^2
sage: b=a**20;p=b.charpoly("x");p
x^2 + 15*x + 4
sage: factor(p)
(x + 17)^2
sage: b.minpoly('x')
x + 17
```

multiplicative_order()

Return the multiplicative order of this field element.

norm()

Return the norm of self down to the prime subfield.

This is the product of the Galois conjugates of self.

EXAMPLES:

```
sage: S.<b> = GF(5^2); S
Finite Field in b of size 5^2
```

```
sage: b.norm()
2
sage: b.charpoly('t')
t^2 + 4*t + 2
```

Next we consider a cubic extension:

```
sage: S.<a> = GF(5^3); S
Finite Field in a of size 5^3
sage: a.norm()
2
sage: a.charpoly('t')
t^3 + 3*t + 3
sage: a * a^5 * (a^25)
2
```

nth_root()

Returns an nth root of self.

INPUT:

- n - integer >= 1 (must fit in C int type)
- extend - bool (default: True); if True, return an nth root in an extension ring, if necessary. Otherwise, raise a ValueError if the root is not in the base ring. Warning: this option is not implemented!
- all - bool (default: False); if True, return all nth roots of self, instead of just one.

OUTPUT:

If self has an nth root, returns one (if all == False) or a list of all of them (if all == True). Otherwise, raises a ValueError (if extend = False) or a NotImplementedError (if extend = True).

Warning: The 'extend' option is not implemented (yet).

AUTHOR:

- David Roe (2007-10-3)

pth_power()

Return the $(p^k)^{th}$ power of self, where p is the characteristic of the field.

INPUT:

- k - integer (default: 1, must fit in C int type)

Note that if k is negative, then this computes the appropriate root.

EXAMPLES:

```
sage: F.<a> = GF(29^2)
sage: z = a^2 + 5*a + 1
sage: z.pth_power()
19*a + 20
sage: z.pth_power(10)
10*a + 28
sage: z.pth_power(-10) == z
True
sage: F.<b> = GF(2^12)
sage: y = b^3 + b + 1
sage: y == (y.pth_power(-3))^(2^3)
True
sage: y.pth_power(2)
b^7 + b^6 + b^5 + b^4 + b^3 + b
```

pth_root()

Return the $(p^k)^{th}$ root of self, where p is the characteristic of the field.

INPUT:

- k - integer (default: 1, must fit in C int type)

Note that if k is negative, then this computes the appropriate power.

EXAMPLES:

```
sage: F.<b> = GF(2^12)
sage: y = b^3 + b + 1
sage: y == (y.pth_root(3))^(2^3)
True
sage: y.pth_root(2)
b^11 + b^10 + b^9 + b^7 + b^5 + b^4 + b^2 + b
```

trace()

Return the trace of this element, which is the sum of the Galois conjugates.

EXAMPLES:

```
sage: S.<a> = GF(5^3); S
Finite Field in a of size 5^3
sage: a.trace()
0
sage: a.charpoly('t')
t^3 + 3*t + 3
sage: a + a^5 + a^25
0
sage: z = a^2 + a + 1
sage: z.trace()
2
sage: z.charpoly('t')
t^3 + 3*t^2 + 2*t + 2
sage: z + z^5 + z^25
2
```

vector()

See _vector_().

EXAMPLE:

```
sage: k.<a> = GF(2^16)
sage: e = a^2 + 1
sage: e.vector() # random-ish error message
doctest:1: DeprecationWarning:The function vector is replaced by _vector_.
(1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
```

class **InfinityElement**()

class **IntegralDomainElement**()

is_nilpotent()

class **Matrix**()

class **MinusInfinityElement**()

class **ModuleElement**()

Generic element of a module.

additive_order()

Return the additive order of self.

order()

Return the additive order of self.

class **MonoidElement**()

Generic element of a monoid.

multiplicative_order()
Return the multiplicative order of self.

order()
Return the multiplicative order of self.

class MultiplicativeGroupElement()
Generic element of a multiplicative group.

order()
Return the multiplicative order of self.

class PlusInfinityElement()

class PrincipalIdealDomainElement()

gcd()
Returns the gcd of self and right, or 0 if both are 0.

lcm()
Returns the least common multiple of self and right.

xgcd()
Return the extended gcd of self and other, i.e., elements r, s, t such that .. math:

```
r = s \cdot self + t \cdot other.
```

Note: There is no guarantee on minimality of the cofactors. In the integer case, see documentation for Integer._xgcd() to obtain minimal cofactors.

class RingElement()

abs()
Return the absolute value of self. (This just calls the __abs__ method, so it is equivalent to the abs() built-in function.)
EXAMPLES:

```
sage: RR(-1).abs()
1.00000000000000
sage: ZZ(-1).abs()
1
sage: CC(I).abs()
1.00000000000000
sage: Mod(-15, 37).abs()
...
ArithmeticError: absolute valued not defined on integers modulo n.
```

additive_order()
Return the additive order of self.

is_nilpotent()
Return True if self is nilpotent, i.e., some power of self is 0.

is_one()

is_unit()

multiplicative_order()
Return the multiplicative order of self, if self is a unit, or raise `ArithmeticError` otherwise.

order()
Return the additive order of self.
This is deprecated; use `additive_order` instead.
EXAMPLES:

```
sage: a = Integers(12)(5)
sage: a.order()
doctest... DeprecationWarning: The function order is deprecated for ring elements; use addit
12
```

class Vector()

bin_op()

canonical_coercion()

canonical_coercion(x,y) is what is called before doing an arithmetic operation between x and y. It returns a pair (z,w) such that z is got from x and w from y via canonical coercion and the parents of z and w are identical.

EXAMPLES:

```
sage: A = Matrix([[0,1],[1,0]])
sage: canonical_coercion(A,1)
([0 1]
[1 0], [1 0]
[0 1])
```

coerce_cmp()

coercion_traceback()

This function is very helpful in debugging coercion errors. It prints the tracebacks of all the errors caught in the coercion detection. Note that failure is cached, so some errors may be omitted the second time around (as it remembers not to retry failed paths for speed reasons.

EXAMPLES:

```
sage: 1 + 1/5
6/5
sage: coercion_traceback()  # Should be empty, as all went well.
sage: 1/5 + GF(5).gen()
...
TypeError: unsupported operand parent(s) for '+': 'Rational Field' and 'Finite Field of size 5'
sage: coercion_traceback()
...
TypeError: no common canonical parent for objects with parents: 'Rational Field' and 'Finite Fie
```

gcd()

generic_power()

Computes a^n, where n is an integer, and a is an object which supports multiplication. Optionally an additional argument, which is used in the case that n == 0:

- one - the "unit" element, returned directly (can be anything)

If this is not supplied, int(1) is returned.

EXAMPLES:

```
sage: from sage.structure.element import generic_power
sage: generic_power(int(12),int(0))
1
sage: generic_power(int(0),int(100))
0
sage: generic_power(Integer(10),Integer(0))
1
sage: generic_power(Integer(0),Integer(23))
0
```

```
sage: sum([generic_power(2,i) for i in range(17)]) #test all 4-bit combinations
131071
sage: F = Zmod(5)
sage: a = generic_power(F(2), 5); a
2
sage: a.parent() is F
True
sage: a = generic_power(F(1), 2)
sage: a.parent() is F
True

sage: generic_power(int(5), 0)
1
```

get_coercion_model()
Return the global coercion model.

EXAMPLES:

```
sage: import sage.structure.element as e
sage: cm = e.get_coercion_model()
sage: cm
<sage.structure.coerce.CoercionModel_cache_maps object at ...>
```

is_AdditiveGroupElement()
Return True if x is of type AdditiveGroupElement.

is_AlgebraElement()
Return True if x is of type AlgebraElement.

is_CommutativeAlgebraElement()
Return True if x is of type CommutativeAlgebraElement.

is_CommutativeRingElement()
Return True if x is of type CommutativeRingElement.

is_DedekindDomainElement()
Return True if x is of type DedekindDomainElement.

is_Element()
Return True if x is of type Element.

EXAMPLES:

```
sage: from sage.structure.element import is_Element
sage: is_Element(2/3)
True
sage: is_Element(QQ^3)
False
```

is_EuclideanDomainElement()
Return True if x is of type EuclideanDomainElement.

is_FieldElement()
Return True if x is of type FieldElement.

is_InfinityElement()
Return True if x is of type InfinityElement.

is_IntegralDomainElement()
Return True if x is of type IntegralDomainElement.

is_Matrix()

is_ModuleElement()
Return True if x is of type ModuleElement.

This is even faster than using isinstance inline.

EXAMPLES:

```
sage: from sage.structure.element import is_ModuleElement
sage: is_ModuleElement(2/3)
True
sage: is_ModuleElement((QQ^3).0)
True
sage: is_ModuleElement('a')
False
```

is_MonoidElement()
Return True if x is of type MonoidElement.

is_MultiplicativeGroupElement()
Return True if x is of type MultiplicativeGroupElement.

is_PrincipalIdealDomainElement()
Return True if x is of type PrincipalIdealDomainElement.

is_RingElement()
Return True if x is of type RingElement.

is_Vector()

lcm()

make_element()
This function is only here to support old pickles.

Pickling functionality is moved to Element.{__getstate__,__setstate__} functions.

parent()

py_scalar_to_element()

set_coercion_model()

xgcd()

10.6 UniqueRepresentation

class **UniqueRepresentation**()
Classes derived from UniqueRepresentation inherit a unique representation behavior for their instances.

EXAMPLES:

The short story: to construct a class whose instances have a unique representation behavior one just have to do:

```
sage: class MyClass(UniqueRepresentation):
...       # all the rest as usual
...       pass
```

Everything below is for the curious or for advanced usage.

What is unique representation?

Instances of a class have a *unique representation behavior* when several instances constructed with the same arguments share the same memory representation. For example, calling twice:

```
sage: f = GF(7)
sage: g = GF(7)
```

to create the finite field of order 7 actually gives back the same object:

```
sage: f == g
True
sage: f is g
True
```

This is a standard design pattern. Besides saving memory, it allows for sharing cached data (say representation theoretical information about a group) as well as for further optimizations (fast hashing, equality testing). This behaviour is typically desirable for parents and categories. It can also be useful for intensive computations where one wants to cache all the operations on a small set of elements (say the multiplication table of a small group), and access this cache as quickly as possible.

The `UniqueRepresentation` and `UniqueFactory` classes provide two alternative implementations of this design pattern. Both implementations have their own merits. `UniqueRepresentation` is very easy to use: a class just needs to derive from it, or make sure some of its super classes does. For basic usage . Also, it groups together the class and the factory in a single gadget; in the example above, one would want to do:

```
sage: isinstance(f, GF)          # todo: not implemented
True
```

but this does not work, because GF is only the factory.

On the other hand the `UniqueRepresentation` class is more intrusive, as it imposes a behavior (and a metaclass) to all the subclasses. Its implementation is also more technical, which leads to some subtleties.

EXAMPLES:

We start with a simple class whose constructor takes a single value as argument (TODO: find a more meaningful example):

```
sage: class MyClass(UniqueRepresentation):
...       def __init__(self, value):
...           self.value = value
...
```

Two coexisting instances of MyClass created with the same argument data are guaranteed to share the same identity:

```
sage: x = MyClass(1)
sage: y = MyClass(1)
sage: x is y
True
sage: z = MyClass(2)
sage: x is z
False
```

In particular, modifying any one of them modifies the other (reference effect):

```
sage: x.value = 3
sage: x.value, y.value
(3, 3)
sage: y.value = 1
sage: x.value, y.value
(1, 1)
```

Unless overriden by the derived class, equality testing is implemented by comparing identities, which is as fast as it can get:

```
sage: x == y
True
sage: z = MyClass(2)
sage: x == z, x is z
(False, False)
```

Similarly, the identity is used as hash function, which is also as fast as it can get. However this means that the hash function may change in between Sage sessions, or even within the same Sage session (see below). Subclasses should overload `__hash__()` if this could be a problem.

The arguments can consist of any combination of positional or keyword arguements, as taken by a usual `__init__()` function. However, all values passed in should be hashable:

```
sage: MyClass(value = [1,2,3])
...
TypeError: list objects are unhashable
```

Argument preprocessing

Sometimes, one wants to do some preprocessing on the arguments, to put them in some canonical form. The following example illustrates how to achieve this; it takes as argument any iterable, and canonicalizes it into a tuple (which is hashable!):

```
sage: class MyClass2(UniqueRepresentation):
...       @staticmethod
...       def __classcall__(cls, iterable):
...           t = tuple(iterable)
...           return super(MyClass2, cls).__classcall__(cls, t)
...
...       def __init__(self, value):
...           self.value = value
...
sage: x = MyClass2([1,2,3])
sage: y = MyClass2(tuple([1,2,3]))
sage: z = MyClass2(i for i in [1,2,3])
sage: x.value
(1, 2, 3)
sage: x is y, y is z
(True, True)
```

A similar situation arises when the constructor accepts default values for some of its parameters. Alas, the obvious implementation does not work:

```
sage: class MyClass3(UniqueRepresentation):
...       def __init__(self, value = 3):
...           self.value = value
```

```
...
sage: MyClass3(3) is MyClass3()
False
```

Instead, one should do:

```
sage: class MyClass3(UniqueRepresentation):
...       @staticmethod
...       def __classcall__(cls, value = 3):
...           return super(MyClass3, cls).__classcall__(cls, value)
...
...       def __init__(self, value):
...           self.value = value
...
sage: MyClass3(3) is MyClass3()
True
```

A bit of explanation is in order. First, the call `MyClass2([1,2,3])` triggers a call to `MyClass2.__classcall__(MyClass2, [1,2,3])`. This is an extension of the standard Python behavior, needed by `UniqueRepresentation`, and implemented by the `ClasscallMetaclass`. Then, `MyClass2.__classcall__` does the desired transformations on the arguments. Finally, it uses `super` to call the default implementation of `__classcall__` provided by `UniqueRepresentation`. This one in turn handles the caching and, if needed, constructs and initializes a new object in the class using `__new__()` and `__init__()` as usual.

Constraints:

- `__classcall__()` is a staticmethod (like, implicitly, `__new__()`)
- the preprocessing on the arguments should be idempotent. Namely, If `MyClass2.__classcall__()` calls `UniqueRepresentation.__classcall__(<some_arguments>)`, then it should accept <some_arguments> as its own input, and pass it down unmodified to `UniqueRepresentation.__classcall__()`.
- `MyClass2.__classcall__()` should return the result of `UniqueRepresentation.__classcall__()` without modifying it.

Other than that `MyClass2.__classcall__()` may play any tricks, like acting as a Factory and returning object from other classes.

Unique representation and mutability

`UniqueRepresentation` is primarily intended for implementing objects which are (at least semantically) immutable. This is in particular assumed by the default implementations of `copy` and `deepcopy`:

```
sage: copy(x) is x
True
sage: from copy import deepcopy
sage: deepcopy(x) is x
True
```

Using `UniqueRepresentation` on mutable objects may lead to subtle behavior:

```
sage: t = MyClass(3)
sage: z = MyClass(2)
sage: t.value = 2
```

Now x and z have the same data structure, but are not considered as equal:

```
sage: t.value == z.value
True
sage: t == z
False
```

More on unique representation and identity

UniqueRepresentation is implemented by mean of a cache. This cache uses weak references so that, when all other references to, say, MyClass(1) have been deleted, the instance is actually deleted from memory. A later call to MyClass(1) reconstructs the instance from scratch, *most likely with a different id*.

TODO: add an example illustrating this behavior

Unique representation and pickling

The default Python pickling implementation (by reconstructing an object from its class and dictionary, see "The pickle protocol" in the Python Library Reference) does not preserves unique representation, as Python has no chance to know whether and where the same object already exists.

UniqueRepresentation tries to ensure appropriate pickling by implementing a __reduce__() method returning the arguments passed to the constructor:

```
sage: import __main__              # Fake MyClass being defined in a python module
sage: __main__.MyClass = MyClass
sage: x = MyClass(1)
sage: loads(dumps(x)) is x
True
```

UniqueRepresentation uses the __reduce__() pickle protocol rather than __getnewargs__() because the later does not handle keyword arguments:

```
sage: x = MyClass(value = 1)
sage: x.__reduce__()
(<function unreduce at ...>, (<class '__main__.MyClass'>, (), {'value': 1}))
sage: x is loads(dumps(x))
True
```

Caveat: the default implementation of __reduce__() in UniqueRepresentation requires to store the constructor's arguments in the instance dictionary upon construction:

sage: x.__dict__ {'_reduction': (<class '__main__.MyClass'>, (), {'value': 1}), 'value': 1}

It is often easy in a derived subclass to reconstruct the constructors arguments from the instance data structure. When this is the case, __reduce__() should be overriden; automagically the arguments won't be stored anymore:

sage: class MyClass3(UniqueRepresentation): ... def __init__(self, value): ... self.value = value ... ... def __reduce__(self): ... return (MyClass3, (self.value,)) ... sage: import __main__; __main__.MyClass3 = MyClass3 # Fake MyClass3 being defined in a python module sage: x = MyClass3(1) sage: loads(dumps(x)) is x True sage: x.__dict__ {'value': 1}

Migrating classes to `UniqueRepresentation` and unpickling

We check that, when migrating a class to `UniqueRepresentation`, older pickle can still be reasonably unpickled. Let us create a (new style) class, and pickle one of its instances:

```
sage: class MyClass4(object):
...       def __init__(self, value):
...           self.value = value
...
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined in a python m
sage: pickle = dumps(MyClass4(1))
```

It can be unpickled:

```
sage: y = loads(pickle)
sage: y.value
1
```

Now, we upgrade the class to derive from `UniqueRepresentation`:

```
sage: class MyClass4(UniqueRepresentation, object):
...       def __init__(self, value):
...           self.value = value
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined in a python m
sage: __main__.MyClass4 = MyClass4
```

The pickle can still be unpickled:

```
sage: y = loads(pickle)
sage: y.value
1
```

Note however that, for the reasons explained above, unique representation is not guaranteed in this case:

```
sage: y is MyClass4(1)
False
```

Todo: illustrate how this can be fixed on a case by case basis.

Now, we redo the same test for a class deriving from SageObject:

```
sage: class MyClass4(SageObject):
...       def __init__(self, value):
...           self.value = value
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined in a python m
sage: pickle = dumps(MyClass4(1))

sage: class MyClass4(UniqueRepresentation, SageObject):
...       def __init__(self, value):
...           self.value = value
sage: __main__.MyClass4 = MyClass4
sage: y = loads(pickle)
sage: y.value
1
```

Caveat: unpickling instances of a formerly old-style class is not supported yet by default:

```
sage: class MyClass4:
...       def __init__(self, value):
...           self.value = value
sage: import __main__; __main__.MyClass4 = MyClass4  # Fake MyClass4 being defined in a python m
sage: pickle = dumps(MyClass4(1))

sage: class MyClass4(UniqueRepresentation, SageObject):
...       def __init__(self, value):
...           self.value = value
sage: __main__.MyClass4 = MyClass4
sage: y = loads(pickle)  # todo: not implemented
sage: y.value            # todo: not implemented
1
```

Rationale for the current implementation

`UniqueRepresentation` and derived classes use the `ClasscallMetaclass` of the standard Python type. The following example explains why.

We define a variant of `MyClass` where the calls to `__init__()` are traced:

```
sage: class MyClass(UniqueRepresentation):
...       def __init__(self, value):
...           print "initializing object"
...           self.value = value
...
```

Let us create an object twice:

```
sage: x = MyClass(1)
initializing object
sage: z = MyClass(1)
```

As desired the __init__ method was only called the first time, which is an important feature.

As far as we can tell, this is not achievable while just using `__new__()` and `__init__()` (as defined by type; see Section "Basic Customization" in the Python Reference Manual). Indeed, `__init__()` is called systematically on the result of `__new__()` whenever the result is an instance of the class.

Another difficulty is that argument preprocessing (as in the example above) cannot be handled by `__new__()`, since the unprocessed arguments will be passed down to `__init__()`.

TESTS:

For the record, this test did fail with previous implementation attempts:

```
sage: class bla(UniqueRepresentation, SageObject):
...       pass
...
sage: b = bla()
```

unreduce(*cls, args, keywords*)
Calls a class on the given arguments:

```
sage: sage.structure.unique_representation.unreduce(Integer, (1,), {})
1
```

Todo: should reuse something preexisting ...

10.7 Mutability Cython Implementation

class Mutability()

is_immutable()
Return True if this object is immutable (can not be changed) and False if it is not.
To make this object immutable use self.set_immutable().
EXAMPLE:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.is_immutable()
False
sage: v.set_immutable()
sage: v.is_immutable()
True
```

is_mutable()

set_immutable()
Make this object immutable, so it can never again be changed.
EXAMPLES:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.set_immutable()
sage: v[3] = 7
...
ValueError: object is immutable; please change a copy instead.
```

10.8 Sequences

A mutable sequence of elements with a common guaranteed category, which can be set immutable.

Sequence derives from list, so has all the functionality of lists and can be used wherever lists are used. When a sequence is created without explicitly given the common universe of the elements, the constructor coerces the first and second element to some *canonical* common parent, if possible, then the second and third, etc. If this is possible, it then coerces everything into the canonical parent at the end. (Note that canonical coercion is very restrictive.) The sequence then has a function `universe()` which returns either the common canonical parent (if the coercion succeeded), or the category of all objects (Objects()). So if you have a list v and type

sage: v = [1, 2/3, 5] sage: w = Sequence(v) sage: w.universe() Rational Field

then since `w.universe()` is $\mathbf{Q}$, you're guaranteed that all elements of w are rationals:

sage: v[0].parent() Integer Ring sage: w[0].parent() Rational Field

If you do assignment to w this property of being rationals is guaranteed to be preserved.

sage: w[0] = 2 sage: w[0].parent() Rational Field sage: w[0] = 'hi' Traceback (most recent call last): ... TypeError: unable to convert hi to a rational

However, if you do `w = Sequence(v)` and the resulting universe is `Objects()`, the elements are not guaranteed to have any special parent. This is what should happen, e.g., with finite field elements of different characteristics:

```
sage: v = Sequence([GF(3)(1), GF(7)(1)])
sage: v.universe()
Category of objects
```

You can make a list immutable with `v.freeze()`. Assignment is never again allowed on an immutable list.

Creation of a sequence involves making a copy of the input list, and substantial coercions. It can be greatly sped up by explicitly specifying the universe of the sequence:

```
sage: v = Sequence(range(10000), universe=ZZ)
```

TESTS:

```
sage: v = Sequence([1..5])
sage: loads(dumps(v)) == v
True
```

class Sequence (*x, universe=None, check=True, immutable=False, cr=False, cr_str=None, use_sage_types=False*)

A mutable list of elements with a common guaranteed universe, which can be set immutable.

A universe is either an object that supports coercion (e.g., a parent), or a category.

INPUT:

- `x` - a list or tuple instance
- `universe` - (default: None) the universe of elements; if None determined using canonical coercions and the entire list of elements. If list is empty, is category Objects() of all objects.
- `check` – (default: True) whether to coerce the elements of x into the universe
- `immutable` - (default: True) whether or not this sequence is immutable
- `cr` - (default: False) if True, then print a carriage return after each comma when printing this sequence.
- **`use_sage_types` – (default: False) if True, coerce the** builtin Python numerical types int, long, float, complex to the corresponding Sage types (this makes functions like vector() more flexible)

OUTPUT:

- a sequence

EXAMPLES:

```
sage: v = Sequence(range(10))
sage: v.universe()
<type 'int'>
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can request that the builtin Python numerical types be coerced to Sage objects:

```
sage: v = Sequence(range(10), use_sage_types=True)
sage: v.universe()
Integer Ring
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can also use seq for "Sequence", which is identical to using Sequence:

```
sage: v = seq([1,2,1/1]); v
[1, 2, 1]
sage: v.universe()
Rational Field
sage: v.parent()
Category of sequences in Rational Field
sage: v.parent()([3,4/3])
[3, 4/3]
```

Note that assignment coerces if possible,

```
sage: v = Sequence(range(10), ZZ)
sage: a = QQ(5)
sage: v[3] = a
sage: parent(v[3])
Integer Ring
sage: parent(a)
Rational Field
sage: v[3] = 2/3
...
TypeError: no conversion of this rational to integer
```

Sequences can be used absolutely anywhere lists or tuples can be used:

```
sage: isinstance(v, list)
True
```

Sequence can be immutable, so entries can't be changed:

```
sage: v = Sequence([1,2,3], immutable=True)
sage: v.is_immutable()
True
sage: v[0] = 5
...
ValueError: object is immutable; please change a copy instead.
```

Only immutable sequences are hashable (unlike Python lists), though the hashing is potentially slow, since it first involves conversion of the sequence to a tuple, and returning the hash of that.

```
sage: v = Sequence(range(10), ZZ, immutable=True)
sage: hash(v)
1591723448              # 32-bit
-4181190870548101704    # 64-bit
```

If you really know what you are doing, you can circumvent the type checking (for an efficiency gain):

```
sage: list.__setitem__(v, int(1), 2/3)         # bad circumvention
sage: v
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list.__setitem__(v, int(1), int(2))      # not so bad circumvention
```

You can make a sequence with a new universe from an old sequence.

```
sage: w = Sequence(v, QQ)
sage: w
[0, 2, 2, 3, 4, 5, 6, 7, 8, 9]
sage: w.universe()
Rational Field
```

```
sage: w[1] = 2/3
sage: w
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
```

Sequences themselves live in a category, the category of all sequences in the given universe.

```
sage: w.category()
Category of sequences in Rational Field
```

This is also the parent of any sequence:

```
sage: w.parent()
Category of sequences in Rational Field
```

The default universe for any sequence, if no compatible parent structure can be found, is the universe of all Sage objects.

This example illustrates how every element of a list is taken into account when constructing a sequence.

```
sage: v = Sequence([1,7,6,GF(5)(3)]); v
[1, 2, 1, 3]
sage: v.universe()
Finite Field of size 5
sage: v.parent()
Category of sequences in Finite Field of size 5
sage: v.parent()([7,8,9])
[2, 3, 4]
```

append(*x*)

EXAMPLES: sage: v = Sequence([1,2,3,4], immutable=True) sage: v.append(34) Traceback (most recent call last): ... ValueError: object is immutable; please change a copy instead. sage: v = Sequence([1/3,2,3,4]) sage: v.append(4) sage: type(v[4]) <type 'sage.rings.rational.Rational'>

category()

EXAMPLES:

```
sage: Sequence([1,2/3,-2/5]).category()
Category of sequences in Rational Field
```

extend(*iterable*)

Extend list by appending elements from the iterable.

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.extend(range(4))
sage: B
[1, 2, 3, 0, 1, 2, 3]
```

insert(*index, object*)

Insert object before index.

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.insert(10, 5)
sage: B
[1, 2, 3, 5]
```

is_immutable()
Return True if this object is immutable (can not be changed) and False if it is not.
To make this object immutable use `set_immutable()`.
EXAMPLE:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.is_immutable()
False
sage: v.set_immutable()
sage: v.is_immutable()
True
```

is_mutable()
EXAMPLES:

```
sage: a = Sequence([1,2/3,-2/5])
sage: a.is_mutable()
True
sage: a[0] = 100
sage: type(a[0])
<type 'sage.rings.rational.Rational'>
sage: a.set_immutable()
sage: a[0] = 50
...
ValueError: object is immutable; please change a copy instead.
sage: a.is_mutable()
False
```

parent()
EXAMPLES:

```
sage: Sequence([1,2/3,-2/5]).parent()
Category of sequences in Rational Field
```

pop(*index=-1*)
Remove and return item at index (default last)
EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.pop(1)
2
sage: B
[1, 3]
```

remove(*value*)
Remove first occurrence of value
EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.remove(2)
sage: B
[1, 3]
```

reverse()
Reverse the elements of self, in place.
EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.reverse(); B
[3, 2, 1]
```

set_immutable()
Make this object immutable, so it can never again be changed.
EXAMPLES:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.set_immutable()
sage: v[3] = 7
...
ValueError: object is immutable; please change a copy instead.
```

sort(*cmp=None, key=None, reverse=False*)
Sort this list *IN PLACE*.
cmp(x, y) -> -1, 0, 1
EXAMPLES:

```
sage: B = Sequence([3,2,1/5])
sage: B.sort()
sage: B
[1/5, 2, 3]
sage: B.sort(reverse=True); B
[3, 2, 1/5]
sage: B.sort(cmp = lambda x,y: cmp(y,x)); B
[3, 2, 1/5]
sage: B.sort(cmp = lambda x,y: cmp(y,x), reverse=True); B
[1/5, 2, 3]
```

universe()
EXAMPLES:

```
sage: Sequence([1,2/3,-2/5]).universe()
Rational Field
sage: Sequence([1,2/3,'-2/5']).universe()
Category of objects
```

class **seq**(*x, universe=None, check=True, immutable=False, cr=False, cr_str=None, use_sage_types=False*)
A mutable list of elements with a common guaranteed universe, which can be set immutable.

A universe is either an object that supports coercion (e.g., a parent), or a category.

INPUT:

- `x` - a list or tuple instance
- `universe` - (default: None) the universe of elements; if None determined using canonical coercions and the entire list of elements. If list is empty, is category Objects() of all objects.
- `check` – (default: True) whether to coerce the elements of x into the universe
- `immutable` - (default: True) whether or not this sequence is immutable
- `cr` - (default: False) if True, then print a carriage return after each comma when printing this sequence.
- **`use_sage_types` – (default: False) if True, coerce the** builtin Python numerical types int, long, float, complex to the corresponding Sage types (this makes functions like vector() more flexible)

OUTPUT:

- a sequence

EXAMPLES:

```
sage: v = Sequence(range(10))
sage: v.universe()
<type 'int'>
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can request that the builtin Python numerical types be coerced to Sage objects:

```
sage: v = Sequence(range(10), use_sage_types=True)
sage: v.universe()
Integer Ring
sage: v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

You can also use seq for "Sequence", which is identical to using Sequence:

```
sage: v = seq([1,2,1/1]); v
[1, 2, 1]
sage: v.universe()
Rational Field
sage: v.parent()
Category of sequences in Rational Field
sage: v.parent()([3,4/3])
[3, 4/3]
```

Note that assignment coerces if possible,

```
sage: v = Sequence(range(10), ZZ)
sage: a = QQ(5)
sage: v[3] = a
sage: parent(v[3])
Integer Ring
sage: parent(a)
Rational Field
sage: v[3] = 2/3
...
TypeError: no conversion of this rational to integer
```

Sequences can be used absolutely anywhere lists or tuples can be used:

```
sage: isinstance(v, list)
True
```

Sequence can be immutable, so entries can't be changed:

```
sage: v = Sequence([1,2,3], immutable=True)
sage: v.is_immutable()
True
sage: v[0] = 5
...
ValueError: object is immutable; please change a copy instead.
```

Only immutable sequences are hashable (unlike Python lists), though the hashing is potentially slow, since it first involves conversion of the sequence to a tuple, and returning the hash of that.

```
sage: v = Sequence(range(10), ZZ, immutable=True)
sage: hash(v)
1591723448              # 32-bit
-4181190870548101704    # 64-bit
```

If you really know what you are doing, you can circumvent the type checking (for an efficiency gain):

```
sage: list.__setitem__(v, int(1), 2/3)         # bad circumvention
sage: v
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
sage: list.__setitem__(v, int(1), int(2))      # not so bad circumvention
```

You can make a sequence with a new universe from an old sequence.

```
sage: w = Sequence(v, QQ)
sage: w
[0, 2, 2, 3, 4, 5, 6, 7, 8, 9]
sage: w.universe()
Rational Field
sage: w[1] = 2/3
sage: w
[0, 2/3, 2, 3, 4, 5, 6, 7, 8, 9]
```

Sequences themselves live in a category, the category of all sequences in the given universe.

```
sage: w.category()
Category of sequences in Rational Field
```

This is also the parent of any sequence:

```
sage: w.parent()
Category of sequences in Rational Field
```

The default universe for any sequence, if no compatible parent structure can be found, is the universe of all Sage objects.

This example illustrates how every element of a list is taken into account when constructing a sequence.

```
sage: v = Sequence([1,7,6,GF(5)(3)]); v
[1, 2, 1, 3]
sage: v.universe()
Finite Field of size 5
sage: v.parent()
Category of sequences in Finite Field of size 5
sage: v.parent()([7,8,9])
[2, 3, 4]
```

append(*x*)

EXAMPLES: sage: v = Sequence([1,2,3,4], immutable=True) sage: v.append(34) Traceback (most recent call last): ... ValueError: object is immutable; please change a copy instead. sage: v = Sequence([1/3,2,3,4]) sage: v.append(4) sage: type(v[4]) <type 'sage.rings.rational.Rational'>

category()

EXAMPLES:

```
sage: Sequence([1,2/3,-2/5]).category()
Category of sequences in Rational Field
```

extend(*iterable*)
Extend list by appending elements from the iterable.
EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.extend(range(4))
sage: B
[1, 2, 3, 0, 1, 2, 3]
```

insert(*index, object*)
Insert object before index.
EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.insert(10, 5)
sage: B
[1, 2, 3, 5]
```

is_immutable()
Return True if this object is immutable (can not be changed) and False if it is not.
To make this object immutable use `set_immutable()`.
EXAMPLE:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.is_immutable()
False
sage: v.set_immutable()
sage: v.is_immutable()
True
```

is_mutable()
EXAMPLES:

```
sage: a = Sequence([1,2/3,-2/5])
sage: a.is_mutable()
True
sage: a[0] = 100
sage: type(a[0])
<type 'sage.rings.rational.Rational'>
sage: a.set_immutable()
sage: a[0] = 50
...
ValueError: object is immutable; please change a copy instead.
sage: a.is_mutable()
False
```

parent()
EXAMPLES:

```
sage: Sequence([1,2/3,-2/5]).parent()
Category of sequences in Rational Field
```

pop(*index=-1*)
Remove and return item at index (default last)
EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.pop(1)
2
sage: B
[1, 3]
```

remove(*value*)

Remove first occurrence of value

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.remove(2)
sage: B
[1, 3]
```

reverse()

Reverse the elements of self, in place.

EXAMPLES:

```
sage: B = Sequence([1,2,3])
sage: B.reverse(); B
[3, 2, 1]
```

set_immutable()

Make this object immutable, so it can never again be changed.

EXAMPLES:

```
sage: v = Sequence([1,2,3,4/5])
sage: v[0] = 5
sage: v
[5, 2, 3, 4/5]
sage: v.set_immutable()
sage: v[3] = 7
...
ValueError: object is immutable; please change a copy instead.
```

sort(*cmp=None, key=None, reverse=False*)

Sort this list *IN PLACE*.

cmp(x, y) -> -1, 0, 1

EXAMPLES:

```
sage: B = Sequence([3,2,1/5])
sage: B.sort()
sage: B
[1/5, 2, 3]
sage: B.sort(reverse=True); B
[3, 2, 1/5]
sage: B.sort(cmp = lambda x,y: cmp(y,x)); B
[3, 2, 1/5]
sage: B.sort(cmp = lambda x,y: cmp(y,x), reverse=True); B
[1/5, 2, 3]
```

universe()

EXAMPLES:

```
sage: Sequence([1,2/3,-2/5]).universe()
Rational Field
sage: Sequence([1,2/3,'-2/5']).universe()
Category of objects
```

10.9 A class for wrapping Sage or Python objects as Sage elements

class ElementWrapper(*value, parent*)

A class for wrapping Sage or Python objects as Sage elements

EXAMPLES:

```
sage: o = ElementWrapper("bla", parent=ZZ); o
'bla'
sage: isinstance(o, sage.structure.element.Element)
True
sage: o.parent()
Integer Ring
sage: o.value
'bla'
```

This is of course a meaningless example: *bla* is not an element of `ZZ`.

Note that `o` is not *an instance of* `str`, but rather *contains a* `str`. Therefore, `o` does not inherit the string methods. On the other hand, it is provided with reasonable default implementations for equality testing, hashing, etc.

The typical use case of *ElementWrapper* is for trivially constructing new element classes from preexisting Sage or Python classes, with a containment relation. Here we construct the tropical monoid of integers endowed with `min` as multiplication. There, it is desirable *not* to inherit the `factor` method from Integer:

```
sage: class MinMonoid(Parent):
...       def _repr_(self):
...           return "The min monoid"
...
sage: M = MinMonoid()
sage: class MinMonoidElement(ElementWrapper):
...       wrapped_class = Integer
...
...       def __mul__(self, other):
...           return MinMonoidElement(min(self.value, other.value), parent = self.parent())
sage: x = MinMonoidElement(5, parent = M); x
5
sage: x.parent()
The min monoid
sage: x.value
5
sage: y = MinMonoidElement(3, parent = M)
sage: x * y
3
```

This example was voluntarily kept to a bare minimum. See the (upcoming) examples in the categories for several full featured applications.

Caveat: the order between the value and the parent argument is likely to change shortly. At this point, all the code using it in the Sage library will be updated. There will be no transition period.

wrapped_class()

The most base type

10.10 Sets

AUTHORS:

- William Stein (2005) - first version
- William Stein (2006-02-16) - large number of documentation and examples; improved code
- Mike Hansen (2007-3-25) - added differences and symmetric differences; fixed operators

EnumeratedSet(*X*)

Return the enumerated set associated to X.

The input object X must be finite.

EXAMPLES:

```
sage: EnumeratedSet([1,1,2,3])
{1, 2, 3}
sage: EnumeratedSet(ZZ)
...
ValueError: X (=Integer Ring) must be finite
```

Set(*X*)

Create the underlying set of X.

If X is a list, tuple, Python set, or `X.is_finite()` is true, this returns a wrapper around Python's enumerated immutable frozenset type with extra functionality. Otherwise it returns a more formal wrapper.

If you need the functionality of mutable sets, use Python's builtin set type.

EXAMPLES:

```
sage: X = Set(GF(9,'a'))
sage: X
{0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2}
sage: type(X)
<class 'sage.sets.set.Set_object_enumerated'>
sage: Y = X.union(Set(QQ))
sage: Y
Set-theoretic union of {0, 1, 2, a, a + 1, a + 2, 2*a, 2*a + 1, 2*a + 2} and Set of elements of
sage: type(Y)
<class 'sage.sets.set.Set_object_union'>
```

Usually sets can be used as dictionary keys.

```
sage: d={Set([2*I,1+I]):10}
sage: d                    # key is randomly ordered
{{I + 1, 2*I}: 10}
sage: d[Set([1+I,2*I])]
10
sage: d[Set((1+I,2*I))]
10
```

The original object is often forgotten.

```
sage: v = [1,2,3]
sage: X = Set(v)
sage: X
{1, 2, 3}
sage: v.append(5)
sage: X
{1, 2, 3}
sage: 5 in X
False
```

Set also accepts iterators, but be careful to only give *finite* sets.

```
sage: list(Set(iter([1, 2, 3, 4, 5])))
[1, 2, 3, 4, 5]
```

class Set_object(*X*)

A set attached to an almost arbitrary object.

EXAMPLES:

```
sage: K = GF(19)
sage: Set(K)
{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18}
sage: S = Set(K)

sage: latex(S)
\left\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18\right\}
sage: loads(S.dumps()) == S
True

sage: latex(Set(ZZ))
\Bold{Z}
```

cardinality()

Return the cardinality of this set, which is either an integer or Infinity.

EXAMPLES:

```
sage: Set(ZZ).cardinality()
+Infinity
sage: Primes().cardinality()
+Infinity
sage: Set(GF(5)).cardinality()
5
sage: Set(GF(5^2,'a')).cardinality()
25
```

difference(*X*)

Return the intersection of self and X.

EXAMPLES:

```
sage: X = Set(ZZ).difference(Primes())
sage: 4 in X
True
sage: 3 in X
False

sage: 4/1 in X
True

sage: X = Set(GF(9,'b')).difference(Set(GF(27,'c')))
sage: X
{0, 1, 2, b, b + 1, b + 2, 2*b, 2*b + 1, 2*b + 2}

sage: X = Set(GF(9,'b')).difference(Set(GF(27,'b')))
sage: X
{0, 1, 2, b, b + 1, b + 2, 2*b, 2*b + 1, 2*b + 2}
```

intersection(*X*)

Return the intersection of self and X.

EXAMPLES:

```
sage: X = Set(ZZ).intersection(Primes())
sage: 4 in X
False
sage: 3 in X
True

sage: 2/1 in X
True

sage: X = Set(GF(9,'b')).intersection(Set(GF(27,'c')))
sage: X
{}

sage: X = Set(GF(9,'b')).intersection(Set(GF(27,'b')))
sage: X
{}
```

is_finite()

EXAMPLES:

```
sage: Set(QQ).is_finite()
False
sage: Set(GF(250037)).is_finite()
True
sage: Set(Integers(2^1000000)).is_finite()
True
sage: Set([1,'a',ZZ]).is_finite()
True
```

object()

Return underlying object.

EXAMPLES:

```
sage: X = Set(QQ)
sage: X.object()
Rational Field
sage: X = Primes()
sage: X.object()
Set of all prime numbers: 2, 3, 5, 7, ...
```

subsets(*size=None*)

Return the Subset object representing the subsets of a set. If size is specified, return the subsets of that size.

EXAMPLES:

```
sage: X = Set([1,2,3])
sage: list(X.subsets())
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
sage: list(X.subsets(2))
[{1, 2}, {1, 3}, {2, 3}]
```

symmetric_difference(*X*)

Returns the symmetric difference of self and X.

EXAMPLES:

```
sage: X = Set([1,2,3]).symmetric_difference(Set([3,4]))
sage: X
{1, 2, 4}
```

union(*X*)
Return the union of self and X.

EXAMPLES:

```
sage: Set(QQ).union(Set(ZZ))
Set-theoretic union of Set of elements of Rational Field and Set of elements of Integer Ring
sage: Set(QQ) + Set(ZZ)
Set-theoretic union of Set of elements of Rational Field and Set of elements of Integer Ring
sage: X = Set(QQ).union(Set(GF(3))); X
Set-theoretic union of Set of elements of Rational Field and {0, 1, 2}
sage: 2/3 in X
True
sage: GF(3)(2) in X
True
sage: GF(5)(2) in X
False
sage: Set(GF(7)) + Set(GF(3))
{0, 1, 2, 3, 4, 5, 6, 1, 2, 0}
```

class **Set_object_difference**(*X*, *Y*)
Formal difference of two sets.

cardinality()
This tries to return the cardinality of this formal intersection.

Note that this is not likely to work in very much generality, and may just hang if either set involved is infinite.

EXAMPLES:

```
sage: X = Set(GF(13)).difference(Set(Primes()))
sage: X.cardinality()
8
```

class **Set_object_enumerated**(*X*)
A finite enumerated set.

cardinality()
EXAMPLES:

```
sage: Set([1,1]).cardinality()
1
```

difference(*other*)
Returns the set difference self-other.

EXAMPLES:

```
sage: X = Set([1,2,3,4])
sage: Y = Set([1,2])
sage: X.difference(Y)
{3, 4}
sage: Z = Set(ZZ)
sage: W = Set([2.5, 4, 5, 6])
sage: W.difference(Z)
{2.50000000000000}
```

frozenset()
Return the Python frozenset object associated to this set, which is an immutable set (hence hashable).

EXAMPLES:

```
sage: X = Set(GF(8,'c'))
sage: X
```

```
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: s = X.set(); s
set([0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1])
sage: hash(s)
...
TypeError: set objects are unhashable
sage: s = X.frozenset(); s
frozenset([0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1])
sage: hash(s)
-1390224788            # 32-bit
 561411537695332972    # 64-bit
sage: type(s)
<type 'frozenset'>
```

intersection(*other*)

Return the intersection of self and other.

EXAMPLES:

```
sage: X = Set(GF(8,'c'))
sage: Y = Set([GF(8,'c').0, 1, 2, 3])
sage: X.intersection(Y)
{1, c}
```

set()

Return the Python set object associated to this set.

Python has a notion of finite set, and often Sage sets have an associated Python set. This function returns that set.

EXAMPLES:

```
sage: X = Set(GF(8,'c'))
sage: X
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: X.set()
set([0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1])
sage: type(X.set())
<type 'set'>
sage: type(X)
<class 'sage.sets.set.Set_object_enumerated'>
```

symmetric_difference(*other*)

Returns the set difference self-other.

EXAMPLES:

```
sage: X = Set([1,2,3,4])
sage: Y = Set([1,2])
sage: X.symmetric_difference(Y)
{3, 4}
sage: Z = Set(ZZ)
sage: W = Set([2.5, 4, 5, 6])
sage: U = W.symmetric_difference(Z)
sage: 2.5 in U
True
sage: 4 in U
False
sage: V = Z.symmetric_difference(W)
sage: V == U
True
sage: 2.5 in V
```

```
True
sage: 6 in V
False
```

union(*other*)

Return the union of self and other.

EXAMPLES:

```
sage: X = Set(GF(8,'c'))
sage: Y = Set([GF(8,'c').0, 1, 2, 3])
sage: X
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1}
sage: Y
{1, c, 3, 2}
sage: X.union(Y)
{0, 1, c, c + 1, c^2, c^2 + 1, c^2 + c, c^2 + c + 1, 2, 3}
```

class Set_object_intersection(*X*, *Y*)

Formal intersection of two sets.

cardinality()

This tries to return the cardinality of this formal intersection.

Note that this is not likely to work in very much generality, and may just hang if either set involved is infinite.

EXAMPLES:

```
sage: X = Set(GF(13)).intersection(Set(ZZ))
sage: X.cardinality()
13
```

class Set_object_symmetric_difference(*X*, *Y*)

Formal symmetric difference of two sets.

cardinality()

This tries to return the cardinality of this formal symmetric difference.

Note that this is not likely to work in very much generality, and may just hang if either set involved is infinite.

EXAMPLES:

```
sage: X = Set(GF(13)).symmetric_difference(Set(range(5)))
sage: X.cardinality()
8
```

class Set_object_union(*X*, *Y*)

A formal union of two sets.

cardinality()

Return the cardinality of this set.

EXAMPLES:

```
sage: X = Set(GF(3)).union(Set(GF(2)))
sage: X
{0, 1, 2, 0, 1}
sage: X.cardinality()
5

sage: X = Set(GF(3)).union(Set(ZZ))
sage: X.cardinality()
+Infinity
```

is_Set(*x*)

Returns true if x is a Sage Set (not to be confused with a Python 2.4 set).

EXAMPLES:

```
sage: from sage.sets.set import is_Set
sage: is_Set([1,2,3])
False
sage: is_Set(set([1,2,3]))
False
sage: is_Set(Set([1,2,3]))
True
sage: is_Set(Set(QQ))
True
sage: is_Set(Primes())
True
```

10.11 The set of prime numbers

Primes()

Return the set of prime numbers.

EXAMPLES:

```
sage: P = Primes(); P
Set of all prime numbers: 2, 3, 5, 7, ...
```

We show various methods about the primes:

```
sage: P.cardinality()
+Infinity
sage: R = Primes()
sage: P == R
True
sage: 5 in P
True
sage: 100 in P
False
```

class **Primes_class**()

The set of prime numbers.

EXAMPLES:

```
sage: P = Primes(); P
Set of all prime numbers: 2, 3, 5, 7, ...
sage: loads(P.dumps()) == P
True
```

cardinality()

There is no largest prime number, so we say the set has infinite cardinality.

EXAMPLES:

```
sage: P = Primes()
sage: P.cardinality()
+Infinity
```

10.12 Families

A Family is an associative container which models a family $(f_i)_{i \in I}$. Then, `f[i]` returns the element of the family indexed by `i`. Whenever available, set and combinatorial class operations (counting, iteration, listing) on the family are induced from those of the index set. Families should be created through the `Family()` function.

AUTHORS:

- Nicolas Thiery (2008-02): initial release
- Florent Hivert (2008-04): various fixes, cleanups and improvements.

class AbstractFamily()

The abstract class for family

Any family belongs to a class which inherits from `AbstractFamily`.

hidden_keys()

Returns the hidden keys of the family, if any.

EXAMPLES:

```
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: f.hidden_keys()
[]
```

map(*f, name=None*)

Returns the family $(f(\texttt{self}[i]))_{i \in I}$, where I is the index set of self.

TODO: good name?

EXAMPLES:

```
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: g = f.map(lambda x: x+'1')
sage: list(g)
['a1', 'b1', 'd1']
```

zip(*f, other, name=None*)

Given two families with same index set I (and same hidden keys if relevant), returns the family $(f(self[i], other[i]))_{i \in I}$

TODO: generalize to any number of families and merge with map?

EXAMPLES:

```
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: g = Family({3: '1', 4: '2', 7: '3'})
sage: h = f.zip(lambda x,y: x+y, g)
sage: list(h)
['a1', 'b2', 'd3']
```

Family(*indices, function=None, hidden_keys=, [], hidden_function=None, lazy=False, name=None*)

A Family is an associative container which models a family $(f_i)_{i \in I}$. Then, f[i] returns the element of the family indexed by i. Whenever available, set and combinatorial class operations (counting, iteration, listing) on the family are induced from those of the index set.

There are several available implementations (classes) for different usages; Family serves as a factory, and will create instances of the appropriate classes depending on its arguments.

EXAMPLES:

In its simplest form, a list l or a tuple by itself is considered as the family $(l[i]_{i \in I})$ where I is the range $0 \ldots, len(l)$. So Family(l) returns the corresponding family.

```
sage: f = Family([1,2,3])
sage: f
Family (1, 2, 3)
sage: f = Family((1,2,3))
sage: f
Family (1, 2, 3)
```

A family can also be constructed from a dictionary t. The resulting family is very close to t, except that the elements of the family are the values of t. Here, we define the family $(f_i)_{i\in\{3,4,7\}}$ with f_3='a', f_4='b', and f_7='d':

```
sage: f = Family({3: 'a', 4: 'b', 7: 'd'})
sage: f
Finite family {3: 'a', 4: 'b', 7: 'd'}
sage: f[7]
'd'
sage: len(f)
3
sage: list(f)
['a', 'b', 'd']
sage: [ x for x in f ]
['a', 'b', 'd']
sage: f.keys()
[3, 4, 7]
sage: 'b' in f
True
sage: 'e' in f
False
```

A family can also be constructed by its index set I and a function f, as in $(f(i))_{i\in I}$:

```
sage: f = Family([3,4,7], lambda i: 2*i)
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f[7]
14
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

By default, all images are computed right away, and stored in an internal dictionary:

```
sage: f = Family((3,4,7), lambda i: 2*i)
sage: f
Finite family {3: 6, 4: 8, 7: 14}
```

Note that this requires all the elements of the list to be hashable. One can ask instead for the images $f(i)$ to be computed lazily, when needed:

```
sage: f = Family([3,4,7], lambda i: 2r*i, lazy=True)
sage: f
Lazy family (<lambda>(i))_{i in [3, 4, 7]}
sage: f[7]
```

```
14
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
```

This allows in particular for modeling infinite families:

```
sage: f = Family(ZZ, lambda i: 2r*i, lazy=True)
sage: f
Lazy family (<lambda>(i))_{i in Integer Ring}
sage: f.keys()
Integer Ring
sage: f[1]
2
sage: f[-5]
-10
sage: i = iter(f)
sage: i.next(), i.next(), i.next(), i.next(), i.next()
(0, 2, -2, 4, -4)
```

Note that the `lazy` keyword parameter is only needed to force laziness. Usually it is automatically set to a correct default value (ie: `False` for finite data structures and true for CombinatorialClasses:

```
sage: f == Family(ZZ, lambda i: 2r*i)
True
```

Beware that for those kind of families len(f) is not supposed to work. As a replacement, use the .cardinality() method:

```
sage: f = Family(Permutations(3), attrcall("to_lehmer_code"))
sage: list(f)
[[0, 0, 0], [0, 1, 0], [1, 0, 0], [1, 1, 0], [2, 0, 0], [2, 1, 0]]
sage: f.cardinality()
6
```

Caveat: Only certain families with lazy behavior can be pickled. In particular, only functions that work with Sage's pickle_function and unpickle_function (in sage.misc.fpickle) will correctly unpickle. The following two work:

```
sage: f = Family(Permutations(3), lambda p: p.to_lehmer_code()); f
Lazy family (<lambda>(i))_{i in Standard permutations of 3}
sage: f == loads(dumps(f))
True

sage: f = Family(Permutations(3), attrcall("to_lehmer_code")); f
Lazy family (i.to_lehmer_code())_{i in Standard permutations of 3}
sage: f == loads(dumps(f))
True
```

But this one don't:

```
sage: def plus_n(n): return lambda x: x+n
sage: f = Family([1,2,3], plus_n(3), lazy=True); f
Lazy family (<lambda>(i))_{i in [1, 2, 3]}
sage: f == loads(dumps(f))
...
ValueError: Cannot pickle code objects from closures
```

Finally, it can occasionally be useful to add some hidden elements in a family, which are accessible as f[i], but do not appear in the keys or the container operations:

```
sage: f = Family([3,4,7], lambda i: 2*i, hidden_keys=[2])
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f.hidden_keys()
[2]
sage: f[7]
14
sage: f[2]
4
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

The following example illustrates when the function is actually called:

```
sage: def compute_value(i):
...       print('computing 2*'+str(i))
...       return 2*i
sage: f = Family([3,4,7], compute_value, hidden_keys=[2])
computing 2*3
computing 2*4
computing 2*7
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f.hidden_keys()
[2]
sage: f[7]
14
sage: f[2]
computing 2*2
4
sage: f[2]
4
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

Here is a close variant where the function for the hidden keys is different from that for the other keys:

```
sage: f = Family([3,4,7], lambda i: 2*i, hidden_keys=[2], hidden_function = lambda i: 3*i)
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f.hidden_keys()
[2]
```

```
sage: f[7]
14
sage: f[2]
6
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

Family behaves the same way with FiniteCombinatorialClass instances and lists. This feature will eventually disappear when FiniteCombinatorialClass won't be needed anymore.

```
sage: f = Family(FiniteCombinatorialClass([1,2,3]))
sage: f
Combinatorial class with elements in [1, 2, 3]


sage: f = Family(FiniteCombinatorialClass([3,4,7]), lambda i: 2*i)
sage: f
Finite family {3: 6, 4: 8, 7: 14}
sage: f.keys()
[3, 4, 7]
sage: f[7]
14
sage: list(f)
[6, 8, 14]
sage: [x for x in f]
[6, 8, 14]
sage: len(f)
3
```

TESTS:

```
sage: f = Family({1:'a', 2:'b', 3:'c'})
sage: f
Finite family {1: 'a', 2: 'b', 3: 'c'}
sage: f[2]
'b'
sage: loads(dumps(f)) == f
True


sage: f = Family({1:'a', 2:'b', 3:'c'}, lazy=True)
Traceback (most recent call last):
ValueError: lazy keyword only makes sense together with function keyword !


sage: f = Family(range(1,27), lambda i: chr(i+96))
sage: f
    Finite family {1: 'a', 2: 'b', 3: 'c', 4: 'd', 5: 'e', 6: 'f', 7: 'g', 8: 'h', 9: 'i', 10: '
sage: f[2]
'b'
```

The factory `Family` is supposed to be idempotent. We test this feature here:

```
sage: from sage.sets.family import FiniteFamily, LazyFamily, TrivialFamily
sage: f = FiniteFamily({3: 'a', 4: 'b', 7: 'd'})
sage: g = Family(f)
sage: f == g
True

sage: f = Family([3,4,7], lambda i: 2r*i, hidden_keys=[2])
sage: g = Family(f)
sage: f == g
True

sage: f = LazyFamily([3,4,7], lambda i: 2r*i)
sage: g = Family(f)
sage: f == g
True

sage: f = TrivialFamily([3,4,7])
sage: g = Family(f)
sage: f == g
True
```

class FiniteFamily (*dictionary, keys=None*)

A FiniteFamily is an associative container which models a finite family $(f_i)_{i \in I}$. Its elements f_i are therefore its values. Instances should be created via the Family factory, which see for further examples and tests.

EXAMPLES: We define the family $(f_i)_{i \in \{3,4,7\}}$ with f_3=a, f_4=b, and f_7=d

```
sage: from sage.sets.family import FiniteFamily
sage: f = FiniteFamily({3: 'a', 4: 'b', 7: 'd'})
```

Individual elements are accessible as in a usual dictionary:

```
sage: f[7]
'd'
```

And the other usual dictionary operations are also available:

```
sage: len(f)
3
sage: f.keys()
[3, 4, 7]
```

However f behaves as a container for the f_i's:

```
sage: list(f)
['a', 'b', 'd']
sage: [ x for x in f ]
['a', 'b', 'd']
```

cardinality()

Returns the number of elements in self.

EXAMPLES:

```
sage: from sage.sets.family import FiniteFamily
sage: f = FiniteFamily({3: 'a', 4: 'b', 7: 'd'})
sage: f.cardinality()
3
```

class FiniteFamilyWithHiddenKeys(*dictionary, hidden_keys, hidden_function*)

A close variant of FiniteFamily where the family contains some hidden keys whose corresponding values are computed lazily (and remembered). Instances should be created via the Family factory, which see for examples and tests.

Caveat: Only instances of this class whose functions are compatible with sage.misc.fpickle can be pickled.

hidden_keys()

Returns self's hidden keys.

EXAMPLES:

```
sage: f = Family([3,4,7], lambda i: 2*i, hidden_keys=[2])
sage: f.hidden_keys()
[2]
```

class LazyFamily(*set, function, name=None*)

A LazyFamily(I, f) is an associative container which models the (possibly infinite) family $(f(i))_{i \in I}$.

Instances should be created via the Family factory, which see for examples and tests.

cardinality()

Return the number of elements in self.

EXAMPLES:

```
sage: from sage.sets.family import LazyFamily
sage: f = LazyFamily([3,4,7], lambda i: 2*i)
sage: f.cardinality()
3
```

keys()

Returns self's keys.

EXAMPLES:

```
sage: from sage.sets.family import LazyFamily
sage: f = LazyFamily([3,4,7], lambda i: 2*i)
sage: f.keys()
[3, 4, 7]
```

class TrivialFamily(*enumeration*)

`TrivialFamily(c)` turn the container c into a family indexed by the set $0, \ldots, len(c)$. The container c can be either a list or a tuple.

Instances should be created via the Family factory, which see for examples and tests.

cardinality()

Return the number of elements in self.

EXAMPLES:

```
sage: from sage.sets.family import TrivialFamily
sage: f = TrivialFamily([3,4,7])
sage: f.cardinality()
3
```

keys()

Returns self's keys.

EXAMPLES:

```
sage: from sage.sets.family import TrivialFamily
sage: f = TrivialFamily([3,4,7])
sage: f.keys()
[0, 1, 2]
```

10.13 Base class for parent objects

CLASS HIEARCHY:

```
SageObject
    CategoryObject
        Parent
```

TESTS: This came up in some subtle bug once.

```
sage: gp(2) + gap(3)
5
```

class EltPair()

class Parent()

Parents are the Sage/mathematical analogues of container objects in computer science.

Internal invariants: • self._element_init_pass_parent == guess_pass_parent(self, self._element_constructor) Ensures that self.__call__ passes down the parent properly to self._element_constructor. See #5979.

Hom()

Return the homspace `Hom(self, codomain, cat)` of all homomorphisms from self to codomain in the category cat. The default category is `category'()`.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: R.Hom(QQ)
Set of Homomorphisms from Multivariate Polynomial Ring in x, y over Rational Field to Ration
```

Homspaces are defined for very general Sage objects, even elements of familiar rings.

```
sage: n = 5; Hom(n,7)
Set of Morphisms from 5 to 7 in Category of elements of Integer Ring
sage: z=(2/3); Hom(z,8/1)
Set of Morphisms from 2/3 to 8 in Category of elements of Rational Field
```

This example illustrates the optional third argument:

```
sage: QQ.Hom(ZZ, Sets())
Set of Morphisms from Rational Field to Integer Ring in Category of sets
```

an_element()

Implementation of a function that returns an element (often non-trivial) of a parent object. This is cached. Parent structures that are should override `_an_element_()` instead.

EXAMPLES:

```
sage: CDF.an_element()
1.0*I
sage: ZZ[['t']].an_element()
t
```

coerce()

Return x as an element of self, if and only if there is a canonical coercion from the parent of x to self.

EXAMPLES:

```
sage: QQ.coerce(ZZ(2))
2
sage: ZZ.coerce(QQ(2))
...
TypeError: no canonical coercion from Rational Field to Integer Ring
```

We make an exception for zero:

```
sage: V = GF(7)^7
sage: V.coerce(0)
(0, 0, 0, 0, 0, 0, 0)
```

coerce_embedding()

Returns the embedding of self into some other parent, if such a parent exists.

This does not mean that there are no coercion maps from self into other fields, this is simply a specific morphism specified out of self and ususally denotes a special relationship (e.g. sub-objects, choice of completion, etc.)

EXAMPLES:

coerce_map_from()

This returns a Map object to coerce from S to self if one exists, or None if no such coercion exists.

EXAMPLES:

```
sage: ZZ.coerce_map_from(int)
Native morphism:
  From: Set of Python objects of type 'int'
  To:   Integer Ring
sage: QQ.coerce_map_from(ZZ)
Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

construction()

Returns a pair (functor, parent) such that functor(parent) return self. If this ring does not have a functorial construction, return None.

EXAMPLES:

```
sage: QQ.construction()
(FractionField, Integer Ring)
sage: f, R = QQ['x'].construction()
sage: f
Poly[x]
sage: R
Rational Field
sage: f(R)
Univariate Polynomial Ring in x over Rational Field
```

convert_map_from()

This function returns a Map from S to self, which may or may not succeed on all inputs. If a coercion map from S to self exists, then the it will be returned. If a coercion from self to S exists, then it will attempt to return a section of that map.

Under the new coercion model, this is the fastest way to convert elements of S to elements of self (short of manually constructing the elements) and is used by __call__.

EXAMPLES:

```
sage: m = ZZ.convert_map_from(QQ)
sage: m(-35/7)
-5
```

```
sage: parent(m(-35/7))
Integer Ring
```

get_action()

Returns an action of self on S or S on self.

To provide additional actions, override _get_action_().

TESTS:

```
sage: M = QQ['y']^3
sage: M.get_action(ZZ['x']['y'])
Right scalar multiplication by Univariate Polynomial Ring in y over Univariate Polynomial Ri
sage: M.get_action(ZZ['x']) # should be None
```

has_coerce_map_from()

Return True if there is a natural map from S to self. Otherwise, return False.

EXAMPLES:

```
sage: RDF.has_coerce_map_from(QQ)
True
sage: RDF.has_coerce_map_from(QQ['x'])
False
sage: RDF['x'].has_coerce_map_from(QQ['x'])
True
sage: RDF['x,y'].has_coerce_map_from(QQ['x'])
True
```

hom()

Return the unique homomorphism from self to codomain that sends `self.gens()` to the entries of `im_gens`. Raises a TypeError if there is no such homomorphism.

INPUT:

- `im_gens` - the images in the codomain of the generators of this object under the homomorphism
- `codomain` - the codomain of the homomorphism
- `check` - whether to verify that the images of generators extend to define a map (using only canonical coercisions).

OUTPUT:

- a homomorphism self –> codomain

Note: As a shortcut, one can also give an object X instead of `im_gens`, in which case return the (if it exists) natural map to X.

EXAMPLE: Polynomial Ring We first illustrate construction of a few homomorphisms involving a polynomial ring.

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = R.hom([5], QQ)
sage: f(x^2 - 19)
6

sage: R.<x> = PolynomialRing(QQ)
sage: f = R.hom([5], GF(7))
...
TypeError: images do not define a valid homomorphism

sage: R.<x> = PolynomialRing(GF(7))
sage: f = R.hom([3], GF(49,'a'))
sage: f
Ring morphism:
```

```
  From: Univariate Polynomial Ring in x over Finite Field of size 7
  To:   Finite Field in a of size 7^2
  Defn: x |--> 3
sage: f(x+6)
2
sage: f(x^2+1)
3
```

EXAMPLE: Natural morphism

```
sage: f = ZZ.hom(GF(5))
sage: f(7)
2
sage: f
Ring Coercion morphism:
  From: Integer Ring
  To:   Finite Field of size 5
```

There might not be a natural morphism, in which case a TypeError exception is raised.

```
sage: QQ.hom(ZZ)
...
TypeError: Natural coercion morphism from Rational Field to Integer Ring not defined.
```

is_exact()

Return True if elements of this ring are represented exactly, i.e., there is no precision loss when doing arithmetic.

NOTE: This defaults to true, so even if it does return True you have no guarantee (unless the ring has properly overloaded this).

EXAMPLES: sage: QQ.is_exact() True sage: ZZ.is_exact() True sage: Qp(7).is_exact() False sage: Zp(7, type='capped-abs').is_exact() False

list()

Return a list of all elements in this object, if possible (the object must define an iterator).

Set_PythonType()

Return the (unique) Parent that represents the set of Python objects of a specified type.

EXAMPLES:

```
sage: from sage.structure.parent import Set_PythonType
sage: Set_PythonType(list)
Set of Python objects of type 'list'
sage: Set_PythonType(list) is Set_PythonType(list)
True
sage: S = Set_PythonType(tuple)
sage: S([1,2,3])
(1, 2, 3)
```

class **Set_PythonType_class**()

cardinality()

EXAMPLES:

```
sage: S = sage.structure.parent.Set_PythonType(bool)
sage: S.cardinality()
2
sage: S = sage.structure.parent.Set_PythonType(int)
sage: S.cardinality()
4294967296                          # 32-bit
```

```
18446744073709551616              # 64-bit
sage: S = sage.structure.parent.Set_PythonType(float)
sage: S.cardinality()
18437736874454810627
sage: S = sage.structure.parent.Set_PythonType(long)
sage: S.cardinality()
+Infinity
```

object()
EXAMPLES:

```
sage: S = sage.structure.parent.Set_PythonType(tuple)
sage: S.object()
<type 'tuple'>
```

class **Set_generic**()
Abstract base class for sets.

category()
The category that this set belongs to, which is the category of all sets.
EXAMPLES:

```
sage: Set(QQ).category()
Category of sets
```

object()

is_Parent()
Return True if x is a parent object, i.e., derives from sage.structure.parent.Parent and False otherwise.
EXAMPLES:

```
sage: from sage.structure.parent import is_Parent
sage: is_Parent(2/3)
False
sage: is_Parent(ZZ)
True
sage: is_Parent(Primes())
True
```

normalize_names()
TESTS:

```
sage: sage.structure.parent.normalize_names(5, 'x')
('x0', 'x1', 'x2', 'x3', 'x4')
sage: sage.structure.parent.normalize_names(2, ['x','y'])
('x', 'y')
```

10.14 The Coercion Model

The coercion model manages how elements of one parent get related to elements of another. For example, the integer 2 can canonically be viewed as an element of the rational numbers. (The Parent of a non-element is its Python type.)

```
sage: ZZ(2).parent()
Integer Ring
sage: QQ(2).parent()
Rational Field
```

The most prominent role of the coercion model is to make sense of binary operations between elements that have distinct parents. It does this by finding a parent where both elements make sense, and doing the operation there. For example:

```
sage: a = 1/2; a.parent()
Rational Field
sage: b = ZZ[x].gen(); b.parent()
Univariate Polynomial Ring in x over Integer Ring
sage: a+b
x + 1/2
sage: (a+b).parent()
Univariate Polynomial Ring in x over Rational Field
```

If there is a coercion (see below) from one of the parents to the other, the operation is always performed in the codomain of that coercion. Otherwise a reasonable attempt to create a new parent with coercion maps from both original parents is made. The results of these discoveries are cached. On failure, a TypeError is always raised.

Some arithmetic operations (such as multiplication) can indicate an action rather than arithmetic in a common parent. For example:

```
sage: E = EllipticCurve('37a')
sage: P = E(0,0)
sage: 5*P
(1/4 : -5/8 : 1)
```

where there is action of $\mathbf{Z}$ on the points of E given by the additive group law. Parents can specify how they act on or are acted upon by other parents.

There are two kinds of ways to get from one parent to another, coercions and conversions.

Coercions are canonical (possibly modulo a finite number of deterministic choices) morphisms, and the set of all coercions between all parents forms a commuting diagram (modulo possibly rounding issues). $\mathbf{Z} \to \mathbf{Q}$ is an example of a coercion. These are invoked implicitly by the coercion model.

Conversions try to construct an element out of their input if at all possible. Examples include sections of coercions, creating an element from a string or list, etc. and may fail on some inputs of a given type while succeeding on others (i.e. they may not be defined on the whole domain). Conversions are always explicitly invoked, and never used by the coercion model to resolve binary operations.

For more information on how to specify coercions, conversions, and actions, see the documentation for Parent.

class CoercionModel_cache_maps()

See also sage.categories.pushout

EXAMPLES:

```
sage: f = ZZ['t','x'].0 + QQ['x'].0 + CyclotomicField(13).gen(); f
t + x + (zeta13)
sage: f.parent()
Multivariate Polynomial Ring in t, x over Cyclotomic Field of order 13 and degree 12
sage: ZZ['x','y'].0 + ~Frac(QQ['y']).0
(x*y + 1)/y
sage: MatrixSpace(ZZ['x'], 2, 2)(2) + ~Frac(QQ['x']).0
[(2*x + 1)/x           0]
[          0 (2*x + 1)/x]
sage: f = ZZ['x,y,z'].0 + QQ['w,x,z,a'].0; f
w + x
sage: f.parent()
Multivariate Polynomial Ring in w, x, y, z, a over Rational Field
```

```
sage: ZZ['x,y,z'].0 + ZZ['w,x,z,a'].1
2*x
```

AUTHOR:

- Robert Bradshaw

analyse()

Emulate the process of doing arithmetic between xp and yp, returning a list of steps and the parent that the result will live in. The `explain` function is easier to use, but if one wants access to the acutal morphism and action objects (rather than their string representations) then this is the function to use.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: steps, res = cm.analyse(GF(7), ZZ)
sage: print steps
['Coercion on right operand via', Natural morphism:
  From: Integer Ring
  To:   Finite Field of size 7, 'Arithmetic performed after coercions.']
sage: print res
Finite Field of size 7
sage: f = steps[1]; type(f)
<type 'sage.rings.integer_mod.Integer_to_IntegerMod'>
sage: f(100)
2
```

bin_op()

Execute the operation op on x and y. It first looks for an action corresponding to op, and failing that, it tries to coerces x and y into the a common parent and calls op on them.

If it cannot make sense of the operation, a TypeError is raised.

INPUT:

- `x` - the left operand
- `y` - the right operand
- `op` - a python function taking 2 arguments

  Note: op is often an arithmitic operation, but need not be so.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.bin_op(1/2, 5, operator.mul)
5/2
```

The operator can be any callable:

```
set Rational Field Integer Ring <function <lambda> at 0xc0b2270> None None
(Rational Field, Rational Field)
sage: R.<x> = ZZ['x']
sage: cm.bin_op(x^2-1, x+1, gcd)
x + 1
```

Actions are detected and performed:

```
sage: M = matrix(ZZ, 2, 2, range(4))
sage: V = vector(ZZ, [5,7])
sage: cm.bin_op(M, V, operator.mul)
(7, 31)
```

TESTS:

```
sage: class Foo:
...       def __rmul__(self, left):
...           return 'hello'
...
sage: H = Foo()
sage: print int(3)*H
hello
sage: print Integer(3)*H
hello
sage: print H*3
...
TypeError: unsupported operand parent(s) for '*': '<type 'instance'>' and 'Integer Ring'

sage: class Nonsense:
...       def __init__(self, s):
...           self.s = s
...       def __repr__(self):
...           return self.s
...       def __mul__(self, x):
...           return Nonsense(self.s + chr(x%256))
...       __add__ = __mul__
...       def __rmul__(self, x):
...           return Nonsense(chr(x%256) + self.s)
...       __radd__ = __rmul__
...
sage: a = Nonsense('blahblah')
sage: a*80
blahblahP
sage: 80*a
Pblahblah
sage: a+80
blahblahP
sage: 80+a
Pblahblah
```

canonical_coercion()

Given two elements x and y, with parents S and R respectively, find a common parent Z such that there are coercions $f : S \mapsto Z$ and $g : R \mapsto Z$ and return $f(x), g(y)$ which will have the same parent.

Raises a type error if no such Z can be found.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.canonical_coercion(mod(2, 10), 17)
(2, 7)
sage: x, y = cm.canonical_coercion(1/2, matrix(ZZ, 2, 2, range(4)))
sage: x
[1/2   0]
[  0 1/2]
sage: y
[0 1]
[2 3]
sage: parent(x) is parent(y)
True
```

There is some support for non-Sage datatypes as well:

```
sage: x, y = cm.canonical_coercion(int(5), 10)
sage: type(x), type(y)
(<type 'sage.rings.integer.Integer'>, <type 'sage.rings.integer.Integer'>)
```

```
sage: x, y = cm.canonical_coercion(int(5), complex(3))
sage: type(x), type(y)
(<type 'complex'>, <type 'complex'>)

sage: class MyClass:
...       def _sage_(self):
...           return 13
sage: a, b = cm.canonical_coercion(MyClass(), 1/3)
sage: a, b
(13, 1/3)
sage: type(a)
<type 'sage.rings.rational.Rational'>
```

We also make an exception for 0, even if ZZ does not map in:

```
sage: canonical_coercion(vector([1, 2, 3]), 0)
((1, 2, 3), (0, 0, 0))
```

coercion_maps()

Give two parents R and S, return a pair of coercion maps $f : R \to Z$ and $g : S \to Z$, if such a Z can be found.

In the (common) case that $R = Z$ or $S = Z$ then `None` is returned for f or g respectively rather than constructing (and subsequently calling) the identity morphism.

If no suitable f, g can be found, a single None is returned. This result is cached.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: f, g = cm.coercion_maps(ZZ, QQ)
sage: print f
Natural morphism:
  From: Integer Ring
  To:   Rational Field
sage: print g
None

sage: f, g = cm.coercion_maps(ZZ['x'], QQ)
sage: print f
Conversion map:
  From: Univariate Polynomial Ring in x over Integer Ring
  To:   Univariate Polynomial Ring in x over Rational Field
sage: print g
Polynomial base injection morphism:
  From: Rational Field
  To:   Univariate Polynomial Ring in x over Rational Field

sage: cm.coercion_maps(QQ, GF(7)) == None
True
```

Note that to break symmetry, if there is a coercion map in both directions, the parent on the left is used:

```
sage: V = QQ^3
sage: W = V.__class__(QQ, 3)
sage: V == W
True
sage: V is W
False
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.coercion_maps(V, W)
```

```
(None,
 Call morphism:
  From: Vector space of dimension 3 over Rational Field
  To:   Vector space of dimension 3 over Rational Field)
sage: cm.coercion_maps(W, V)
(None,
 Call morphism:
  From: Vector space of dimension 3 over Rational Field
  To:   Vector space of dimension 3 over Rational Field)
sage: v = V([1,2,3])
sage: w = W([1,2,3])
sage: parent(v+w) is V
True
sage: parent(w+v) is W
True
```

discover_action()

INPUT

- R - the left Parent (or type)
- S - the right Parent (or type)
- op - the operand, typically an element of the operator module.

OUTPUT:

- An action A such that s op r is given by A(s,r).

The steps taken are illistrated below.

EXAMPLES:

```
sage: P.<x> = ZZ['x']
sage: P.get_action(ZZ)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Integer
sage: ZZ.get_action(P) is None
True
sage: cm = sage.structure.element.get_coercion_model()
```

If R or S is a Parent, ask it for an action by/on R:

```
sage: cm.discover_action(ZZ, P, operator.mul)
Left scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Integer R
```

If R or S a type, recursively call get_action with the Sage versions of R and/or S:

```
sage: cm.discover_action(P, int, operator.mul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Integer
with precomposition on right by Native morphism:
  From: Set of Python objects of type 'int'
  To:   Integer Ring
```

If op in an inplace operation, look for the non-inplace action:

```
sage: cm.discover_action(P, ZZ, operator.imul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Integer
```

If op is division, look for action on right by inverse:

```
sage: cm.discover_action(P, ZZ, operator.div)
Right inverse action by Rational Field on Univariate Polynomial Ring in x over Integer Ring
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

discover_coercion()

This actually implements the finding of coercion maps as described in the `coercion_maps` method.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
```

If R is S, then two identity morphisms suffice:

```
sage: cm.discover_coercion(SR, SR)
(None, None)
```

If there is a coercion map either direction, use that:

```
sage: cm.discover_coercion(ZZ, QQ)
(Natural morphism:
  From: Integer Ring
  To:   Rational Field, None)
sage: cm.discover_coercion(RR, QQ)
(None,
 Generic map:
  From: Rational Field
  To:   Real Field with 53 bits of precision)
```

Otherwise, try and compute an appropriate cover:

```
sage: cm.discover_coercion(ZZ['x,y'], RDF)
(Call morphism:
  From: Multivariate Polynomial Ring in x, y over Integer Ring
  To:   Multivariate Polynomial Ring in x, y over Real Double Field,
 Call morphism:
  From: Real Double Field
  To:   Multivariate Polynomial Ring in x, y over Real Double Field)
```

Sometimes there is a reasonable "cover," but no canonical coercion:

```
sage: sage.categories.pushout.pushout(QQ, QQ^3)
Vector space of dimension 3 over Rational Field
sage: print cm.discover_coercion(QQ, QQ^3)
None
```

division_parent()

Deduces where the result of division in parent lies by calculating the inverse of `parent.one_element()` or `parent.an_element()`.

The result is cached.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.division_parent(ZZ)
Rational Field
sage: cm.division_parent(QQ)
Rational Field
sage: cm.division_parent(ZZ['x'])
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: cm.division_parent(GF(41))
Finite Field of size 41
sage: cm.division_parent(Integers(100))
Ring of integers modulo 100
sage: cm.division_parent(SymmetricGroup(5))
Symmetric group of order 5! as a permutation group
```

exception_stack()

Returns the list of exceptions that were caught in the course of executing the last binary operation. Useful

for diagnosis when user-defined maps or actions raise exceptions that are caught in the course of coercion detection.

If all went well, this should be the empty list. If things aren't happening as you expect, this is a good place to check. See also `coercion_traceback()`.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: 1/2 + 2
5/2
sage: cm.exception_stack()
[]
sage: 1/2 + GF(3)(2)
...
TypeError: unsupported operand parent(s) for '+': 'Rational Field' and 'Finite Field of size
```

Now see what the actual problem was:

```
sage: import traceback
sage: cm.exception_stack()
[(<type 'exceptions.TypeError'>, TypeError("BUG: the base_extend method must be defined for
sage: print ''.join(sum([traceback.format_exception(*info) for info in cm.exception_stack()]
...
TypeError: no common canonical parent for objects with parents: 'Rational Field' and 'Finite
```

This is typically accessed via the `coercion_traceback()` function.

```
sage: coercion_traceback()
...
TypeError: no common canonical parent for objects with parents: 'Rational Field' and 'Finite
```

explain()

This function can be used to understand what coercions will happen for an arithmetic operation between xp and yp (which may be either elements or parents). If the parent of the result can be determined then it will be returned.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()

sage: cm.explain(ZZ, ZZ)
Identical parents, arithmetic performed immediately.
Result lives in Integer Ring
Integer Ring

sage: cm.explain(QQ, int)
Coercion on right operand via
    Native morphism:
      From: Set of Python objects of type 'int'
      To:   Rational Field
Arithmetic performed after coercions.
Result lives in Rational Field
Rational Field

sage: cm.explain(ZZ['x'], QQ)
Action discovered.
    Right scalar multiplication by Rational Field on Univariate Polynomial Ring in x over In
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field

sage: cm.explain(ZZ['x'], QQ, operator.add)
Coercion on left operand via
```

```
    Conversion map:
      From: Univariate Polynomial Ring in x over Integer Ring
      To:   Univariate Polynomial Ring in x over Rational Field
Coercion on right operand via
    Polynomial base injection morphism:
      From: Rational Field
      To:   Univariate Polynomial Ring in x over Rational Field
Arithmetic performed after coercions.
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field
```

Sometimes with non-sage types there is not enough information to deduce what will actually happen:

```
sage: cm.explain(RealField(100), float, operator.add)
Right operand is numeric, will attempt coercion in both directions.
Unknown result parent.
sage: parent(RealField(100)(1) + float(1))
<type 'float'>
sage: cm.explain(QQ, float, operator.add)
Right operand is numeric, will attempt coercion in both directions.
Unknown result parent.
sage: parent(QQ(1) + float(1))
<type 'float'>
```

Special care is taken to deal with division:

```
sage: cm.explain(ZZ, ZZ, operator.div)
Identical parents, arithmetic performed immediately.
Result lives in Rational Field
Rational Field

sage: cm.explain(ZZ['x'], QQ['x'], operator.div)
Coercion on left operand via
    Conversion map:
      From: Univariate Polynomial Ring in x over Integer Ring
      To:   Univariate Polynomial Ring in x over Rational Field
Arithmetic performed after coercions.
Result lives in Fraction Field of Univariate Polynomial Ring in x over Rational Field
Fraction Field of Univariate Polynomial Ring in x over Rational Field

sage: cm.explain(int, ZZ, operator.div)
Coercion on left operand via
    Native morphism:
      From: Set of Python objects of type 'int'
      To:   Integer Ring
Arithmetic performed after coercions.
Result lives in Rational Field
Rational Field

sage: cm.explain(ZZ['x'], ZZ, operator.div)
Action discovered.
    Right inverse action by Rational Field on Univariate Polynomial Ring in x over Integer R
    with precomposition on right by Natural morphism:
      From: Integer Ring
      To:   Rational Field
Result lives in Univariate Polynomial Ring in x over Rational Field
Univariate Polynomial Ring in x over Rational Field
```

Note: This function is accurate only in so far as analyse is kept in sync with the `bin_op()` and `canonical_coercion()` which are kept seperate for maximal efficiency.

get_action()

Get the action of R on S or S on R associated to the operation op.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.get_action(ZZ['x'], ZZ, operator.mul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Integer
sage: cm.get_action(ZZ['x'], ZZ, operator.imul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Integer
sage: cm.get_action(ZZ['x'], QQ, operator.mul)
Right scalar multiplication by Rational Field on Univariate Polynomial Ring in x over Intege
sage: cm.get_action(QQ['x'], int, operator.mul)
Right scalar multiplication by Integer Ring on Univariate Polynomial Ring in x over Rational
with precomposition on right by Native morphism:
  From: Set of Python objects of type 'int'
  To:   Integer Ring

sage: R.<x> = QQ['x']
sage: A = cm.get_action(R, ZZ, operator.div); A
Right inverse action by Rational Field on Univariate Polynomial Ring in x over Rational Fiel
with precomposition on right by Natural morphism:
  From: Integer Ring
  To:   Rational Field
sage: A(x+10, 5)
1/5*x + 2
```

get_cache()

This returns the current cache of coercion maps and actions, primarily useful for debugging and introspection.

EXAMPLES:

```
sage: 1 + 1/2
3/2
sage: cm = sage.structure.element.get_coercion_model()
sage: maps, actions = cm.get_cache()
```

Now lets see what happens when we do a binary operations with an integer and a rational:

```
sage: left_morphism, right_morphism = maps[ZZ, QQ]
sage: print left_morphism
Natural morphism:
  From: Integer Ring
  To:   Rational Field
sage: print right_morphism
None
```

We can see that it coerces the left operand from an integer to a rational, and doesn't do anything to the right.

Now for some actions:

```
sage: R.<x> = ZZ['x']
sage: 1/2 * x
1/2*x
sage: maps, actions = cm.get_cache()
sage: act = actions[QQ, R, operator.mul]; act
Left scalar multiplication by Rational Field on Univariate Polynomial Ring in x over Integer
sage: act.actor()
Rational Field
sage: act.domain()
Univariate Polynomial Ring in x over Integer Ring
```

```
sage: act.codomain()
Univariate Polynomial Ring in x over Rational Field
sage: act(1/5, x+10)
1/5*x + 2
```

get_stats()

This returns the state of the cache of coercion maps and actions, primarily useful for debugging and introspection. If a class is not part of the coercion system, we should call the __rmul__ method when it makes sense.

The coercion maps are stored in a specialized TripleDict hashtable, and the stats returned are (min, avg, max) of the number of items per bucket. The lower the better.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.get_stats()                # random
((0, 0.16058394160583941, 2), (0, 0.13138686131386862, 3))
```

reset_cache()

Clear the coercion cache.

This should have no impact on the result of arithmetic operations, as the exact same coercions and actions will be re-discovered when needed.

It may be useful for debugging, and may also free some memory.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.get_stats()                # random
((0, 0.3307086614173229, 3), (0, 0.1496062992125984, 2))
sage: cm.reset_cache()
sage: cm.get_stats()
((0, 0.0, 0), (0, 0.0, 0))
```

verify_action()

Verify that `action` takes an element of R on the left and S on the right, raising an error if not.

This is used for consistency checking in the coercion model.

EXAMPLES:

```
sage: R.<x> = ZZ['x']
sage: cm = sage.structure.element.get_coercion_model()
sage: cm.verify_action(R.get_action(QQ), R, QQ, operator.mul)
Right scalar multiplication by Rational Field on Univariate Polynomial Ring in x over Intege
sage: cm.verify_action(R.get_action(QQ), RDF, R, operator.mul)
...
RuntimeError: There is a BUG in the coercion model:
    Action found for R <built-in function mul> S does not have the correct domains
    R = Real Double Field
    S = Univariate Polynomial Ring in x over Integer Ring
    (should be Univariate Polynomial Ring in x over Integer Ring, Rational Field)
    action = Right scalar multiplication by Rational Field on Univariate Polynomial Ring in
```

verify_coercion_maps()

Make sure this is a valid pair of homomorphisms from R and S to a common parent. This function is used to protect the user against buggy parents.

EXAMPLES:

```
sage: cm = sage.structure.element.get_coercion_model()
sage: homs = QQ.coerce_map_from(ZZ), None
sage: cm.verify_coercion_maps(ZZ, QQ, homs) == homs
```

```
        True
        sage: homs = QQ.coerce_map_from(ZZ), RR.coerce_map_from(QQ)
        sage: cm.verify_coercion_maps(ZZ, QQ, homs) == homs
        ...
        RuntimeError: ('BUG in coercion model, codomains must be identical', Natural morphism:
          From: Integer Ring
          To:   Rational Field, Generic map:
          From: Rational Field
          To:   Real Field with 53 bits of precision)
```

parent()

py_scalar_parent()

Returns the Sage equivalent of the given python type, if one exists. If there is no equivalent, return None.

EXAMPLES:

```
sage: from sage.structure.coerce import py_scalar_parent
sage: py_scalar_parent(int)
Integer Ring
sage: py_scalar_parent(long)
Integer Ring
sage: py_scalar_parent(float)
Real Double Field
sage: py_scalar_parent(complex)
Complex Double Field
sage: py_scalar_parent(dict),
(None,)
```

10.15 Coerce actions

class IntegerMulAction()

class LAction()

class LeftModuleAction()

class ModuleAction()

codomain()

The codomain of self, which may or may not be equal to the domain.

EXAMPLES:

```
sage: from sage.structure.coerce_actions import LeftModuleAction
sage: A = LeftModuleAction(QQ, ZZ['x,y,z'])
sage: A.codomain()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

domain()

The domain of self, which is the module that is being acted on.

EXAMPLES:

```
sage: from sage.structure.coerce_actions import LeftModuleAction
sage: A = LeftModuleAction(QQ, ZZ['x,y,z'])
sage: A.domain()
Multivariate Polynomial Ring in x, y, z over Integer Ring
```

class PyScalarAction()

class RAction()

class RightModuleAction()

is_inplace

parent()

10.16 Coerce maps

class CCallableConvertMap_class()

class CallableConvertMap()

class DefaultConvertMap()
This morphism simply calls the codomain's element_constructor method, passing in the codomain as the first argument.

class DefaultConvertMap_unique()
This morphism simply defers action to the codomain's element_constructor method, WITHOUT passing in the codomain as the first argument.

This is used for creating elements that don't take a parent as the first argument to their __init__ method, for example, Integers, Rationals, Algebraic Reals... all have a unique parent. It is also used when the element_constructor is a bound method (whose self argument is assumed to be bound to the codomain).

class ListMorphism()

class NamedConvertMap()
This is used for creating a elements via the _xxx_ methods.

For example, many elements implement an _integer_ method to convert to ZZ, or a _rational_ method to convert to QQ.

method_name

class TryMap()

test_CCallableConvertMap()
For testing CCallableConvertMap_class.

TESTS:

```
sage: from sage.structure.coerce_maps import test_CCallableConvertMap
sage: f = test_CCallableConvertMap(ZZ, 'test'); f
Conversion via c call 'test' map:
  From: Integer Ring
  To:   Integer Ring
sage: f(3)
24
sage: f(9)
720
```

CHAPTER

ELEVEN

MISCELLANEOUS

11.1 Miscellaneous functions

AUTHORS:

- William Stein
- William Stein (2006-04-26): added workaround for Windows where most users's home directory has a space in it.
- Robert Bradshaw (2007-09-20): Ellipsis range/iterator.

class AttrCallObject (*name, args, kwds*)

alarm (*seconds*)
Raise a KeyboardInterrupt exception in a given number of seconds. This is useful for automatically interrupting long computations and can be trapped using exception handling (just catch KeyboardInterrupt).

INPUT:

- `seconds` - integer

TESTS:

```
sage: try: alarm(1); sleep(2)
... except KeyboardInterrupt: print "Alarm went off"
Alarm went off
```

assert_attribute (*x, attr, init=None*)
If the object x has the attribute attr, do nothing. If not, set x.attr to init.

attrcall (*name, *args, **kwds*)
Returns a callable which takes in an object, gets the method named name from that object, and calls it with the specified arguments and keywords.

INPUT:

- `name` - a string of the name of the method you want to call
- `args, kwds` - arguments and keywords to be passed to the methdod

EXAMPLES:

```
sage: f = attrcall('r_core', 3); f
*.r_core(3)
sage: [f(p) for p in Partitions(5)]
[[2], [1, 1], [1, 1], [3, 1, 1], [2], [2], [1, 1]]
```

branch_current_hg()
Return the current hg Mercurial branch name. If the branch is 'main', which is the default branch, then just '' is returned.

branch_current_hg_notice(*branch*)
Return a string describing the current branch and that the library is being loaded. This is called by the `SAGE_ROOT/local/bin/sage-sage` script.

INPUT:

- `string` - a representation of the name of the Sage library branch.

OUTPUT: string

Note: If the branch is main, then return an empty string.

class **cached_attribute**(*method, name=None*)
Computes attribute value and caches it in the instance.

class **cached_class_attribute**(*method, name=None*)
Computes attribute value and caches it in the class.

cancel_alarm()

cmp_props(*left, right, props*)

coeff_repr(*c, is_latex=False*)

cputime(*t=0*)
Return the time in CPU second since Sage started, or with optional argument t, return the time since time t. This is how much time Sage has spent using the CPU. It does not count time spent by subprocesses spawned by Sage (e.g., Gap, Singular, etc.).

This is done via a call to resource.getrusage, so it avoids the wraparound problems in time.clock() on Cygwin.

INPUT:

- `t` - (optional) float, time in CPU seconds

OUTPUT:

- `float` - time in CPU seconds

EXAMPLES:

```
sage: t = cputime()
sage: F = factor(2^199-1)
sage: cputime(t)          # somewhat random
0.29000000000000004


sage: w = walltime()
sage: F = factor(2^199-1)
sage: walltime(w)         # somewhat random
0.8823847770690918
```

delete_tmpfiles()

deprecation(*message*)
Issue a deprecation warning.

EXAMPLE:

```
sage: def foo():
...     sage.misc.misc.deprecation("The function foo is replaced by bar.")
...
sage: def my_function():
...     foo()
...
sage: my_function() # random: I don't know how to test the output.
doctest:1: DeprecationWarning: The function foo is replaced by bar.
```

ellipsis_iter(**args, **kwds*)

Same as ellipsis_range, but as an iterator (and may end with an Ellipsis).

See also ellipsis_range.

Use (1,2,...) notation.

EXAMPLES:

```
sage: A = ellipsis_iter(1,2,Ellipsis)
sage: [A.next() for _ in range(10)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: A.next()
11
sage: A = ellipsis_iter(1,3,5,Ellipsis)
sage: [A.next() for _ in range(10)]
[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
sage: A = ellipsis_iter(1,2,Ellipsis,5,10,Ellipsis)
sage: [A.next() for _ in range(10)]
[1, 2, 3, 4, 5, 10, 11, 12, 13, 14]
```

TESTS:

These were carefully chosen tests, only to be changed if the semantics of ellipsis ranges change. In otherwords, if they don't pass it's probably a bug in the implementation, not in the doctest.

```
sage: list(1,..,10)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: list(1,3,..,10)
[1, 3, 5, 7, 9]
sage: list(1,..,10,..,20)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
sage: list(1,3,..,10,..,20)
[1, 3, 5, 7, 9, 10, 12, 14, 16, 18, 20]
sage: list(1,3,..,10,10,..,20)
[1, 3, 5, 7, 9, 10, 12, 14, 16, 18, 20]
sage: list(0,2,..,10,10,..,20,20,..,25)
[0, 2, 4, 6, 8, 10, 10, 12, 14, 16, 18, 20, 20, 22, 24]
sage: list(10,..,1)
[]
sage: list(10,11,..,1)
[]
sage: list(10,9,..,1)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
sage: list(100,..,10,..,20)
[10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
sage: list(0,..,10,..,-20)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: list(100,..,10,..,-20)
[]
```

```
sage: list(100,102,..,10,..,20)
[10, 12, 14, 16, 18, 20]
```

ellipsis_range(**args, **kwds*)

Return arithmetic sequence determined by the numeric arguments and ellipsis. Best illustrated by examples.

Use [1,2,..,n] notation.

EXAMPLES:

```
sage: ellipsis_range(1,Ellipsis,11,100)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 100]
sage: ellipsis_range(0,2,Ellipsis,10,Ellipsis,20)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
sage: ellipsis_range(0,2,Ellipsis,11,Ellipsis,20)
[0, 2, 4, 6, 8, 10, 11, 13, 15, 17, 19]
sage: ellipsis_range(0,2,Ellipsis,11,Ellipsis,20, step=3)
[0, 2, 5, 8, 11, 14, 17, 20]
sage: ellipsis_range(10,Ellipsis,0)
[]
```

TESTS: These were carefully chosen tests, only to be changed if the semantics of ellipsis ranges change. In otherwords, if they don't pass it's probably a bug in the implementation, not in the doctest.

Note 10 only appears once (though it is in both ranges).

```
sage: ellipsis_range(0,Ellipsis,10,Ellipsis,20,step=2)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Sometimes one or more ranges is empty.

```
sage: ellipsis_range(100,Ellipsis,10,Ellipsis,20,step=2)
[10, 12, 14, 16, 18, 20]
sage: ellipsis_range(0,Ellipsis,10,Ellipsis,-20,step=2)
[0, 2, 4, 6, 8, 10]
sage: ellipsis_range(100,Ellipsis,10,Ellipsis,-20,step=2)
[]
```

We always start on the leftmost point of the range.

```
sage: ellipsis_range(0,Ellipsis,10,Ellipsis,20,step=3)
[0, 3, 6, 9, 10, 13, 16, 19]
sage: ellipsis_range(100,Ellipsis,10,Ellipsis,20,step=3)
[10, 13, 16, 19]
sage: ellipsis_range(0,Ellipsis,10,Ellipsis,-20,step=3)
[0, 3, 6, 9]
sage: ellipsis_range(100,Ellipsis,10,Ellipsis,-20,step=3)
[]
sage: ellipsis_range(0,1,Ellipsis,-10)
[]
sage: ellipsis_range(0,1,Ellipsis,-10,step=1)
[0]
sage: ellipsis_range(100,0,1,Ellipsis,-10)
[100]
```

Note the duplicate 5 in the output.

```
sage: ellipsis_range(0,Ellipsis,5,5,Ellipsis,10)
[0, 1, 2, 3, 4, 5, 5, 6, 7, 8, 9, 10]
```

Examples in which the step determines the parent of the elements:

```
sage: [1..3, step=0.5]
[1.00000000000000, 1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000]
sage: v = [1..5, step=1/1]; v
[1, 2, 3, 4, 5]
sage: parent(v[2])
Rational Field
```

embedded()

Return True if this copy of Sage is running embedded in the Sage notebook.

EXAMPLES:

```
sage: sage.misc.misc.embedded()     # output True if in the notebook
False
```

exists(*S, P*)

If S contains an element x such that P(x) is True, this function returns True and the element x. Otherwise it returns False and None.

Note that this function is NOT suitable to be used in an if-statement or in any place where a boolean expression is expected. For those situations, use the Python built-in

any(P(x) for x in S)

INPUT:

- `S` - object (that supports enumeration)
- `P` - function that returns True or False

OUTPUT:

- `bool` - whether or not P is True for some element x of S
- `object` - x

EXAMPLES: lambda functions are very useful when using the exists function:

```
sage: exists([1,2,5], lambda x : x > 7)
(False, None)
sage: exists([1,2,5], lambda x : x > 3)
(True, 5)
```

The following example is similar to one in the MAGMA handbook. We check whether certain integers are a sum of two (small) cubes:

```
sage: cubes = [t**3 for t in range(-10,11)]
sage: exists([(x,y) for x in cubes for y in cubes], lambda v : v[0]+v[1] == 218)
(True, (-125, 343))
sage: exists([(x,y) for x in cubes for y in cubes], lambda v : v[0]+v[1] == 219)
(False, None)
```

forall(*S, P*)

If P(x) is true every x in S, return True and None. If there is some element x in S such that P is not True, return False and x.

Note that this function is NOT suitable to be used in an if-statement or in any place where a boolean expression is expected. For those situations, use the Python built-in

all(P(x) for x in S)

INPUT:

- S - object (that supports enumeration)
- P - function that returns True or False

OUTPUT:

- bool - whether or not P is True for all elements of S
- object - x

EXAMPLES: lambda functions are very useful when using the forall function. As a toy example we test whether certain integers are greater than 3.

```
sage: forall([1,2,5], lambda x : x > 3)
(False, 1)
sage: forall([1,2,5], lambda x : x > 0)
(True, None)
```

Next we ask whether every positive integer less than 100 is a product of at most 2 prime factors:

```
sage: forall(range(1,100),  lambda n : len(factor(n)) <= 2)
(False, 30)
```

The answer is no, and 30 is a counterexample. However, every positive integer 100 is a product of at most 3 primes.

```
sage: forall(range(1,100),  lambda n : len(factor(n)) <= 3)
(True, None)
```

generic_cmp(*x, y*)

Compare x and y and return -1, 0, or 1.

This is similar to x.__cmp__(y), but works even in some cases when a .__cmp__ method isn't defined.

get_verbose()

Return the global Sage verbosity level.

INPUT: int level: an integer between 0 and 2, inclusive.

OUTPUT: changes the state of the verbosity flag.

EXAMPLES:

```
sage: get_verbose()
0
sage: set_verbose(2)
sage: get_verbose()
2
sage: set_verbose(0)
```

get_verbose_files()

getitem(*v, n*)

Variant of getitem that coerces to an int if a TypeError is raised.

(This is not needed anymore - classes should define an __index__ method.)

Thus, e.g., getitem(v,n) will work even if v is a Python list and n is a Sage integer.

EXAMPLES:

```
sage: v = [1,2,3]
```

The following used to fail in Sage <= 1.3.7. Now it works fine:

```
sage: v[ZZ(1)]
2
```

This always worked.

```
sage: getitem(v, ZZ(1))
2
```

graphics_filename(*ext='png'*)
Return the next available canonical filename for a plot/graphics file.

is_in_string(*line, pos*)
Returns True if the character at position pos in line occurs within a string.

EXAMPLES: sage: from sage.misc.misc import is_in_string sage: line = 'test('#')' sage: is_in_string(line, line.rfind('#')) True sage: is_in_string(line, line.rfind(')')) False

class lazy_prop(*calculate_function*)

newton_method_sizes(*N*)
Returns a sequence of integers $1 = a_1 \leq a_2 \leq \cdots \leq a_n = N$ such that $a_j = \lceil a_{j+1}/2 \rceil$ for all j.

This is useful for Newton-style algorithms that double the precision at each stage. For example if you start at precision 1 and want an answer to precision 17, then it's better to use the intermediate stages 1, 2, 3, 5, 9, 17 than to use 1, 2, 4, 8, 16, 17.

INPUT:

- N - positive integer

EXAMPLES:

```
sage: newton_method_sizes(17)
[1, 2, 3, 5, 9, 17]
sage: newton_method_sizes(16)
[1, 2, 4, 8, 16]
sage: newton_method_sizes(1)
[1]
```

AUTHORS:

- David Harvey (2006-09-09)

pad_zeros(*s, size=3*)
EXAMPLES:

```
sage: pad_zeros(100)
'100'
sage: pad_zeros(10)
'010'
sage: pad_zeros(10, 5)
'00010'
sage: pad_zeros(389, 5)
'00389'
sage: pad_zeros(389, 10)
'0000000389'
```

powerset(*X*)
Iterator over the *list* of all subsets of the iterable X, in no particular order. Each list appears exactly once, up to order.

INPUT:

- X - an iterable

OUTPUT: iterator of lists

EXAMPLES:

```
sage: list(powerset([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
sage: [z for z in powerset([0,[1,2]])]
[[], [0], [[1, 2]], [0, [1, 2]]]
```

Iterating over the power set of an infinite set is also allowed:

```
sage: i = 0
sage: for x in powerset(ZZ):
...     if i > 10:
...         break
...     else:
...         i += 1
...     print x,
[] [0] [1] [0, 1] [-1] [0, -1] [1, -1] [0, 1, -1] [2] [0, 2] [1, 2]
```

You may also use subsets as an alias for powerset:

```
sage: subsets([1,2,3])   # random object location in output
<generator object at 0xaeae418c>
sage: list(subsets([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

The reason we return lists instead of sets is that the elements of
sets must be hashable and many structures on which one wants the
powerset consist of non-hashable objects.
```

AUTHORS:

- William Stein
- Nils Bruin (2006-12-19): rewrite to work for not-necessarily finite objects X.

prop(*f*)

random_sublist(*X*, *s*)

Return a pseudo-random sublist of the list X where the probability of including a particular element is s.

INPUT:

- X - list
- s - floating point number between 0 and 1

OUTPUT: list

EXAMPLES:

```
sage: S = [1,7,3,4,18]
sage: random_sublist(S, 0.5)
[1, 3, 4]
sage: random_sublist(S, 0.5)
[1, 3]
```

repr_lincomb(*symbols, coeffs, is_latex=False*)

Compute a string representation of a linear combination of some formal symbols.

INPUT:

- `symbols` - list of symbols
- `coeffs` - list of coefficients of the symbols

OUTPUT:

- `str` - a string

EXAMPLES:

```
sage: repr_lincomb(['a','b','c'], [1,2,3])
'a + 2*b + 3*c'
sage: repr_lincomb(['a','b','c'], [1,'2+3*x',3])
'a + (2+3*x)*b + 3*c'
sage: repr_lincomb(['a','b','c'], ['1+x^2','2+3*x',3])
'(1+x^2)*a + (2+3*x)*b + 3*c'
sage: repr_lincomb(['a','b','c'], ['1+x^2','-2+3*x',3])
'(1+x^2)*a + (-2+3*x)*b + 3*c'
sage: repr_lincomb(['a','b','c'], [1,-2,-3])
'a - 2*b - 3*c'
sage: t = PolynomialRing(RationalField(),'t').gen()
sage: repr_lincomb(['a', 's', ''], [-t,t-2,t**2+2])
'-t*a + (t-2)*s + (t^2+2)'
```

set_verbose(*level, files='all'*)

Set the global Sage verbosity level.

INPUT:

- `level` - an integer between 0 and 2, inclusive.
- **`files` (default: 'all'): list of files to make verbose, or** 'all' to make ALL files verbose (the default).

OUTPUT: changes the state of the verbosity flag and possibly appends to the list of files that are verbose.

EXAMPLES:

```
sage: set_verbose(2)
sage: verbose("This is Sage.", level=1)  # not tested
VERBOSE1 (?): This is Sage.
sage: verbose("This is Sage.", level=2)  # not tested
VERBOSE2 (?): This is Sage.
sage: verbose("This is Sage.", level=3)  # not tested
[no output]
sage: set_verbose(0)
```

set_verbose_files(*file_name*)

sourcefile(*object*)

Work out which source or compiled file an object was defined in.

srange(*start, end=None, step=1, universe=None, check=True, include_endpoint=False, endpoint_tolerance=1.0000000000000001e-05*)

Return list of numbers `a, a+step, ..., a+k*step`, where `a+k*step < b` and `a+(k+1)*step >= b` over exact rings, and makes a best attempt for inexact rings (see note below).

This provides one way to iterate over Sage integers as opposed to Python int's. It also allows you to specify step sizes for such an iteration. Note, however, that what is returned is a full list of Integers and not an iterator. It is potentially much slower than the Python range function, depending on the application. The function xsrange() provides an iterator with similar functionality which would usually be more efficient than using srange().

INPUT:

- `a` - number

- b - number (default: None)
- step - number (default: 1)
- universe - Parent or type where all the elements should live (default: deduce from inputs)
- check - make sure a, b, and step all lie in the same universe
- include_endpoint - whether or not to include the endpoint (default: False)
- endpoint_tolerance - used to determine whether or not the endpoint is hit for inexact rings (default 1e-5)

OUTPUT:

- list

If b is None, then b is set equal to a and a is set equal to the 0 in the parent of b.

Unlike range, a and b can be any type of numbers, and the resulting list involves numbers of that type.

Note: The list elements are computed via repeated addition rather than multiplication, which may produce slightly different results with inexact rings. For example:

```
sage: sum([1.1] * 10) == 1.1 * 10
False
```

Also, the question of whether the endpoint is hit exactly for a given `a + k*step` is fuzzy for an inexact ring. If `a + k*step = b` for some k within `endpoint_tolerance` of being integral, it is considered an exact hit, thus avoiding spurious values falling just below the endpoint.

Note: This function is called `srange` to distinguish it from the builtin Python `range` command. The s at the beginning of the name stands for "Sage".

EXAMPLES:

```
sage: v = srange(5); v
[0, 1, 2, 3, 4]
sage: type(v[2])
<type 'sage.rings.integer.Integer'>
sage: srange(1, 10)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: srange(10, 1, -1)
[10, 9, 8, 7, 6, 5, 4, 3, 2]
sage: srange(10,1,-1, include_endpoint=True)
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
sage: srange(1, 10, universe=RDF)
[1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0]

sage: srange(1, 10, 1/2)
[1, 3/2, 2, 5/2, 3, 7/2, 4, 9/2, 5, 11/2, 6, 13/2, 7, 15/2, 8, 17/2, 9, 19/2]
sage: srange(1, 5, 0.5)
[1.00000000000000, 1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000, 3.500
sage: srange(0, 1, 0.4)
[0.000000000000000, 0.400000000000000, 0.800000000000000]
sage: srange(1.0, 5.0, include_endpoint=True)
[1.00000000000000, 2.00000000000000, 3.00000000000000, 4.00000000000000, 5.00000000000000]
sage: srange(1.0, 1.1)
[1.00000000000000]
sage: srange(1.0, 1.0)
[]
sage: V = VectorSpace(QQ, 2)
sage: srange(V([0,0]), V([5,5]), step=V([2,2]))
[(0, 0), (2, 2), (4, 4)]
```

Including the endpoint:: sage: srange(0, 10, step=2, include_endpoint=True) [0, 2, 4, 6, 8, 10] sage: srange(0, 10, step=3, include_endpoint=True) [0, 3, 6, 9]

Try some inexact rings:

```
sage: srange(0.5, 1.1, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
sage: srange(0.5, 1, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.8, 0.9]
sage: srange(0.5, 0.9, 0.1, universe=RDF, include_endpoint=False)
[0.5, 0.6, 0.7, 0.8]
sage: srange(0, 1.1, 0.1, universe=RDF, include_endpoint=True)
[0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1]
sage: srange(0, 0.2, 0.1, universe=RDF, include_endpoint=True)
[0.0, 0.1, 0.2]
sage: srange(0, 0.3, 0.1, universe=RDF, include_endpoint=True)
[0.0, 0.1, 0.2, 0.3]
```

strunc(*s, n=60*)

Truncate at first space after position n, adding '...' if nontrivial truncation.

subsets(*X*)

Iterator over the *list* of all subsets of the iterable X, in no particular order. Each list appears exactly once, up to order.

INPUT:

- X - an iterable

OUTPUT: iterator of lists

EXAMPLES:

```
sage: list(powerset([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
sage: [z for z in powerset([0,[1,2]])]
[[], [0], [[1, 2]], [0, [1, 2]]]
```

Iterating over the power set of an infinite set is also allowed:

```
sage: i = 0
sage: for x in powerset(ZZ):
...     if i > 10:
...         break
...     else:
...         i += 1
...     print x,
[] [0] [1] [0, 1] [-1] [0, -1] [1, -1] [0, 1, -1] [2] [0, 2] [1, 2]
```

You may also use subsets as an alias for powerset:

```
sage: subsets([1,2,3])    # random object location in output
<generator object at 0xaeae418c>
sage: list(subsets([1,2,3]))
[[], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]

The reason we return lists instead of sets is that the elements of
sets must be hashable and many structures on which one wants the
powerset consist of non-hashable objects.
```

AUTHORS:

- William Stein
- Nils Bruin (2006-12-19): rewrite to work for not-necessarily finite objects X.

sxrange(*start, end=None, step=1, universe=None, check=True, include_endpoint=False, endpoint_tolerance=1.0000000000000001e-05*)

Return an iterator over numbers `a, a+step, ..., a+k*step`, where `a+k*step < b` and `a+(k+1)*step > b`.

INPUT: universe – Parent or type where all the elements should live (default: deduce from inputs) check – make sure a, b, and step all lie in the same universe include_endpoint – whether or not to include the endpoint (default: False) endpoint_tolerance – used to determine whether or not the endpoint is hit for inexact rings (default 1e-5)

- `a` - number
- `b` - number
- `step` - number (default: 1)

OUTPUT: iterator

Unlike range, a and b can be any type of numbers, and the resulting iterator involves numbers of that type.

See Also:

`srange()`

Note: This function is called `xsrange` to distinguish it from the builtin Python `xrange` command.

EXAMPLES:

```
sage: list(xsrange(1,10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: Q = RationalField()
sage: list(xsrange(1, 10, Q('1/2')))
[1, 3/2, 2, 5/2, 3, 7/2, 4, 9/2, 5, 11/2, 6, 13/2, 7, 15/2, 8, 17/2, 9, 19/2]
sage: list(xsrange(1, 5, 0.5))
[1.00000000000000, 1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000, 3.500
sage: list(xsrange(0, 1, 0.4))
[0.000000000000000, 0.400000000000000, 0.800000000000000]
```

Negative ranges are also allowed:

```
sage: list(xrange(4,1,-1))
[4, 3, 2]
sage: list(sxrange(4,1,-1))
[4, 3, 2]
sage: list(sxrange(4,1,-1/2))
[4, 7/2, 3, 5/2, 2, 3/2]
```

tmp_dir(*name='dir'*)

Create and return a temporary directory in `$HOME/.sage/temp/hostname/pid/`

tmp_filename(*name='tmp'*)

to_gmp_hex(*n*)

todo(*mesg=''*)

typecheck(*x, C, var='x'*)

Check that x is of instance C. If not raise a TypeError with an error message.

union(*x, y=None*)

Return the union of x and y, as a list. The resulting list need not be sorted and can change from call to call.

INPUT:

- x - iterable
- y - iterable (may optionally omitted)

OUTPUT: list

EXAMPLES:

```
sage: answer = union([1,2,3,4], [5,6]); answer
[1, 2, 3, 4, 5, 6]
sage: union([1,2,3,4,5,6], [5,6]) == answer
True
sage: union((1,2,3,4,5,6), [5,6]) == answer
True
sage: union((1,2,3,4,5,6), set([5,6])) == answer
True
```

uniq(*x*)

Return the sublist of all elements in the list x that is sorted and is such that the entries in the sublist are unique.

EXAMPLES:

```
sage: v = uniq([1,1,8,-5,3,-5,'a','x','a'])
sage: v             # potentially random ordering of output
['a', 'x', -5, 1, 3, 8]
sage: set(v) == set(['a', 'x', -5, 1, 3, 8])
True
```

unset_verbose_files(*file_name*)

verbose(*mesg='', t=0, level=1, caller_name=None*)

Print a message if the current verbosity is at least level.

INPUT:

- mesg - str, a message to print
- t - int, optional, if included, will also print cputime(t), - which is the time since time t. Thus t should have been obtained with t=cputime()
- level - int, (default: 1) the verbosity level of what we are printing
- caller_name - string (default: None), the name of the calling function; in most cases Python can deduce this, so it need not be provided.

OUTPUT: possibly prints a message to stdout; also returns cputime()

EXAMPLE:

```
sage: set_verbose(1)
sage: t = cputime()
sage: t = verbose("This is Sage.", t, level=1, caller_name="william")       # not tested
VERBOSE1 (william): This is Sage. (time = 0.0)
sage: set_verbose(0)
```

walltime(*t=0*)

Return the wall time in second, or with optional argument t, return the wall time since time t. "Wall time" means the time on a wall clock, i.e., the actual time.

INPUT:

- t - (optional) float, time in CPU seconds

OUTPUT:

- float - time in seconds

EXAMPLES:

```
sage: w = walltime()
sage: F = factor(2^199-1)
sage: walltime(w)    # somewhat random
0.8823847770690918
```

word_wrap(*s, ncols=85*)

xsrange(*start, end=None, step=1, universe=None, check=True, include_endpoint=False, endpoint_tolerance=1.0000000000000001e-05*)

Return an iterator over numbers a, a+step, ..., a+k*step, where a+k*step < b and a+(k+1)*step > b.

INPUT: universe – Parent or type where all the elements should live (default: deduce from inputs) check – make sure a, b, and step all lie in the same universe include_endpoint – whether or not to include the endpoint (default: False) endpoint_tolerance – used to determine whether or not the endpoint is hit for inexact rings (default 1e-5)

- a - number
- b - number
- step - number (default: 1)

OUTPUT: iterator

Unlike range, a and b can be any type of numbers, and the resulting iterator involves numbers of that type.

See Also:

srange()

Note: This function is called xsrange to distinguish it from the builtin Python xrange command.

EXAMPLES:

```
sage: list(xsrange(1,10))
[1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: Q = RationalField()
sage: list(xsrange(1, 10, Q('1/2')))
[1, 3/2, 2, 5/2, 3, 7/2, 4, 9/2, 5, 11/2, 6, 13/2, 7, 15/2, 8, 17/2, 9, 19/2]
sage: list(xsrange(1, 5, 0.5))
[1.00000000000000, 1.50000000000000, 2.00000000000000, 2.50000000000000, 3.00000000000000, 3.500
sage: list(xsrange(0, 1, 0.4))
[0.000000000000000, 0.400000000000000, 0.800000000000000]
```

Negative ranges are also allowed:

```
sage: list(xrange(4,1,-1))
[4, 3, 2]
sage: list(sxrange(4,1,-1))
[4, 3, 2]
sage: list(sxrange(4,1,-1/2))
[4, 7/2, 3, 5/2, 2, 3/2]
```

11.2 Sage package management commands

A Sage package has the extension .spkg. It is a tarball that is (usually) bzip2 compressed that contains arbitrary data and an spkg-install file. An Sage package typically has the following components:

- spkg-install - shell script that is run to install the package
- Sage.txt - file that describes how the package was made, who maintains it, etc.
- sage - directory with extra patched version of files that needed during the install

Use the `install_package` command to install a new package, and use `optional_packages` to list all optional packages available on the central Sage server. The `upgrade` command upgrades all *standard* packages - there is no auto-upgrade command for optional packages.

All package management can also be done via the Sage command line.

experimental_packages()

Return two lists. The first contains the installed and the second contains the not-installed experimental packages that are available from the Sage repository. You must have an internet connection.

OUTPUT:

•installed experimental packages (as a list)

•NOT installed experimental packages (as a list)

Use `install_package(package_name)` to install or re-install a given package.

See Also:

`install_package()`, `upgrade()`

install_all_optional_packages(*force=True, dry_run=False*)

Install all available optional spkg's in the official Sage spkg repository. Returns a list of all spkg's that *fail* to install.

INPUT: **force – bool (default: True); whether to force reinstall of** spkg's that are already installed.

dry_run – bool (default: False); if True, just list the packages that would be installed in order, but don't actually install them.

OUTPUT: list of strings

NOTE: This is designed mainly for testing purposes. This also doesn't do anything with respect to dependencies – the packages are installed in alphabetical order. Dependency issues will be dealt with in a future version.

AUTHOR: – William Stein (2008-12)

EXAMPLES: sage: sage.misc.package.install_all_optional_packages(dry_run=True) # optional - internet Installing ... []

install_package(*package=None, force=False*)

Install a package or return a list of all packages that have been installed into this Sage install.

You must have an internet connection. Also, you will have to restart Sage for the changes to take affect.

It is not needed to provide the version number.

INPUT:

•`package` - optional; if specified, install the given package. If not, list all installed packages.

IMPLEMENTATION: calls 'sage -f'.

See Also:

`optional_packages()`, `upgrade()`

is_package_installed(*package*)
Return true if a package starting with the given string is installed.

EXAMPLES:

```
sage: is_package_installed('sage')
True
```

optional_packages()
Return two lists. The first contains the installed and the second contains the not-installed optional packages that are available from the Sage repository. You must have an internet connection.

OUTPUT:

- installed optional packages (as a list)
- NOT installed optional packages (as a list)

Use `install_package(package_name)` to install or re-install a given package.

See Also:

`install_package()`, `upgrade()`

package_mesg(*package_name*)

standard_packages()
Return two lists. The first contains the installed and the second contains the not-installed standard packages that are available from the Sage repository. You must have an internet connection.

OUTPUT:

- installed standard packages (as a list)
- NOT installed standard packages (as a list)

Use `install_package(package_name)` to install or re-install a given package.

See Also:

`install_package()`, `upgrade()`

upgrade()
Download and build the latest version of Sage.

You must have an internet connection. Also, you will have to restart Sage for the changes to take affect.

This upgrades to the latest version of core packages (optional packages are not automatically upgraded).

This will not work on systems that don't have a C compiler.

See Also:

`install_package()`, `optional_packages()`

11.3 A tool for inspecting Python pickles

AUTHORS:

- Carl Witty (2009-03)

The explain_pickle function takes a pickle and produces Sage code that will evaluate to the contents of the pickle. Ideally, the combination of explain_pickle to produce Sage code and sage_eval to evaluate the code would be a 100% compatible implementation of cPickle's unpickler; this is almost the case now.

EXAMPLES:

```
sage: explain_pickle(dumps(12345))
pg_make_integer = unpickle_global('sage.rings.integer', 'make_integer')
pg_make_integer('c1p')
sage: explain_pickle(dumps(polygen(QQ)))
pg_Polynomial_rational_dense = unpickle_global('sage.rings.polynomial.polynomial_element_generic', 'P
pg_PolynomialRing = unpickle_global('sage.rings.polynomial.polynomial_ring_constructor', 'PolynomialR
pg_RationalField = unpickle_global('sage.rings.rational_field', 'RationalField')
pg = unpickle_instantiate(pg_RationalField, ())
pg_make_rational = unpickle_global('sage.rings.rational', 'make_rational')
pg_Polynomial_rational_dense(pg_PolynomialRing(pg, 'x', None, False), [pg_make_rational('0'), pg_make
sage: sage_eval(explain_pickle(dumps(polygen(QQ)))) == polygen(QQ)
True
```

By default (as above) the code produced contains calls to several utility functions (unpickle_global, etc.); this is done so that the code is truly equivalent to the pickle. If the pickle can be loaded into a future version of Sage, then the code that explain_pickle produces today should work in that future Sage as well.

It is also possible to produce simpler code, that is tied to the current version of Sage; here are the above two examples again:

```
sage: explain_pickle(dumps(12345), in_current_sage=True)
from sage.rings.integer import make_integer
make_integer('c1p')
sage: explain_pickle(dumps(polygen(QQ)), in_current_sage=True)
from sage.rings.polynomial.polynomial_element_generic import Polynomial_rational_dense
from sage.rings.rational import make_rational
Polynomial_rational_dense(PolynomialRing(RationalField(), 'x', None, False), [make_rational('0'), mak
```

The explain_pickle function has several use cases.

- Write pickling support for your classes

  You can use explain_pickle to see what will happen when a pickle is unpickled. Consider: is this sequence of commands something that can be easily supported in all future Sage versions, or does it expose internal design decisions that are subject to change?

- Debug old pickles

  If you have a pickle from an old version of Sage that no longer unpickles, you can use explain_pickle to see what it is trying to do, to figure out how to fix it.

- Use explain_pickle in doctests to help maintenance

  If you have a `loads(dumps(S))` doctest, you could also add an `explain_pickle(dumps(S))` doctest. Then if something changes in a way that would invalidate old pickles, the output of `explain_pickle` will also change. At that point, you can add the previous output of `explain_pickle` as a new set of doctests (and then update the :obj‘explain_pickle‘ doctest to use the new output), to ensure that old pickles will continue to work. (These problems will also be caught using the `picklejar`, but having the tests directly in the relevant module is clearer.)

As mentioned above, there are several output modes for `explain_pickle`, that control fidelity versus simplicity of the output. For example, the GLOBAL instruction takes a module name and a class name and produces the corresponding class. So GLOBAL of `sage.rings.integer`, `Integer` is approximately equivalent to `sage.rings.integer.Integer`.

However, this class lookup process can be customized (using sage.structure.sage_object.register_unpickle_override). For instance, if some future version of Sage renamed `sage/rings/integer.pyx` to

sage/rings/knuth_was_here.pyx, old pickles would no longer work unless register_unpickle_override was used; in that case, GLOBAL of 'sage.rings.integer', 'integer' would mean sage.rings.knuth_was_here.integer.

By default, explain_pickle will map this GLOBAL instruction to unpickle_global('sage.rings.integer', 'integer'). Then when this code is evaluated, unpickle_global will look up the current mapping in the register_unpickle_override table, so the generated code will continue to work even in hypothetical future versions of Sage where integer.pyx has been renamed.

If you pass the flag in_current_sage=True, then explain_pickle will generate code that may only work in the current version of Sage, not in future versions. In this case, it would generate:

```
from sage.rings.integer import integer
```

and if you ran explain_pickle in hypothetical future sage, it would generate:

```
from sage.rings.knuth_was_here import integer
```

but the current code wouldn't work in the future sage.

If you pass the flag default_assumptions=True, then explain_pickle will generate code that would work in the absence of any special unpickling information. That is, in either current Sage or hypothetical future Sage, it would generate:

```
from sage.rings.integer import integer
```

The intention is that default_assumptions output is prettier (more human-readable), but may not actually work; so it is only intended for human reading.

There are several functions used in the output of explain_pickle. Here I give a brief description of what they usually do, as well as how to modify their operation (for instance, if you're trying to get old pickles to work).

- unpickle_global(module, classname): unpickle_global('sage.foo.bar', 'baz') is usually equivalent to sage.foo.bar.baz, but this can be customized with register_unpickle_override.
- unpickle_newobj(klass, args): Usually equivalent to klass.__new__(klass, *args). If klass is a Python class, then you can define __new__() to control the result (this result actually need not be an instance of klass). (This doesn't work for Cython classes.)
- unpickle_build(obj, state): If obj has a __setstate__() method, then this is equivalent to obj.__setstate__(state). Otherwise uses state to set the attributes of obj. Customize by defining __setstate__().
- unpickle_instantiate(klass, args): Usually equivalent to klass(*args). Cannot be customized.
- unpickle_appends(lst, vals): Appends the values in vals to lst. If not isinstance(lst, list), can be customized by defining a append() method.

class EmptyNewstyleClass()
A featureless new-style class (inherits from object); used for testing explain_pickle.

class EmptyOldstyleClass()
A featureless old-style class (does not inherit from object); used for testing explain_pickle.

class PickleDict(*items*)
An object which can be used as the value of a PickleObject. The items is a list of key-value pairs, where the keys and values are SageInputExpressions. We use this to help construct dictionary literals, instead of always starting with an empty dictionary and assigning to it.

class PickleExplainer(*sib, in_current_sage=False, default_assumptions=False, pedantic=False*)

An interpreter for the pickle virtual machine, that executes symbolically and constructs SageInputExpressions instead of directly constructing values.

APPEND()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(['a'])
    0: \x80 PROTO      2
    2: ]    EMPTY_LIST
    3: q    BINPUT     1
    5: U    SHORT_BINSTRING 'a'
    8: a    APPEND
    9: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
['a']
result: ['a']
```

As shown above, we prefer to create a list literal. This is not possible if the list is recursive:

```
sage: v = []
sage: v.append(v)
sage: test_pickle(v)
    0: \x80 PROTO      2
    2: ]    EMPTY_LIST
    3: q    BINPUT     1
    5: h    BINGET     1
    7: a    APPEND
    8: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si = []
list.append(si, si)
si
result: [[...]]
```

APPENDS()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(['a', 'b'])
    0: \x80 PROTO      2
    2: ]    EMPTY_LIST
    3: q    BINPUT     1
    5: (    MARK
    6: U        SHORT_BINSTRING 'a'
    9: U        SHORT_BINSTRING 'b'
   12: e        APPENDS    (MARK at 5)
   13: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
['a', 'b']
result: ['a', 'b']
```

As shown above, we prefer to create a list literal. This is not possible if the list is recursive:

```
sage: v = []
sage: v.append(v)
sage: v.append(v)
```

```
sage: test_pickle(v)
    0: \x80 PROTO      2
    2: ]    EMPTY_LIST
    3: q    BINPUT     1
    5: (    MARK
    6: h        BINGET     1
    8: h        BINGET     1
   10: e        APPENDS    (MARK at 5)
   11: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si = []
list.extend(si, [si, si])
si
result: [[...], [...]]
```

BINFLOAT(*f*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(float(pi))
    0: \x80 PROTO      2
    2: G    BINFLOAT   3.1415926535897931
   11: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
float(RR(3.1415926535897931))
result: 3.1415926535897931
```

BINGET(*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST + BINPUT + 'x' + POP + BINGET + 'x' + '.')
    0: ]    EMPTY_LIST
    1: q    BINPUT     120
    3: 0    POP
    4: h    BINGET     120
    6: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

BININT(*n*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(dumps(100000r, compress=False))
    0: \x80 PROTO      2
    2: J    BININT     100000
    7: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
100000
result: 100000
```

BININT1(*n*)

TESTS:

```
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(dumps(100r, compress=False))
        0: \x80 PROTO      2
        2: K    BININT1    100
        4: .    STOP
    highest protocol among opcodes = 2
    explain_pickle in_current_sage=True/False:
    100
    result: 100
```

BININT2(*n*)

TESTS:

```
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(dumps(1000r, compress=False))
        0: \x80 PROTO      2
        2: M    BININT2    1000
        5: .    STOP
    highest protocol among opcodes = 2
    explain_pickle in_current_sage=True/False:
    1000
    result: 1000
```

BINPERSID()

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(INT + "0\n" + BINPERSID + '.', args=('Yo!',))
        0: I    INT        0
        3: Q    BINPERSID
        4: .    STOP
    highest protocol among opcodes = 1
    explain_pickle in_current_sage=True/False:
    unpickle_persistent(0)
    result: 'Yo!'
```

BINPUT(*n*)

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(EMPTY_LIST + BINPUT + 'x' + POP + BINGET + 'x')
        0: ]    EMPTY_LIST
        1: q    BINPUT     120
        3: 0    POP
        4: h    BINGET     120
        6: .    STOP
    highest protocol among opcodes = 1
    explain_pickle in_current_sage=True/False:
    []
    result: []
```

BINSTRING(*s*)

TESTS:

```
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle('T\5\0\0\0hello.')
        0: T    BINSTRING 'hello'
       10: .    STOP
```

```
    highest protocol among opcodes = 1
    explain_pickle in_current_sage=True/False:
    'hello'
    result: 'hello'
```

BINUNICODE(*s*)

TESTS:

```
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(u'hi\u1234\U00012345')
        0: \x80 PROTO      2
        2: X    BINUNICODE u'hi\u1234\U00012345'
       16: q    BINPUT     1
       18: .    STOP
    highest protocol among opcodes = 2
    explain_pickle in_current_sage=True/False:
    u'hi\u1234\U00012345'
    result: u'hi\u1234\U00012345'
```

BUILD()

TESTS:

```
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(TestBuild())
        0: \x80 PROTO      2
        2: c    GLOBAL     'sage.misc.explain_pickle TestBuild'
       38: q    BINPUT     1
       40: )    EMPTY_TUPLE
       41: \x81 NEWOBJ
       42: q    BINPUT     2
       44: }    EMPTY_DICT
       45: q    BINPUT     3
       47: U    SHORT_BINSTRING 'x'
       50: K    BININT1    3
       52: s    SETITEM
       53: }    EMPTY_DICT
       54: q    BINPUT     4
       56: U    SHORT_BINSTRING 'y'
       59: K    BININT1    4
       61: s    SETITEM
       62: \x86 TUPLE2
       63: b    BUILD
       64: .    STOP
    highest protocol among opcodes = 2
    explain_pickle in_current_sage=True:
    from sage.misc.explain_pickle import TestBuild
    si = unpickle_newobj(TestBuild, ())
    si.__dict__['x'] = 3
    si.y = 4
    si
    explain_pickle in_current_sage=False:
    pg_TestBuild = unpickle_global('sage.misc.explain_pickle', 'TestBuild')
    si = unpickle_newobj(pg_TestBuild, ())
    unpickle_build(si, ({'x':3}, {'y':4}))
    si
    result: TestBuild: x=3; y=4

    sage: test_pickle(TestBuildSetstate(), verbose_eval=True)
        0: \x80 PROTO      2
```

```
     2: c    GLOBAL     'sage.misc.explain_pickle TestBuildSetstate'
    46: q    BINPUT     1
    48: )    EMPTY_TUPLE
    49: \x81 NEWOBJ
    50: q    BINPUT     2
    52: }    EMPTY_DICT
    53: q    BINPUT     3
    55: U    SHORT_BINSTRING 'x'
    58: K    BININT1    3
    60: s    SETITEM
    61: }    EMPTY_DICT
    62: q    BINPUT     4
    64: U    SHORT_BINSTRING 'y'
    67: K    BININT1    4
    69: s    SETITEM
    70: \x86 TUPLE2
    71: b    BUILD
    72: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from sage.misc.explain_pickle import TestBuildSetstate
si = unpickle_newobj(TestBuildSetstate, ())
si.__setstate__(({'x':3}, {'y':4}))
si
explain_pickle in_current_sage=False:
pg_TestBuildSetstate = unpickle_global('sage.misc.explain_pickle', 'TestBuildSetstate')
si = unpickle_newobj(pg_TestBuildSetstate, ())
unpickle_build(si, ({'x':3}, {'y':4}))
si
evaluating explain_pickle in_current_sage=True:
setting state from ({'x': 3}, {'y': 4})
evaluating explain_pickle in_current_sage=False:
setting state from ({'x': 3}, {'y': 4})
loading pickle with cPickle:
setting state from ({'x': 3}, {'y': 4})
result: TestBuild: x=4; y=3
```

DICT()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(DICT, args=('mark', 'a', 1, 2, 'b'))
    0: (    MARK
    1: P        PERSID     '1'
    4: P        PERSID     '2'
    7: P        PERSID     '3'
   10: P        PERSID     '4'
   13: d        DICT       (MARK at 0)
   14: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
{unpickle_persistent('1'):unpickle_persistent('2'), unpickle_persistent('3'):unpickle_persis
result: {'a': 1, 2: 'b'}
```

DUP()

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(EMPTY_LIST + DUP + TUPLE2 + STOP)
        0: ]    EMPTY_LIST
        1: 2    DUP
        2: \x86 TUPLE2
        3: .    STOP
    highest protocol among opcodes = 2
    explain_pickle in_current_sage=True/False:
    si = []
    (si, si)
    result: ([], [])
```

EMPTY_DICT()

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(EMPTY_DICT)
        0: }    EMPTY_DICT
        1: .    STOP
    highest protocol among opcodes = 1
    explain_pickle in_current_sage=True/False:
    {}
    result: {}
```

EMPTY_LIST()

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(EMPTY_LIST)
        0: ]    EMPTY_LIST
        1: .    STOP
    highest protocol among opcodes = 1
    explain_pickle in_current_sage=True/False:
    []
    result: []
```

EMPTY_TUPLE()

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(EMPTY_TUPLE)
        0: )    EMPTY_TUPLE
        1: .    STOP
    highest protocol among opcodes = 1
    explain_pickle in_current_sage=True/False:
    ()
    result: ()
```

EXT1(*n*)

TESTS:

```
    sage: from copy_reg import *
    sage: from sage.misc.explain_pickle import *
    sage: add_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 42)
    sage: test_pickle(EmptyNewstyleClass())
        0: \x80 PROTO      2
```

```
    2: \x82 EXT1       42
    4: )    EMPTY_TUPLE
    5: \x81 NEWOBJ
    6: q    BINPUT     1
    8: }    EMPTY_DICT
    9: q    BINPUT     2
   11: b    BUILD
   12: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si = unpickle_newobj(unpickle_extension(42), ())
unpickle_build(si, {})
si
result: EmptyNewstyleClass
sage: remove_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 42)
```

EXT2 (*n*)

TESTS:

```
sage: from copy_reg import *
sage: from sage.misc.explain_pickle import *
sage: add_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 31415)
sage: test_pickle(EmptyNewstyleClass())
    0: \x80 PROTO      2
    2: \x83 EXT2       31415
    5: )    EMPTY_TUPLE
    6: \x81 NEWOBJ
    7: q    BINPUT     1
    9: }    EMPTY_DICT
   10: q    BINPUT     2
   12: b    BUILD
   13: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si = unpickle_newobj(unpickle_extension(31415), ())
unpickle_build(si, {})
si
result: EmptyNewstyleClass
sage: remove_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 31415)
```

EXT4 (*n*)

TESTS:

```
sage: from copy_reg import *
sage: from sage.misc.explain_pickle import *
sage: add_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 27182818)
sage: test_pickle(EmptyNewstyleClass())
    0: \x80 PROTO      2
    2: \x84 EXT4       27182818
    7: )    EMPTY_TUPLE
    8: \x81 NEWOBJ
    9: q    BINPUT     1
   11: }    EMPTY_DICT
   12: q    BINPUT     2
   14: b    BUILD
   15: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si = unpickle_newobj(unpickle_extension(27182818), ())
```

```
    unpickle_build(si, {})
    si
    result: EmptyNewstyleClass
    sage: remove_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 27182818)
```

FLOAT (*f*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(FLOAT + '2.71828\n')
    0: F    FLOAT      2.71828
    9: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
2.71828
result: 2.71828
```

GET (*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST + PUT + '1\n' + POP + GET + '1\n' + '.')
    0: ]    EMPTY_LIST
    1: p    PUT        1
    4: 0    POP
    5: g    GET        1
    8: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

GLOBAL (*name*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
```

We've used register_unpickle_override so that unpickle_global will map TestGlobalOldName to TestGlobalNewName.

```
sage: test_pickle(TestGlobalOldName())
    0: \x80 PROTO      2
    2: c    GLOBAL     'sage.misc.explain_pickle TestGlobalOldName'
   46: q    BINPUT     1
   48: )    EMPTY_TUPLE
   49: \x81 NEWOBJ
   50: q    BINPUT     2
   52: }    EMPTY_DICT
   53: q    BINPUT     3
   55: b    BUILD
   56: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from sage.misc.explain_pickle import TestGlobalNewName
unpickle_newobj(TestGlobalNewName, ())
explain_pickle in_current_sage=False:
pg_TestGlobalOldName = unpickle_global('sage.misc.explain_pickle', 'TestGlobalOldName')
si = unpickle_newobj(pg_TestGlobalOldName, ())
```

```
unpickle_build(si, {})
si
result: TestGlobalNewName
```

Note that default_assumptions blithely assumes that it should use the old name, giving code that doesn't actually work as desired:

```
sage: explain_pickle(dumps(TestGlobalOldName()), default_assumptions=True)
from sage.misc.explain_pickle import TestGlobalOldName
unpickle_newobj(TestGlobalOldName, ())
```

A class name need not be a valid identifier:

```
sage: sage.misc.explain_pickle.__dict__['funny$name'] = TestGlobalFunnyName # see comment at
sage: test_pickle((TestGlobalFunnyName(), TestGlobalFunnyName()))
    0: \x80 PROTO      2
    2: c    GLOBAL     'sage.misc.explain_pickle funny$name'
   39: q    BINPUT     1
   41: )    EMPTY_TUPLE
   42: \x81 NEWOBJ
   43: q    BINPUT     2
   45: }    EMPTY_DICT
   46: q    BINPUT     3
   48: b    BUILD
   49: h    BINGET     1
   51: )    EMPTY_TUPLE
   52: \x81 NEWOBJ
   53: q    BINPUT     4
   55: }    EMPTY_DICT
   56: q    BINPUT     5
   58: b    BUILD
   59: \x86 TUPLE2
   60: q    BINPUT     6
   62: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
si1 = unpickle_global('sage.misc.explain_pickle', 'funny$name')
si2 = unpickle_newobj(si1, ())
unpickle_build(si2, {})
si3 = unpickle_newobj(si1, ())
unpickle_build(si3, {})
(si2, si3)
result: (TestGlobalFunnyName, TestGlobalFunnyName)
```

INST(*name*)

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps(EmptyOldstyleClass(), protocol=0))
    0: (    MARK
    1: i        INST       'sage.misc.explain_pickle EmptyOldstyleClass' (MARK at 0)
   46: p    PUT        0
   49: (    MARK
   50: d        DICT       (MARK at 49)
   51: p    PUT        1
   54: b    BUILD
   55: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True:
```

```
    from new import instance
    from sage.misc.explain_pickle import EmptyOldstyleClass
    instance(EmptyOldstyleClass)
    explain_pickle in_current_sage=False:
    pg_EmptyOldstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyOldstyleClass')
    pg = unpickle_instantiate(pg_EmptyOldstyleClass, ())
    unpickle_build(pg, {})
    pg
    result: EmptyOldstyleClass
```

INT(*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(INT + "-12345\n")
    0: I    INT        -12345
    8: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
-12345
result: -12345
```

INT can also be used to record True and False:

```
sage: test_pickle(INT + "00\n")
    0: I    INT        False
    4: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
False
result: False
sage: test_pickle(INT + "01\n")
    0: I    INT        True
    4: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
True
result: True
```

LIST()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(MARK + NONE + NEWFALSE + LIST)
    0: (    MARK
    1: N        NONE
    2: \x89     NEWFALSE
    3: l        LIST       (MARK at 0)
    4: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
[None, False]
result: [None, False]
```

LONG(*n*)

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(LONG + "12345678909876543210123456789L\n")
    0: L    LONG       12345678909876543210123456789L
   32: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
12345678909876543210123456789
result: 12345678909876543210123456789L
```

LONG1(*n*)
TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(1L)
    0: \x80 PROTO      2
    2: \x8a LONG1      1L
    5: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
1L
result: 1L
```

LONG4(*n*)
TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(LONG4 + '\014\0\0\0' + 'hello, world')
    0: \x8b LONG4      310796053766044358915011638 80L
   17: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
310796053766044358915011638 80
result: 310796053766044358915011638 80L
```

LONG_BINGET(*n*)
TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST + LONG_BINPUT + 'Sage' + POP + LONG_BINGET + 'Sage')
    0: ]    EMPTY_LIST
    1: r    LONG_BINPUT 1701273939
    6: 0    POP
    7: j    LONG_BINGET 1701273939
   12: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

LONG_BINPUT(*n*)
TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_LIST + LONG_BINPUT + 'Sage' + POP + LONG_BINGET + 'Sage')
    0: ]    EMPTY_LIST
    1: r    LONG_BINPUT 1701273939
```

```
     6: 0    POP
     7: j    LONG_BINGET 1701273939
    12: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

MARK()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(MARK + TUPLE)
    0: (    MARK
    1: t        TUPLE      (MARK at 0)
    2: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
()
result: ()
```

NEWFALSE()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(NEWFALSE)
    0: \x89 NEWFALSE
    1: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
False
result: False
```

NEWOBJ()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EmptyNewstyleClass())
    0: \x80 PROTO      2
    2: c    GLOBAL     'sage.misc.explain_pickle EmptyNewstyleClass'
   47: q    BINPUT     1
   49: )    EMPTY_TUPLE
   50: \x81 NEWOBJ
   51: q    BINPUT     2
   53: }    EMPTY_DICT
   54: q    BINPUT     3
   56: b    BUILD
   57: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from sage.misc.explain_pickle import EmptyNewstyleClass
unpickle_newobj(EmptyNewstyleClass, ())
explain_pickle in_current_sage=False:
pg_EmptyNewstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyNewstyleClass')
si = unpickle_newobj(pg_EmptyNewstyleClass, ())
unpickle_build(si, {})
si
```

```
    result: EmptyNewstyleClass
```

NEWTRUE()

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(NEWTRUE)
        0: \x88 NEWTRUE
        1: .    STOP
    highest protocol among opcodes = 2
    explain_pickle in_current_sage=True/False:
    True
    result: True
```

NONE()

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(NONE)
        0: N    NONE
        1: .    STOP
    highest protocol among opcodes = 0
    explain_pickle in_current_sage=True/False:
    None
    result: None
```

OBJ()

TESTS:

```
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(EmptyOldstyleClass())
        0: \x80 PROTO      2
        2: (    MARK
        3: c        GLOBAL     'sage.misc.explain_pickle EmptyOldstyleClass'
       48: q        BINPUT     1
       50: o        OBJ        (MARK at 2)
       51: q    BINPUT     2
       53: }    EMPTY_DICT
       54: q    BINPUT     3
       56: b    BUILD
       57: .    STOP
    highest protocol among opcodes = 2
    explain_pickle in_current_sage=True:
    from new import instance
    from sage.misc.explain_pickle import EmptyOldstyleClass
    instance(EmptyOldstyleClass)
    explain_pickle in_current_sage=False:
    pg_EmptyOldstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyOldstyleClass')
    pg = unpickle_instantiate(pg_EmptyOldstyleClass, ())
    unpickle_build(pg, {})
    pg
    result: EmptyOldstyleClass
```

PERSID(*id*)

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
```

```
    sage: test_pickle(PERSID + "0\n" + '.', args=('Yo!',))
        0: P    PERSID     '0'
        3: .    STOP
    highest protocol among opcodes = 0
    explain_pickle in_current_sage=True/False:
    unpickle_persistent('0')
    result: 'Yo!'
```

POP()

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(INT + "0\n" + POP + INT + "42\n")
        0: I    INT        0
        3: 0    POP
        4: I    INT        42
        8: .    STOP
    highest protocol among opcodes = 0
    explain_pickle in_current_sage=True/False:
    42
    result: 42
```

POP_MARK()

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(MARK + NONE + NEWFALSE + POP_MARK + NEWTRUE)
        0: (    MARK
        1: N        NONE
        2: \x89     NEWFALSE
        3: 1        POP_MARK    (MARK at 0)
        4: \x88 NEWTRUE
        5: .    STOP
    highest protocol among opcodes = 2
    explain_pickle in_current_sage=True/False:
    True
    result: True
```

PROTO(*proto*)

TESTS:

```
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(0r)
        0: \x80 PROTO      2
        2: K    BININT1    0
        4: .    STOP
    highest protocol among opcodes = 2
    explain_pickle in_current_sage=True/False:
    0
    result: 0
```

PUT(*n*)

TESTS:

```
    sage: from pickle import *
    sage: from sage.misc.explain_pickle import *
    sage: test_pickle(EMPTY_LIST + PUT + '1\n' + POP + GET + '1\n' + '.')
        0: ]    EMPTY_LIST
```

```
    1: p    PUT        1
    4: 0    POP
    5: g    GET        1
    8: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
[]
result: []
```

REDUCE()

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps(EmptyNewstyleClass(), protocol=1))
    0: c    GLOBAL     'copy_reg _reconstructor'
   25: q    BINPUT     0
   27: (    MARK
   28: c        GLOBAL     'sage.misc.explain_pickle EmptyNewstyleClass'
   73: q        BINPUT     1
   75: c        GLOBAL     '__builtin__ object'
   95: q        BINPUT     2
   97: N        NONE
   98: t        TUPLE      (MARK at 27)
   99: q    BINPUT     3
  101: R    REDUCE
  102: q    BINPUT     4
  104: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True:
from copy_reg import _reconstructor
from sage.misc.explain_pickle import EmptyNewstyleClass
from __builtin__ import object
_reconstructor(EmptyNewstyleClass, object, None)
explain_pickle in_current_sage=False:
pg__reconstructor = unpickle_global('copy_reg', '_reconstructor')
pg_EmptyNewstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyNewstyleClass')
pg_object = unpickle_global('__builtin__', 'object')
pg__reconstructor(pg_EmptyNewstyleClass, pg_object, None)
result: EmptyNewstyleClass

sage: test_pickle(TestReduceGetinitargs(), verbose_eval=True)
Running __init__ for TestReduceGetinitargs
    0: \x80 PROTO      2
    2: (    MARK
    3: c        GLOBAL     'sage.misc.explain_pickle TestReduceGetinitargs'
   51: q        BINPUT     1
   53: o        OBJ        (MARK at 2)
   54: q    BINPUT     2
   56: }    EMPTY_DICT
   57: q    BINPUT     3
   59: b    BUILD
   60: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from sage.misc.explain_pickle import TestReduceGetinitargs
TestReduceGetinitargs()
explain_pickle in_current_sage=False:
```

```
pg_TestReduceGetinitargs = unpickle_global('sage.misc.explain_pickle', 'TestReduceGetinitarg
pg = unpickle_instantiate(pg_TestReduceGetinitargs, ())
unpickle_build(pg, {})
pg
evaluating explain_pickle in_current_sage=True:
Running __init__ for TestReduceGetinitargs
evaluating explain_pickle in_current_sage=False:
Running __init__ for TestReduceGetinitargs
loading pickle with cPickle:
Running __init__ for TestReduceGetinitargs
result: TestReduceGetinitargs

sage: test_pickle(TestReduceNoGetinitargs(), verbose_eval=True)
Running __init__ for TestReduceNoGetinitargs
    0: \x80 PROTO      2
    2: (    MARK
    3: c        GLOBAL     'sage.misc.explain_pickle TestReduceNoGetinitargs'
   53: q        BINPUT     1
   55: o        OBJ        (MARK at 2)
   56: q    BINPUT     2
   58: }    EMPTY_DICT
   59: q    BINPUT     3
   61: b    BUILD
   62: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
from new import instance
from sage.misc.explain_pickle import TestReduceNoGetinitargs
instance(TestReduceNoGetinitargs)
explain_pickle in_current_sage=False:
pg_TestReduceNoGetinitargs = unpickle_global('sage.misc.explain_pickle', 'TestReduceNoGetini
pg = unpickle_instantiate(pg_TestReduceNoGetinitargs, ())
unpickle_build(pg, {})
pg
evaluating explain_pickle in_current_sage=True:
evaluating explain_pickle in_current_sage=False:
loading pickle with cPickle:
result: TestReduceNoGetinitargs
```

SETITEM()

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps({'a': 'b'}))
    0: (    MARK
    1: d        DICT       (MARK at 0)
    2: p    PUT        0
    5: S    STRING     'a'
   10: p    PUT        1
   13: S    STRING     'b'
   18: p    PUT        2
   21: s    SETITEM
   22: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
{'a':'b'}
result: {'a': 'b'}
```

We see above that we output the result as a dictionary literal, when possible. This is impossible when a key or value is recursive. First we test recursive values:

```
sage: value_rec = dict()
sage: value_rec['circular'] = value_rec
sage: test_pickle(pickle.dumps(value_rec))
    0: (    MARK
    1: d        DICT       (MARK at 0)
    2: p    PUT        0
    5: S    STRING     'circular'
   17: p    PUT        1
   20: g    GET        0
   23: s    SETITEM
   24: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
si = {}
si['circular'] = si
si
result: {'circular': {...}}
```

Then we test recursive keys:

```
sage: key_rec = dict()
sage: key = EmptyNewstyleClass()
sage: key.circular = key_rec
sage: key_rec[key] = 'circular'
sage: test_pickle(pickle.dumps(key_rec))
    0: (    MARK
    1: d        DICT       (MARK at 0)
    2: p    PUT        0
    5: c    GLOBAL     'copy_reg _reconstructor'
   30: p    PUT        1
   33: (    MARK
   34: c        GLOBAL     'sage.misc.explain_pickle EmptyNewstyleClass'
   79: p        PUT        2
   82: c        GLOBAL     '__builtin__ object'
  102: p        PUT        3
  105: N        NONE
  106: t        TUPLE      (MARK at 33)
  107: p    PUT        4
  110: R    REDUCE
  111: p    PUT        5
  114: (    MARK
  115: d        DICT       (MARK at 114)
  116: p    PUT        6
  119: S    STRING     'circular'
  131: p    PUT        7
  134: g    GET        0
  137: s    SETITEM
  138: b    BUILD
  139: g    GET        7
  142: s    SETITEM
  143: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True:
si1 = {}
from copy_reg import _reconstructor
from sage.misc.explain_pickle import EmptyNewstyleClass
```

```
from __builtin__ import object
si2 = _reconstructor(EmptyNewstyleClass, object, None)
si2.__dict__['circular'] = si1
si1[si2] = 'circular'
si1
explain_pickle in_current_sage=False:
si1 = {}
pg__reconstructor = unpickle_global('copy_reg', '_reconstructor')
pg_EmptyNewstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyNewstyleClass')
pg_object = unpickle_global('__builtin__', 'object')
si2 = pg__reconstructor(pg_EmptyNewstyleClass, pg_object, None)
unpickle_build(si2, {'circular':si1})
si1[si2] = 'circular'
si1
result: {EmptyNewstyleClass: 'circular'}
```

SETITEMS()

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps({'a': 'b', 1r : 2r}, protocol=2))
    0: \x80 PROTO      2
    2: }    EMPTY_DICT
    3: q    BINPUT     0
    5: (    MARK
    6: U        SHORT_BINSTRING 'a'
    9: q        BINPUT     1
   11: U        SHORT_BINSTRING 'b'
   14: q        BINPUT     2
   16: K        BININT1    1
   18: K        BININT1    2
   20: u        SETITEMS   (MARK at 5)
   21: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
{'a':'b', 1:2}
result: {'a': 'b', 1: 2}
```

Similar to the tests for SETITEM, we test recursive keys and values:

```
sage: recdict = {}
sage: recdict['Circular value'] = recdict
sage: key = EmptyOldstyleClass()
sage: key.recdict = recdict
sage: recdict[key] = 'circular_key'
sage: test_pickle(pickle.dumps(recdict, protocol=2))
    0: \x80 PROTO      2
    2: }    EMPTY_DICT
    3: q    BINPUT     0
    5: (    MARK
    6: (        MARK
    7: c            GLOBAL     'sage.misc.explain_pickle EmptyOldstyleClass'
   52: q            BINPUT     1
   54: o            OBJ        (MARK at 6)
   55: q        BINPUT     2
   57: }        EMPTY_DICT
   58: q        BINPUT     3
   60: U        SHORT_BINSTRING 'recdict'
```

```
   69: q        BINPUT     4
   71: h        BINGET     0
   73: s        SETITEM
   74: b        BUILD
   75: U        SHORT_BINSTRING 'circular_key'
   89: q        BINPUT     5
   91: U        SHORT_BINSTRING 'Circular value'
  107: q        BINPUT     6
  109: h        BINGET     0
  111: u        SETITEMS   (MARK at 5)
  112: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True:
si1 = {}
from new import instance
from sage.misc.explain_pickle import EmptyOldstyleClass
si2 = instance(EmptyOldstyleClass)
si2.__dict__['recdict'] = si1
si1[si2] = 'circular_key'
si1['Circular value'] = si1
si1
explain_pickle in_current_sage=False:
si = {}
pg_EmptyOldstyleClass = unpickle_global('sage.misc.explain_pickle', 'EmptyOldstyleClass')
pg = unpickle_instantiate(pg_EmptyOldstyleClass, ())
unpickle_build(pg, {'recdict':si})
si[pg] = 'circular_key'
si['Circular value'] = si
si
result: {EmptyOldstyleClass: 'circular_key', 'Circular value': {...}}
```

SHORT_BINSTRING(*s*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(dumps('hello', compress=False))
    0: \x80 PROTO      2
    2: U    SHORT_BINSTRING 'hello'
    9: q    BINPUT     1
   11: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
'hello'
result: 'hello'
```

STOP()

TESTS:

```
sage: from pickle import *
sage: from sage.misc.explain_pickle import *
sage: test_pickle(EMPTY_TUPLE)
    0: )    EMPTY_TUPLE
    1: .    STOP
highest protocol among opcodes = 1
explain_pickle in_current_sage=True/False:
()
result: ()
```

STRING(*s*)

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle("S'Testing...'\n.")
    0: S    STRING     'Testing...'
   14: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
'Testing...'
result: 'Testing...'
```

TUPLE()

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps(('a',)))
    0: (    MARK
    1: S        STRING     'a'
    6: p        PUT        0
    9: t        TUPLE      (MARK at 0)
   10: p    PUT        1
   13: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
('a',)
result: ('a',)
```

We prefer to produce tuple literals, as above; but if the tuple is recursive, we need a more complicated construction. Note that the cPickle unpickler can't handle this case, so we're not bug-for-bug identical to cPickle here (see http://bugs.python.org/issue5794):

```
sage: v = ([],)
sage: v[0].append(v)
sage: test_pickle(pickle.dumps(v))
    0: (    MARK
    1: (        MARK
    2: l            LIST       (MARK at 1)
    3: p        PUT        0
    6: (        MARK
    7: g            GET        0
   10: t            TUPLE      (MARK at 6)
   11: p        PUT        1
   14: a        APPEND
   15: 0        POP
   16: 0        POP        (MARK at 0)
   17: g    GET        1
   20: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
si1 = []
si2 = (si1,)
list.append(si1, si2)
si2
result: ([(...)],) (cPickle raised an exception!)
```

TUPLE1()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(('a',))
    0: \x80 PROTO      2
    2: U    SHORT_BINSTRING 'a'
    5: \x85 TUPLE1
    6: q    BINPUT     1
    8: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
('a',)
result: ('a',)
```

TUPLE2()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(('a','b'))
    0: \x80 PROTO      2
    2: U    SHORT_BINSTRING 'a'
    5: U    SHORT_BINSTRING 'b'
    8: \x86 TUPLE2
    9: q    BINPUT     1
   11: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
('a', 'b')
result: ('a', 'b')
```

TUPLE3()

TESTS:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(('a','b','c'))
    0: \x80 PROTO      2
    2: U    SHORT_BINSTRING 'a'
    5: U    SHORT_BINSTRING 'b'
    8: U    SHORT_BINSTRING 'c'
   11: \x87 TUPLE3
   12: q    BINPUT     1
   14: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
('a', 'b', 'c')
result: ('a', 'b', 'c')
```

UNICODE(*s*)

TESTS:

```
sage: import pickle
sage: from sage.misc.explain_pickle import *
sage: test_pickle(pickle.dumps(u'hi\u1234\U00012345'))
    0: V    UNICODE    u'hi\u1234\U00012345'
   20: p    PUT        0
   23: .    STOP
highest protocol among opcodes = 0
explain_pickle in_current_sage=True/False:
u'hi\u1234\U00012345'
result: u'hi\u1234\U00012345'
```

check_value(*v*)

Check that the given value is either a SageInputExpression or a PickleObject. Used for internal sanity checking.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: pe.check_value(7)
...
AssertionError
sage: pe.check_value(sib(7))
```

is_mutable_pickle_object(*v*)

Test whether a PickleObject is mutable (has never been converted to a SageInputExpression).

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: v = PickleObject(1, sib(1))
sage: pe.is_mutable_pickle_object(v)
True
sage: sib(v)
{atomic:1}
sage: pe.is_mutable_pickle_object(v)
False
```

pop()

Pop a value from the virtual machine's stack, and return it.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: pe.push(sib(7))
sage: pe.pop()
{atomic:7}
```

pop_to_mark()

Pop all values down to the 'mark' from the virtual machine's stack, and return the values as a list.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: pe.push_mark()
sage: pe.push(sib(7))
sage: pe.push(sib('hello'))
sage: pe.pop_to_mark()
[{atomic:7}, {atomic:'hello'}]
```

push(*v*)

Push a value onto the virtual machine's stack.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: pe.push(sib(7))
sage: pe.stack[-1]
{atomic:7}
```

push_and_share(*v*)

Push a value onto the virtual machine's stack; also mark it as shared for sage_input if we are in pedantic mode.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: pe.push_and_share(sib(7))
sage: pe.stack[-1]
{atomic:7}
sage: pe.stack[-1]._sie_share
True
```

push_mark()

Push a 'mark' onto the virtual machine's stack.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: pe.push_mark()
sage: pe.stack[-1]
'mark'
sage: pe.stack[-1] is the_mark
True
```

run_pickle(*p*)

Given an (uncompressed) pickle as a string, run the pickle in this virtual machine. Once a STOP has been executed, return the result (a SageInputExpression representing code which, when evaluated, will give the value of the pickle).

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: sib(pe.run_pickle('T\5\0\0\0hello.'))
{atomic:'hello'}
```

share(*v*)

Mark a sage_input value as shared, if we are in pedantic mode.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: pe = PickleExplainer(sib, in_current_sage=True, default_assumptions=False, pedantic=Tr
sage: v = sib(7)
```

```
sage: v._sie_share
False
sage: pe.share(v)
{atomic:7}
sage: v._sie_share
True
```

class PickleInstance(*klass*)
An object which can be used as the value of a PickleObject. Unlike other possible values of a PickleObject, a PickleInstance doesn't represent an exact value; instead, it gives the class (type) of the object.

class PickleObject(*value, expression*)
Pickles have a stack-based virtual machine. The explain_pickle pickle interpreter mostly uses SageInputExpressions, from sage_input, as the stack values. However, sometimes we want some more information about the value on the stack, so that we can generate better (prettier, less confusing) code. In such cases, we push a PickleObject instead of a SageInputExpression. A PickleObject contains a value (which may be a standard Python value, or a PickleDict or PickleInstance), an expression (a SageInputExpression), and an "immutable" flag (which checks whether this object has been converted to a SageInputExpression; if it has, then we must not mutate the object, since the SageInputExpression would not reflect the changes).

class TestAppendList()
A subclass of list, with deliberately-broken append and extend methods. Used for testing explain_pickle.

append()
A deliberately broken append method.
EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: v = TestAppendList()
sage: v.append(7)
...
TypeError: append() takes exactly 1 argument (2 given)
```

We can still append by directly using the list method: sage: list.append(v, 7) sage: v [7]

extend()
A deliberately broken extend method.
EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: v = TestAppendList()
sage: v.extend([3,1,4,1,5,9])
...
TypeError: extend() takes exactly 1 argument (2 given)
```

We can still extend by directly using the list method: sage: list.extend(v, (3,1,4,1,5,9)) sage: v [3, 1, 4, 1, 5, 9]

class TestAppendNonlist()
A list-like class, carefully designed to test exact unpickling behavior. Used for testing explain_pickle.

class TestBuild()
A simple class with a __getstate__ but no __setstate__. Used for testing explain_pickle.

class TestBuildSetstate()
A simple class with a __getstate__ and a __setstate__. Used for testing explain_pickle.

class TestGlobalFunnyName()
A featureless new-style class which has a name that's not a legal Python identifier.
EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: globals()['funny$name'] = TestGlobalFunnyName # see comment at end of file
sage: TestGlobalFunnyName.__name__
'funny$name'
sage: globals()['funny$name'] is TestGlobalFunnyName
True
```

class TestGlobalNewName()
A featureless new-style class. When you try to unpickle an instance of TestGlobalOldName, it is redirected to create an instance of this class instead. Used for testing explain_pickle.

EXAMPLES: sage: from sage.misc.explain_pickle import * sage: loads(dumps(TestGlobalOldName())) TestGlobalNewName

class TestGlobalOldName()
A featureless new-style class. When you try to unpickle an instance of this class, it is redirected to create a TestGlobalNewName instead. Used for testing explain_pickle.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: loads(dumps(TestGlobalOldName()))
TestGlobalNewName
```

class TestReduceGetinitargs()
An old-style class with a __getinitargs__ method. Used for testing explain_pickle.

class TestReduceNoGetinitargs()
An old-style class with no __getinitargs__ method. Used for testing explain_pickle.

explain_pickle(*pickle=None, file=None, compress=True, **kwargs*)
Explain a pickle. That is, produce source code such that evaluating the code is equivalent to loading the pickle. Feeding the result of `explain_pickle` to `sage_eval` should be totally equivalent to loading the `pickle` with `cPickle`.

INPUTS:

- `pickle` – the pickle to explain, as a string (default: None)
- `file` – a filename of a pickle (default: None)
- **`compress` – if False, don't attempt to decompress the pickle** (default: True)
- **`in_current_sage` – if True, produce potentially simpler code that is** tied to the current version of Sage. (default: False)
- **`default_assumptions` – if True, produce potentially simpler code that** assumes that generic unpickling code will be used. This code may not actually work. (default: False)
- **`eval` – if True, then evaluate the resulting code and return the** evaluated result. (default: False)
- **`preparse` – if True, then produce code to be evaluated with** Sage's preparser; if False, then produce standard Python code; if None, then produce code that will work either with or without the preparser. (default: True)
- **`pedantic` – if True, then carefully ensures that the result has** at least as much sharing as the result of cPickle (it may have more, for immutable objects). (default: False)

Exactly one of `pickle` (a string containing a pickle) or `file` (the filename of a pickle) must be provided.

EXAMPLES:

```
sage: explain_pickle(dumps({('a', 'b'): [1r, 2r]}))
{('a', 'b'):[1r, 2r]}
sage: explain_pickle(dumps(RR(pi)), in_current_sage=True)
from sage.rings.real_mpfr import __create__RealNumber_version0
from sage.rings.real_mpfr import __create__RealField_version0
__create__RealNumber_version0(__create__RealField_version0(53r, False, 'RNDN'), '3.4gvml245kc0@0
sage: s = 'hi'
sage: explain_pickle(dumps((s, s)))
('hi', 'hi')
sage: explain_pickle(dumps((s, s)), pedantic=True)
si = 'hi'
(si, si)
sage: explain_pickle(dumps(5r))
5r
sage: explain_pickle(dumps(5r), preparse=False)
5
sage: explain_pickle(dumps(5r), preparse=None)
int(5)
sage: explain_pickle(dumps(22/7))
pg_make_rational = unpickle_global('sage.rings.rational', 'make_rational')
pg_make_rational('m/7')
sage: explain_pickle(dumps(22/7), in_current_sage=True)
from sage.rings.rational import make_rational
make_rational('m/7')
sage: explain_pickle(dumps(22/7), default_assumptions=True)
from sage.rings.rational import make_rational
make_rational('m/7')
```

explain_pickle_string(*pickle, in_current_sage=False, default_assumptions=False, eval=False, preparse=True, pedantic=False*)

This is a helper function for explain_pickle. It takes a decompressed pickle string as input; other than that, its options are all the same as explain_pickle.

EXAMPLES:

```
sage: sage.misc.explain_pickle.explain_pickle_string(dumps("Hello, world", compress=False))
'Hello, world'
```

(See the documentation for `explain_pickle` for many more examples.)

name_is_valid(*name*)

Test whether a string is a valid Python identifier. (We use a conservative test, that only allows ASCII identifiers.)

EXAMPLES:

```
sage: from sage.misc.explain_pickle import name_is_valid
sage: name_is_valid('fred')
True
sage: name_is_valid('Yes!ValidName')
False
sage: name_is_valid('_happy_1234')
True
```

test_pickle(*p, verbose_eval=False, pedantic=False, args=()*)

Tests explain_pickle on a given pickle p. p can be:

- a string containing an uncompressed pickle (which will always end with a '.')

- a string containing a pickle fragment (not ending with '.') test_pickle will synthesize a pickle that will push args onto the stack (using persistent IDs), run the pickle fragment, and then STOP (if the string 'mark' occurs in args, then a mark will be pushed)
- an arbitrary object; test_pickle will pickle the object

Once it has a pickle, test_pickle will print the pickle's disassembly, run explain_pickle with in_current_sage=True and False, print the results, evaluate the results, unpickle the object with cPickle, and compare all three results.

If verbose_eval is True, then test_pickle will print messages before evaluating the pickles; this is to allow for tests where the unpickling prints messages (to verify that the same operations occur in all cases).

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: test_pickle(['a'])
    0: \x80 PROTO      2
    2: ]    EMPTY_LIST
    3: q    BINPUT     1
    5: U    SHORT_BINSTRING 'a'
    8: a    APPEND
    9: .    STOP
highest protocol among opcodes = 2
explain_pickle in_current_sage=True/False:
['a']
result: ['a']
```

unpickle_appends(*lst, vals*)

Given a list (or list-like object) and a sequence of values, appends the values to the end of the list. This is careful to do so using the exact same technique that cPickle would use. Used by `explain_pickle`.

EXAMPLES:

```
sage: v = []
sage: unpickle_appends(v, (1, 2, 3))
sage: v
[1, 2, 3]
```

unpickle_build(*obj, state*)

Set the state of an object. Used by `explain_pickle`.

EXAMPLES:

```
sage: from sage.misc.explain_pickle import *
sage: v = EmptyNewstyleClass()
sage: unpickle_build(v, {'hello': 42})
sage: v.hello
42
```

unpickle_extension(*code*)

Takes an integer index and returns the extension object with that index. Used by `explain_pickle`.

EXAMPLES:

```
sage: from copy_reg import *
sage: add_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 42)
sage: unpickle_extension(42)
<class 'sage.misc.explain_pickle.EmptyNewstyleClass'>
sage: remove_extension('sage.misc.explain_pickle', 'EmptyNewstyleClass', 42)
```

unpickle_instantiate(*fn, args*)
Instantiate a new object of class fn with arguments args. Almost always equivalent to `fn(*args)`. Used by `explain_pickle`.

EXAMPLES:

```
sage: unpickle_instantiate(Integer, ('42',))
42
```

unpickle_newobj(*klass, args*)
Create a new object; this corresponds to the C code klass->tp_new(klass, args, NULL). Used by `explain_pickle`.

EXAMPLES: sage: unpickle_newobj(tuple, ([1, 2, 3],)) (1, 2, 3)

unpickle_persistent(*s*)
Takes an integer index and returns the persistent object with that index; works by calling whatever callable is stored in unpickle_persistent_loader. Used by `explain_pickle`.

EXAMPLES:

```
sage: import sage.misc.explain_pickle
sage: sage.misc.explain_pickle.unpickle_persistent_loader = lambda n: n+7
sage: unpickle_persistent(35)
42
```

11.4 Get resource usage of process

AUTHORS:

- William Stein (2006-03-04): initial versoin

VmB(*VmKey*)
Function used internally by this module.

get_memory_usage(*t=None*)
Return memory usage.

INPUT:

- `t` - None or output of previous call; (only used on Linux)

OUTPUT:

- `Linux` - Returns float number (in megabytes)
- `OS X` - returns string (VSIZE column of top)
- `other` - not implemented for any other operating systems

EXAMPLES:

We test that memory usage doesn't change instantly:

```
sage: t = get_memory_usage()
sage: get_memory_usage(t)          # amount of memory more than when we defined t.
0.0
```

linux_memory_usage()
Return memory usage in megabytes.

top()
Return the top output line that contains this running Sage process.

EXAMPLES: sage: top() # random output '72373 python 0.0% 0:01.36 1 14+ 1197 39M+ 34M+ 55M+ 130M+'

11.5 Multidimensional enumeration

AUTHORS:

- Joel B. Mohler (2006-10-12)
- William Stein (2006-07-19)
- Jon Hanke

cartesian_product_iterator(*X*)
Iterate over the Cartesian product.

INPUT:

•X - list or tuple of lists

OUTPUT: iterator over the cartesian product of the elements of X

EXAMPLES:

```
sage: list(cartesian_product_iterator([[1,2], ['a','b']]))
[(1, 'a'), (1, 'b'), (2, 'a'), (2, 'b')]
```

mrange(*sizes, typ=<type 'list'>*)
Return the multirange list with given sizes and type.

This is the list version of xmrange. Use xmrange for the iterator.

More precisely, return the iterator over all objects of type typ of n-tuples of Python ints with entries between 0 and the integers in the sizes list. The iterator is empty if sizes is empty or contains any non-positive integer.

INPUT:

•sizes - a list of nonnegative integers

•typ - (default: list) a type or class; more generally, something that can be called with a list as input.

OUTPUT: a list

EXAMPLES:

```
sage: mrange([3,2])
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
sage: mrange([3,2], tuple)
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
sage: mrange([3,2], sum)
[0, 1, 1, 2, 2, 3]
```

Examples that illustrate empty multi-ranges.

```
sage: mrange([])
[]
sage: mrange([5,3,-2])
[]
sage: mrange([5,3,0])
[]
```

AUTHORS:

- Jon Hanke
- William Stein

mrange_iter(*iter_list, typ=<type 'list'>*)

Return the multirange list derived from the given list of iterators.

This is the list version of xmrange_iter. Use xmrange_iter for the iterator.

More precisely, return the iterator over all objects of type typ of n-tuples of Python ints with entries between 0 and the integers in the sizes list. The iterator is empty if sizes is empty or contains any non-positive integer.

INPUT:

- `sizes` - a list of nonnegative integers
- `typ` - (default: list) a type or class; more generally, something that can be called with a list as input.

OUTPUT: a list

EXAMPLES:

```
sage: mrange_iter([range(3),[0,2]])
[[0, 0], [0, 2], [1, 0], [1, 2], [2, 0], [2, 2]]
sage: mrange_iter([['Monty','Flying'],['Python','Circus']], tuple)
[('Monty', 'Python'), ('Monty', 'Circus'), ('Flying', 'Python'), ('Flying', 'Circus')]
sage: mrange_iter([[2,3,5,7],[1,2]], sum)
[3, 4, 4, 5, 6, 7, 8, 9]
```

Examples that illustrate empty multi-ranges.

```
sage: mrange_iter([])
[]
sage: mrange_iter([range(5),xrange(3),xrange(-2)])
[]
sage: mrange_iter([range(5),range(3),range(0)])
[]
```

AUTHORS:

- Joel B. Mohler

class xmrange(*sizes, typ=<type 'list'>*)

Return the multirange iterate with given sizes and type.

More precisely, return the iterator over all objects of type typ of n-tuples of Python ints with entries between 0 and the integers in the sizes list. The iterator is empty if sizes is empty or contains any non-positive integer.

Use mrange for the non-iterator form.

INPUT:

- `sizes` - a list of nonnegative integers
- `typ` - (default: list) a type or class; more generally, something that can be called with a list as input.

OUTPUT: a generator

EXAMPLES: We create multi-range iterators, print them and also iterate through a tuple version.

```
sage: z = xmrange([3,2]);z
xmrange([3, 2])
sage: z = xmrange([3,2], tuple);z
xmrange([3, 2], <type 'tuple'>)
sage: for a in z:
...     print a
(0, 0)
(0, 1)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
```

We illustrate a few more iterations.

```
sage: list(xmrange([3,2]))
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
sage: list(xmrange([3,2], tuple))
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

Here we compute the sum of each element of the multi-range iterator:

```
sage: list(xmrange([3,2], sum))
[0, 1, 1, 2, 2, 3]
```

Next we compute the product:

```
sage: list(xmrange([3,2], prod))
[0, 0, 0, 1, 0, 2]
```

Examples that illustrate empty multi-ranges.

```
sage: list(xmrange([]))
[]
sage: list(xmrange([5,3,-2]))
[]
sage: list(xmrange([5,3,0]))
[]
```

We use a multi-range iterator to iterate through the cartesian product of sets.

```
sage: X = ['red', 'apple', 389]
sage: Y = ['orange', 'horse']
sage: for i,j in xmrange([len(X), len(Y)]):
...     print (X[i], Y[j])
('red', 'orange')
('red', 'horse')
('apple', 'orange')
('apple', 'horse')
(389, 'orange')
(389, 'horse')
```

AUTHORS:

- Jon Hanke

•William Stein

class xmrange_iter(*iter_list, typ=<type 'list'>*)

Return the multirange iterate derived from the given iterators and type.

Note: This basically gives you the cartesian product of sets.

More precisely, return the iterator over all objects of type typ of n-tuples of Python ints with entries between 0 and the integers in the sizes list. The iterator is empty if sizes is empty or contains any non-positive integer.

Use mrange_iter for the non-iterator form.

INPUT:

- `list_iter` - a list of objects usable as iterators (possibly lists)
- `typ` - (default: list) a type or class; more generally, something that can be called with a list as input.

OUTPUT: a generator

EXAMPLES: We create multi-range iterators, print them and also iterate through a tuple version.

```
sage: z = xmrange_iter([xrange(3),xrange(2)]);z
xmrange_iter([xrange(3), xrange(2)])
sage: z = xmrange_iter([range(3),range(2)], tuple);z
xmrange_iter([[0, 1, 2], [0, 1]], <type 'tuple'>)
sage: for a in z:
...     print a
(0, 0)
(0, 1)
(1, 0)
(1, 1)
(2, 0)
(2, 1)
```

We illustrate a few more iterations.

```
sage: list(xmrange_iter([range(3),range(2)]))
[[0, 0], [0, 1], [1, 0], [1, 1], [2, 0], [2, 1]]
sage: list(xmrange_iter([range(3),range(2)], tuple))
[(0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1)]
```

Here we compute the sum of each element of the multi-range iterator:

```
sage: list(xmrange_iter([range(3),range(2)], sum))
[0, 1, 1, 2, 2, 3]
```

Next we compute the product:

```
sage: list(xmrange_iter([range(3),range(2)], prod))
[0, 0, 0, 1, 0, 2]
```

Examples that illustrate empty multi-ranges.

```
sage: list(xmrange_iter([]))
[]
sage: list(xmrange_iter([xrange(5),xrange(3),xrange(-2)]))
[]
sage: list(xmrange_iter([xrange(5),xrange(3),xrange(0)]))
[]
```

We use a multi-range iterator to iterate through the cartesian product of sets.

```
sage: X = ['red', 'apple', 389]
sage: Y = ['orange', 'horse']
sage: for i,j in xmrange_iter([X, Y], tuple):
...     print (i, j)
('red', 'orange')
('red', 'horse')
('apple', 'orange')
('apple', 'horse')
(389, 'orange')
(389, 'horse')
```

AUTHORS:

- Joel B. Mohler

11.6 Installing shortcut scripts

install_scripts(*bin_directory=None*)

Run this command as `install_scripts(bin_directory)` to create scripts in the given bin directory that, independently of Sage, run various software components included with Sage: ['gap', 'gp', 'singular', 'maxima', 'M2', 'kash', 'mwrank', 'ipython', 'hg', 'hgmerge', 'R']

This command:

- verbosely tell you which scripts it adds, and
- will *not* overwrite any scripts you already have in the given bin directory.

INPUT:

- `bin_directory` - string; the directory into which to put the scripts

OUTPUT: Verbosely prints what it is doing and creates files in bin_directory that are world executable and readable.

Note: You may need to run Sage as root in order to run `install_scripts` successfully, since the user running Sage will need write permissions on `bin_directory`.

AUTHORS:

- William Stein: code / design
- Arthur Gaer: design

11.7 Sage Interface to the HG/Mercurial Revision Control System

These functions make setup and use of source control with Sage easier, using the distributed Mercurial HG source control system. To learn about Mercurial, see http://www.selenic.com/mercurial/wiki/ , in particular Understanding-Mercurial .

This system should all be fully usable from the Sage notebook (except for merging, currently). This system should all be mostly usable from the Sage notebook.

- Use `hg_sage.record()` to record all of your changes.
- Use `hg_sage.bundle('filename')` to bundle them up to send them.
- Use `hg_sage.inspect('filename.hg')` to inspect a bundle.

- Use `hg_sage.unbundle('filename.hg')` to import a bundle into your repository.
- Use `hg_sage.pull()` to synchronize with the latest official stable Sage changesets.

class HG(*dir, name, pull_url, push_url, target=None, cloneable=False, obj_name=''*)

add(*files, options=''*)

Add the given list of files (or file) or directories to your HG repository. They must exist already.

To see a list of files that haven't been added to the repository do self.status(). They will appear with an explanation point next them.

Add needs to be called whenever you add a new file or directory to your project. Of course, it also needs to be called when you first create the project, to let hg know which files should be kept track of.

INPUT:

- `files` - list or string; name of file or directory.
- `options` - string (e.g., '-dry-run')

apply(*bundle, update=True, options=''*)

Apply patches from a hg patch to the repository.

If the bundle is a .patch file, instead call the import_patch method. To see what is in a bundle before applying it, using self.incoming(bundle).

INPUT:

- `bundle` - an hg bundle (created with the bundle command)
- `update` - if True (the default), update the working directory after unbundling.

browse(*port=8200, address='localhost', open_viewer=True, options=''*)

Start a web server for this repository.

This server is very nice - you can browse all files in the repository, see their changelogs, see who wrote any given line, etc. Very nice.

INPUT:

- `port` - port that the server will listen on
- `address` - (default: 'localhost') address to listen on
- `open_viewer` - boolean (default: True); whether to pop up the web page
- `options` - a string passed directly to hg's serve command.

bundle(*filename, options='', url=None, base=None, to=None*)

Create an hg changeset bundle with the given filename against the repository at the given url (which is by default the 'official' Sage repository, unless push_url() is changed in a setup file).

If you have internet access, it's best to just do `hg_sage.bundle(filename)`. If you don't find a revision r that you and the person unbundling both have (by looking at `hg_sage.log()`), then do `hg_sage.bundle(filename, base=r)`.

Use self.inspect('file.bundle') to inspect the resulting bundle.

This is a file that you should probably send to William Stein (wstein@gmail.com), post to a web page, or send to sage-devel. It will be written to the current directory.

INPUT:

- `filename` - output file in which to put bundle
- `options` - pass to hg
- `url` - url to bundle against (default: SAGE_SERVER, or push_url())
- `base` - a base changeset revision number to bundle against (doesn't require internet access)

changes(*branches=None, keyword=None, limit=None, rev=None, merges=True, only_merges=False, patch=None, template=False, include=None, exclude=None, verbose=False*)

Display the change log for this repository. This is a list of changesets ordered by revision number.

By default this command outputs: changeset id and hash, tags, non-trivial parents, user, date and time, and a summary for each commit.

INPUT:

- `branches` - (string, default: None) show given branches
- `keyword` - (string, default: None) search for a keyword
- `limit` - (integer, default: None, or 20 in notebook mdoe) limit number of changes displayed
- `rev` - (integer) show the specified revision
- `merges` - (bool, default: True) whether or not to show merges
- `only_merges` - (bool, default: False) if true, show only merges
- `patch` - (string, default: None) show given patch
- `template` - (string, default: None) display with template
- `include` - (string, default: None) include names matching the given patterns
- `exclude` - (string, default: None) exclude names matching the given patterns
- `verbose` - (bool, default: False) If true, the list of changed files and full commit message is shown.

checkout (*options=''*)

update or merge working directory

Update the working directory to the specified revision.

If there are no outstanding changes in the working directory and there is a linear relationship between the current version and the requested version, the result is the requested version.

To merge the working directory with another revision, use the merge command.

By default, update will refuse to run if doing so would require merging or discarding local changes.

aliases: up, checkout, co

INPUT:

- `options` - string (default: ''):

```
-C --clean  overwrite locally modified files
-d --date   tipmost revision matching date
-r --rev    revision
```

ci (*files='', comment=None, options='', diff=True*)

Commit your changes to the repository.

Quit out of the editor without saving to not record your changes.

INPUT:

- `files` - space separated string of file names (optional) If specified only those files are commited. The path must be absolute or relative to self.dir().
- **`comment` - optional changeset comment. If you don't give** it you will be dumped into an editor. If you're using the Sage notebook, you *must* specify a comment.
- `options` - string:

```
-A --addremove  mark new/missing files as added/removed before committing
-m --message    use <text> as commit message
-l --logfile    read the commit message from <file>
-d --date       record datecode as commit date
-u --user       record user as commiter
-I --include    include names matching the given patterns
-X --exclude    exclude names matching the given patterns
```

- `diff` - (default: True) if True show diffs between your repository and your working repository before recording changes.

Note: If you create new files you should first add them with the add method.

clone (*name, rev=None*)

Clone the current branch of the Sage library, and make it active.

Only available for the `hg_sage` repository.

Use `hg_sage.switch('branch_name')` to switch to a different branch. You must restart Sage after switching.

INPUT:

- `name` - string
- `rev` - integer or None (default)

If rev is None, clones the latest recorded version of the repository. This is very fast, e.g., about 30-60 seconds (including any build). If a specific revision is specified, cloning may take much longer (e.g., 5 minutes), since all Pyrex code has to be regenerated and compiled.

EXAMPLES:

Make a clone of the repository called testing. A copy of the current repository will be created in a directory sage-testing, then SAGE_ROOT/devel/sage will point to sage-testing, and when you next restart Sage that's the version you'll be using.

```
sage: hg_sage.clone('testing')     # not tested
...
```

Make a clone of the repository as it was at revision 1328.

```
sage: hg_sage.clone('testing', 1328)     # not tested
...
```

co (*options=''*)

update or merge working directory

Update the working directory to the specified revision.

If there are no outstanding changes in the working directory and there is a linear relationship between the current version and the requested version, the result is the requested version.

To merge the working directory with another revision, use the merge command.

By default, update will refuse to run if doing so would require merging or discarding local changes.

aliases: up, checkout, co

INPUT:

- `options` - string (default: ''):

```
-C --clean   overwrite locally modified files
-d --date    tipmost revision matching date
-r --rev     revision
```

commit (*files='', comment=None, options='', diff=True*)

Commit your changes to the repository.

Quit out of the editor without saving to not record your changes.

INPUT:

- `files` - space separated string of file names (optional) If specified only those files are commited. The path must be absolute or relative to self.dir().
- **`comment` - optional changeset comment. If you don't give** it you will be dumped into an editor. If you're using the Sage notebook, you *must* specify a comment.
- `options` - string:

```
-A --addremove  mark new/missing files as added/removed before committing
-m --message    use <text> as commit message
-l --logfile    read the commit message from <file>
-d --date       record datecode as commit date
-u --user       record user as commiter
-I --include    include names matching the given patterns
-X --exclude    exclude names matching the given patterns
```

- `diff` - (default: True) if True show diffs between your repository and your working repository before recording changes.

Note: If you create new files you should first add them with the add method.

current_branch (*print_flag=True*)

Lists the current branch.

diff(*files=”, rev=None*)

Show differences between revisions for the specified files as a unified diff.

By default this command tells you exactly what you have changed in your working repository since you last commited changes.

INPUT:

- `files` - space separated list of files (relative to self.dir())
- `rev` - None or a list of integers.

Differences between files are shown using the unified diff format.

When two revision arguments are given, then changes are shown between those revisions. If only one revision is specified then that revision is compared to the working directory, and, when no revisions are specified, the working directory files are compared to its parent.

dir()

Return the directory where this repository is located.

export(*revs, filename=None, text=False, options=”*)

Export patches with the changeset header and diffs for one or more revisions.

If multiple revisions are given, one plain text unified diff file is generated for each one. These files should be applied using import_patch in order from smallest to largest revision number. The information shown in the changeset header is: author, changeset hash, parent and commit comment.

Note: If you are sending a patch to somebody using export and it depends on previous patches, make sure to include those revisions too! Alternatively, use the `bundle()` method, which includes enough information to patch against the default repository (but is an annoying and mysterious binary file).

INPUT:

- `revs` - integer or list of integers (revision numbers); use the log() method to see these numbers.
- `filename` - (default: ‘%R.patch’) The name of the file is given using a format string. The formatting rules are as follows:

```
%%   literal "%" character
%H   changeset hash (40 bytes of hexadecimal)
%N   number of patches being generated
%R   changeset revision number
%b   basename of the exporting repository
%h   short-form changeset hash (12 bytes of hexadecimal)
%n   zero-padded sequence number, starting at 1
```

- `options` - string (default: ‘’)
  - **–’**`-a -text’` - treat all files as text
  - **–’**`-switch-parent’` - diff against the second parent

  Without the `-a` option, export will avoid generating diffs of files it detects as binary. With `-a`, export will generate a diff anyway, probably with undesirable results.
  With the `-switch-parent` option, the diff will be against the second parent. It can be useful to review a merge.

head(*options=”*)

show current repository heads

Show all repository head changesets.

Repository “heads” are changesets that don’t have children changesets. They are where development generally takes place and are the usual targets for update and merge operations.

INPUT:

- `options` - string (default: ‘’):

```
-r --rev       show only heads which are descendants of rev
   --style     display using template map file
   --template  display with template
```

heads(*options=''*)

show current repository heads

Show all repository head changesets.

Repository "heads" are changesets that don't have children changesets. They are where development generally takes place and are the usual targets for update and merge operations.

INPUT:

•`options` - string (default: ''):

```
-r --rev       show only heads which are descendants of rev
   --style     display using template map file
   --template  display with template
```

help(*cmd=''*)

Return help about the given command, or if cmd is omitted a list of commands.

If this hg object is called hg_sage, then you call a command using `hg_sage('usual hg command line notation')`

history(*branches=None, keyword=None, limit=None, rev=None, merges=True, only_merges=False, patch=None, template=False, include=None, exclude=None, verbose=False*)

Display the change log for this repository. This is a list of changesets ordered by revision number.

By default this command outputs: changeset id and hash, tags, non-trivial parents, user, date and time, and a summary for each commit.

INPUT:

•`branches` - (string, default: None) show given branches

•`keyword` - (string, default: None) search for a keyword

•`limit` - (integer, default: None, or 20 in notebook mdoe) limit number of changes displayed

•`rev` - (integer) show the specified revision

•`merges` - (bool, default: True) whether or not to show merges

•`only_merges` - (bool, default: False) if true, show only merges

•`patch` - (string, default: None) show given patch

•`template` - (string, default: None) display with template

•`include` - (string, default: None) include names matching the given patterns

•`exclude` - (string, default: None) exclude names matching the given patterns

•`verbose` - (bool, default: False) If true, the list of changed files and full commit message is shown.

import_patch(*filename, options=''*)

Import an ordered set of patches from patch file, i.e., a plain text file created using the export command.

If there are outstanding changes in the working directory, import will abort unless given the -f flag.

If imported patch was generated by the export command, user and description from patch override values from message headers and body. Values given as options with -m and -u override these.

INPUT:

•`filename` - string

•`options` - string (default: ''):

```
options: [-p NUM] [-b BASE] [-m MESSage] [-f] PATCH...
  -p --strip         directory strip option for patch. This has the same meaning as the
  -m --message       use text as commit message
  -b --base          base path
  -f --force         skip check for outstanding uncommitted changes
```

ALIASES: patch

incoming(*source, options='-p'*)

Show new changesets found in the given source and display the corresponding diffs. This even works if the

source is a bundle file (ends in .hg or .bundle). This is great because it lets you "see inside" the myserious binary-only .hg files.

Show new changesets found in the specified path/URL or the default pull location. These are the changesets that would be pulled if a pull was requested.

For remote repository, using -bundle avoids downloading the changesets twice if the incoming is followed by a pull.

See pull for valid source format details.

ALIAS: inspect

INPUT:

- `filename` - string
- `options` - (default: '-p'):

```
string '[-p] [-n] [-M] [-r REV] ...'
  -M --no-merges       do not show merges
  -f --force           run even when remote repository is unrelated
  --style              display using template map file
  -n --newest-first    show newest record first
  --bundle             file to store the bundles into
  -p --patch           show patch
  -r --rev             a specific revision you would like to pull
  --template           display with template
  -e --ssh             specify ssh command to use
  --remotecmd          specify hg command to run on the remote side
```

inspect(*source, options='-p'*)

Show new changesets found in the given source and display the corresponding diffs. This even works if the source is a bundle file (ends in .hg or .bundle). This is great because it lets you "see inside" the myserious binary-only .hg files.

Show new changesets found in the specified path/URL or the default pull location. These are the changesets that would be pulled if a pull was requested.

For remote repository, using -bundle avoids downloading the changesets twice if the incoming is followed by a pull.

See pull for valid source format details.

ALIAS: inspect

INPUT:

- `filename` - string
- `options` - (default: '-p'):

```
string '[-p] [-n] [-M] [-r REV] ...'
  -M --no-merges       do not show merges
  -f --force           run even when remote repository is unrelated
  --style              display using template map file
  -n --newest-first    show newest record first
  --bundle             file to store the bundles into
  -p --patch           show patch
  -r --rev             a specific revision you would like to pull
  --template           display with template
  -e --ssh             specify ssh command to use
  --remotecmd          specify hg command to run on the remote side
```

list_branches(*print_flag=True*)

Print all branches in the current Sage installation.

log(*branches=None, keyword=None, limit=None, rev=None, merges=True, only_merges=False, patch=None, template=False, include=None, exclude=None, verbose=False*)

Display the change log for this repository. This is a list of changesets ordered by revision number.

By default this command outputs: changeset id and hash, tags, non-trivial parents, user, date and time, and a summary for each commit.

INPUT:

- `branches` - (string, default: None) show given branches
- `keyword` - (string, default: None) search for a keyword
- `limit` - (integer, default: None, or 20 in notebook mdoe) limit number of changes displayed
- `rev` - (integer) show the specified revision
- `merges` - (bool, default: True) whether or not to show merges
- `only_merges` - (bool, default: False) if true, show only merges
- `patch` - (string, default: None) show given patch
- `template` - (string, default: None) display with template
- `include` - (string, default: None) include names matching the given patterns
- `exclude` - (string, default: None) exclude names matching the given patterns
- `verbose` - (bool, default: False) If true, the list of changed files and full commit message is shown.

merge(*options=''*)

Merge working directory with another revision

Merge the contents of the current working directory and the requested revision. Files that changed between either parent are marked as changed for the next commit and a commit must be performed before any further updates are allowed.

INPUT:

- `options` - default: '':

```
-f --force  force a merge with outstanding changes
-r --rev    revision to merge
```

move(*src, dest, options=''*)

Move (rename) the given file, from src to dest. This command takes effect in the next commit.

INPUT:

- `src, dest` - strings that define a file, relative to self.dir()
- `options`:

```
-A --after    record a rename that has already occurred
-f --force    forcibly copy over an existing managed file
-n --dry-run  do not perform actions, just print output
```

mv(*src, dest, options=''*)

Move (rename) the given file, from src to dest. This command takes effect in the next commit.

INPUT:

- `src, dest` - strings that define a file, relative to self.dir()
- `options`:

```
-A --after    record a rename that has already occurred
-f --force    forcibly copy over an existing managed file
-n --dry-run  do not perform actions, just print output
```

outgoing(*url=None, opts=''*)

Use this to find changsets that are in your branch, but not in the specified destination repository. If no destination is specified, the official repository is used. By default, push_url() is used.

From the Mercurial documentation:

> Show changesets not found in the specified destination repository or the default push location.
> These are the changesets that would be pushed if a push was requested.
> See push() for valid destination format details.

INPUT:

- `url` - (Default: self.push_url()) the official repository
  - –http://[user@]host[:port]/[path]
  - –https://[user@]host[:port]/[path]
  - –ssh://[user@]host[:port]/[path]
  - –local directory (starting with a /)
  - –name of a branch (for hg_sage); no /'s
- `options` - (Default: None):

```
-M --no-merges     do not show merges
-f --force         run even when remote repository is unrelated
-p --patch         show patch
--style            display using template map file
-r --rev           a specific revision you would like to push
-n --newest-first  show newest record first
--template         display with template
-e --ssh           specify ssh command to use
--remotecmd        specify hg command to run on the remote side
```

patch (*filename, options=''*)

Import an ordered set of patches from patch file, i.e., a plain text file created using the export command.

If there are outstanding changes in the working directory, import will abort unless given the -f flag.

If imported patch was generated by the export command, user and description from patch override values from message headers and body. Values given as options with -m and -u override these.

INPUT:

- `filename` - string
- `options` - string (default: ''):

```
options: [-p NUM] [-b BASE] [-m MESSage] [-f] PATCH...
  -p --strip         directory strip option for patch. This has the same meaning as the
  -m --message       use text as commit message
  -b --base          base path
  -f --force         skip check for outstanding uncommitted changes
```

ALIASES: patch

pull (*url=None, options='-u'*)

Pull all new patches from the repository at the given url, or use the default 'official' repository if no url is specified.

INPUT:

- `url` - (Default: self.push_url()) the official repository
  - –http://[user@]host[:port]/[path]
  - –https://[user@]host[:port]/[path]
  - –ssh://[user@]host[:port]/[path]
  - –local directory (starting with a /)
  - –name of a branch (for hg_sage); no /'s
- `options` - (Default: '-u'):

```
-u --update     update the working directory to tip after pull
-e --ssh        specify ssh command to use
-f --force      run even when remote repository is unrelated
-r --rev        a specific revision you would like to pull
--remotecmd     specify hg command to run on the remote side
```

Some notes about using SSH with Mercurial:

- SSH requires an accessible shell account on the destination machine and a copy of hg in the remote path or specified with as remotecmd.
- path is relative to the remote user's home directory by default. Use an extra slash at the start of a path to specify an absolute path: ssh://example.com//tmp/repository
- Mercurial doesn't use its own compression via SSH; the right thing to do is to configure it in your /.ssh/ssh_config, e.g.:

```
Host *.mylocalnetwork.example.com
  Compression off
Host *
  Compression on
```

Alternatively specify 'ssh -C' as your ssh command in your hgrc or with the -ssh command line option.

pull_url()
Return the default 'master url' for this repository.

push(*url=None, options=''*)
Push all new patches from the repository to the given destination.

INPUT:

- `url` - (Default: self.push_url()) the official repository
  - http://[user@]host[:port]/[path]
  - https://[user@]host[:port]/[path]
  - ssh://[user@]host[:port]/[path]
  - local directory (starting with a /)
  - name of a branch (for hg_sage); no /'s
- `options` - (Default: '-u'):

```
-e --ssh        specify ssh command to use
-f --force      run even when remote repository is unrelated
-r --rev        a specific revision you would like to pull
--remotecmd     specify hg command to run on the remote side
```

Some notes about using SSH with Mercurial:

- SSH requires an accessible shell account on the destination machine and a copy of hg in the remote path or specified with as remotecmd.
- path is relative to the remote user's home directory by default. Use an extra slash at the start of a path to specify an absolute path: ssh://example.com//tmp/repository
- Mercurial doesn't use its own compression via SSH; the right thing to do is to configure it in your /.ssh/ssh_config, e.g.:

```
Host *.mylocalnetwork.example.com
  Compression off
Host *
  Compression on
```

Alternatively specify 'ssh -C' as your ssh command in your hgrc or with the -ssh command line option.

push_url()
Return the default url for uploading this repository.

record(*files='', comment=None, options='', diff=True*)
Commit your changes to the repository.

Quit out of the editor without saving to not record your changes.

INPUT:

- `files` - space separated string of file names (optional) If specified only those files are commited. The path must be absolute or relative to self.dir().

- **`comment` - optional changeset comment. If you don't give** it you will be dumped into an editor. If you're using the Sage notebook, you *must* specify a comment.
- `options` - string:

```
-A --addremove  mark new/missing files as added/removed before committing
-m --message    use <text> as commit message
-l --logfile    read the commit message from <file>
-d --date       record datecode as commit date
-u --user       record user as commiter
-I --include    include names matching the given patterns
-X --exclude    exclude names matching the given patterns
```

- `diff` - (default: True) if True show diffs between your repository and your working repository before recording changes.

Note: If you create new files you should first add them with the add method.

`remove` (*files, options=''*)

Remove the given list of files (or file) or directories from your HG repository.

INPUT:

- `files` - list or string; name of file or directory.
- `options` - string (e.g., '-f')

`rename` (*src, dest, options=''*)

Move (rename) the given file, from src to dest. This command takes effect in the next commit.

INPUT:

- `src, dest` - strings that define a file, relative to self.dir()
- `options`:

```
-A --after    record a rename that has already occurred
-f --force    forcibly copy over an existing managed file
-n --dry-run  do not perform actions, just print output
```

`revert` (*files='', options='', rev=None*)

Revert files or dirs to their states as of some revision

With no revision specified, revert the named files or directories to the contents they had in the parent of the working directory. This restores the contents of the affected files to an unmodified state. If the working directory has two parents, you must explicitly specify the revision to revert to.

Modified files are saved with a .orig suffix before reverting. To disable these backups, use -no-backup.

Using the -r option, revert the given files or directories to their contents as of a specific revision. This can be helpful to 'roll back' some or all of a change that should not have been committed.

Revert modifies the working directory. It does not commit any changes, or change the parent of the working directory. If you revert to a revision other than the parent of the working directory, the reverted files will thus appear modified afterwards.

If a file has been deleted, it is recreated. If the executable mode of a file was changed, it is reset.

If names are given, all files matching the names are reverted.

If no arguments are given, all files in the repository are reverted.

OPTIONS:

```
   --no-backup  do not save backup copies of files
-I --include    include names matching given patterns
-X --exclude    exclude names matching given patterns
-n --dry-run    do not perform actions, just print output
```

`rm` (*files, options=''*)

Remove the given list of files (or file) or directories from your HG repository.

INPUT:

- `files` - list or string; name of file or directory.
- `options` - string (e.g., '-f')

rollback()
Remove recorded patches without changing the working copy.

save(*filename, options='', url=None, base=None, to=None*)
Create an hg changeset bundle with the given filename against the repository at the given url (which is by default the 'official' Sage repository, unless push_url() is changed in a setup file).

If you have internet access, it's best to just do `hg_sage.bundle(filename)`. If you don't find a revision r that you and the person unbundling both have (by looking at `hg_sage.log()`), then do `hg_sage.bundle(filename, base=r)`.

Use self.inspect('file.bundle') to inspect the resulting bundle.

This is a file that you should probably send to William Stein (wstein@gmail.com), post to a web page, or send to sage-devel. It will be written to the current directory.

INPUT:

- `filename` - output file in which to put bundle
- `options` - pass to hg
- `url` - url to bundle against (default: SAGE_SERVER, or push_url())
- `base` - a base changeset revision number to bundle against (doesn't require internet access)

send(*filename, options='', url=None, base=None, to=None*)
Create an hg changeset bundle with the given filename against the repository at the given url (which is by default the 'official' Sage repository, unless push_url() is changed in a setup file).

If you have internet access, it's best to just do `hg_sage.bundle(filename)`. If you don't find a revision r that you and the person unbundling both have (by looking at `hg_sage.log()`), then do `hg_sage.bundle(filename, base=r)`.

Use self.inspect('file.bundle') to inspect the resulting bundle.

This is a file that you should probably send to William Stein (wstein@gmail.com), post to a web page, or send to sage-devel. It will be written to the current directory.

INPUT:

- `filename` - output file in which to put bundle
- `options` - pass to hg
- `url` - url to bundle against (default: SAGE_SERVER, or push_url())
- `base` - a base changeset revision number to bundle against (doesn't require internet access)

serve(*port=8200, address='localhost', open_viewer=True, options=''*)
Start a web server for this repository.

This server is very nice - you can browse all files in the repository, see their changelogs, see who wrote any given line, etc. Very nice.

INPUT:

- `port` - port that the server will listen on
- `address` - (default: 'localhost') address to listen on
- `open_viewer` - boolean (default: True); whether to pop up the web page
- `options` - a string passed directly to hg's serve command.

status()

switch(*name=None*)
Switch to a different branch. You must restart Sage after switching.

Only available for `hg_sage.`

INPUT:

- `name` - name of a Sage branch (default: None)

If the name is not given, this function returns a list of all branches.

unbundle(*bundle, update=True, options=''*)

Apply patches from a hg patch to the repository.

If the bundle is a .patch file, instead call the import_patch method. To see what is in a bundle before applying it, using self.incoming(bundle).

INPUT:

- `bundle` - an hg bundle (created with the bundle command)
- `update` - if True (the default), update the working directory after unbundling.

up(*options=''*)

update or merge working directory

Update the working directory to the specified revision.

If there are no outstanding changes in the working directory and there is a linear relationship between the current version and the requested version, the result is the requested version.

To merge the working directory with another revision, use the merge command.

By default, update will refuse to run if doing so would require merging or discarding local changes.

aliases: up, checkout, co

INPUT:

- `options` - string (default: ''):

```
-C --clean  overwrite locally modified files
-d --date   tipmost revision matching date
-r --rev    revision
```

update(*options=''*)

update or merge working directory

Update the working directory to the specified revision.

If there are no outstanding changes in the working directory and there is a linear relationship between the current version and the requested version, the result is the requested version.

To merge the working directory with another revision, use the merge command.

By default, update will refuse to run if doing so would require merging or discarding local changes.

aliases: up, checkout, co

INPUT:

- `options` - string (default: ''):

```
-C --clean  overwrite locally modified files
-d --date   tipmost revision matching date
-r --rev    revision
```

what(*files='', rev=None*)

Show differences between revisions for the specified files as a unified diff.

By default this command tells you exactly what you have changed in your working repository since you last commited changes.

INPUT:

- `files` - space separated list of files (relative to self.dir())
- `rev` - None or a list of integers.

Differences between files are shown using the unified diff format.

When two revision arguments are given, then changes are shown between those revisions. If only one revision is specified then that revision is compared to the working directory, and, when no revisions are specified, the working directory files are compared to its parent.

get_remote_file(*f, **kwds*)

Wrap the get_remote_file method to move the file if it ends in ?stuff, as happens with funny urls from web servers.

pager()

Return a page program, which is either cat or less at present.

Return cat if embedded in the notebook, and less otherwise.

11.8 Functional notation

These are function so that you can write foo(x) instead of x.foo() in certain common cases.

AUTHORS:

- William Stein: Initial version
- David Joyner (2005-12-20): More Examples

N(*x, prec=None, digits=None*)

Return a numerical approximation of x with at least prec bits of precision.

Note: Both upper case N and lower case n are aliases for `numerical_approx()`.

INPUT:

- `x` - an object that has a numerical_approx method, or can be coerced into a real or complex field
- `prec (optional)` - an integer (bits of precision)
- `digits (optional)` - an integer (digits of precision)

If neither the prec or digits are specified, the default is 53 bits of precision.

EXAMPLES:

```
sage: numerical_approx(pi, 10)
3.1
sage: numerical_approx(pi, digits=10)
3.141592654
sage: numerical_approx(pi^2 + e, digits=20)
12.587886229548403854
sage: n(pi^2 + e)
12.5878862295484
sage: N(pi^2 + e)
12.5878862295484
sage: n(pi^2 + e, digits=50)
12.587886229548403854194778471228813633070946500941
sage: a = CC(-5).n(prec=100)
sage: b = ComplexField(100)(-5)
sage: a == b
True
sage: type(a) == type(b)
True
```

You can also usually use method notation:

```
sage: (pi^2 + e).n()
12.5878862295484
```

TESTS:

```
sage: numerical_approx(I)
1.00000000000000*I
sage: x = QQ['x'].gen()
sage: F.<k> = NumberField(x^2+2, embedding=sqrt(CC(2))*CC.0)
sage: numerical_approx(k)
1.41421356237309*I

sage: type(numerical_approx(CC(1/2)))
<type 'sage.rings.complex_number.ComplexNumber'>
```

acos(*x*)
Return the arc cosine of x.

additive_order(*x*)
Return the additive order of x.

arg(*x*)
Return the argument of a complex number x.

EXAMPLES:

```
sage: z = CC(1,2)
sage: theta = arg(z)
sage: cos(theta)*abs(z)
1.00000000000000
sage: sin(theta)*abs(z)
2.00000000000000
```

asin(*x*)
Return the arc sine of x.

atan(*x*)
Return the arc tangent of x.

base_field(*x*)
Return the base field over which x is defined.

base_ring(*x*)
Return the base ring over which x is defined.

EXAMPLES:

```
sage: R = PolynomialRing(GF(7), 'x')
sage: base_ring(R)
Finite Field of size 7
```

basis(*x*)
Return the fixed basis of x.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: S = V.subspace([[1,2,0],[2,2,-1]])
sage: basis(S)
[
(1, 0, -1),
(0, 1, 1/2)
]
```

category(*x*)
Return the category of x.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: category(V)
Category of vector spaces over Rational Field
```

ceil(*x*)

characteristic_polynomial(*x*, *var='x'*)

Return the characteristic polynomial of x in the given variable.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: charpoly(A)
x^3 - 15*x^2 - 18*x
sage: charpoly(A, 't')
t^3 - 15*t^2 - 18*t
```

```
sage: k.<alpha> = GF(7^10); k
Finite Field in alpha of size 7^10
sage: alpha.charpoly('T')
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3
sage: characteristic_polynomial(alpha, 'T')
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3
```

charpoly(*x*, *var='x'*)

Return the characteristic polynomial of x in the given variable.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: charpoly(A)
x^3 - 15*x^2 - 18*x
sage: charpoly(A, 't')
t^3 - 15*t^2 - 18*t
```

```
sage: k.<alpha> = GF(7^10); k
Finite Field in alpha of size 7^10
sage: alpha.charpoly('T')
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3
sage: characteristic_polynomial(alpha, 'T')
T^10 + T^6 + T^5 + 4*T^4 + T^3 + 2*T^2 + 3*T + 3
```

coerce(*P*, *x*)

cyclotomic_polynomial(*n*, *var='x'*)

EXAMPLES:

```
sage: cyclotomic_polynomial(3)
x^2 + x + 1
sage: cyclotomic_polynomial(4)
x^2 + 1
sage: cyclotomic_polynomial(9)
x^6 + x^3 + 1
sage: cyclotomic_polynomial(10)
x^4 - x^3 + x^2 - x + 1
```

```
sage: cyclotomic_polynomial(11)
x^10 + x^9 + x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + x^2 + x + 1
```

decomposition (*x*)
Return the decomposition of x.

denominator (*x*)
Return the denominator of x.

EXAMPLES:

```
sage: denominator(17/11111)
11111
sage: R.<x> = PolynomialRing(QQ)
sage: F = FractionField(R)
sage: r = (x+1)/(x-1)
sage: denominator(r)
x - 1
```

det (*x*)
Return the determinant of x.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: det(A)
0
```

dim (*x*)
Return the dimension of x.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: S = V.subspace([[1,2,0],[2,2,-1]])
sage: dimension(S)
2
```

dimension (*x*)
Return the dimension of x.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: S = V.subspace([[1,2,0],[2,2,-1]])
sage: dimension(S)
2
```

disc (*x*)
EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^29 - 17*x - 1, 'alpha')
sage: K = S.number_field()
sage: discriminant(K)
-15975100446626038280218213241591829458737190477345113376757479850566957249523
```

discriminant (*x*)
EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^29 - 17*x - 1, 'alpha')
sage: K = S.number_field()
sage: discriminant(K)
-15975100446626038280218213241591829458737190477345113376757479850566957249523
```

eta (*x*)
Return the value of the eta function at x, which must be in the upper half plane.

The η function is

$$\eta(z) = e^{\pi i z/12} \prod_{n=1}^{\infty} (1 - e^{2\pi i n z})$$

EXAMPLES:

```
sage: eta(1+I)
0.742048775837 + 0.19883137023*I
```

exp (*x*)
Return the value of the exponentation function at x.

factor (*x, *args, **kwds*)
Return the prime factorization of x.

EXAMPLES:

```
sage: factor(factorial(10))
2^8 * 3^4 * 5^2 * 7
sage: n = next_prime(10^6); n
1000003
sage: factor(n)
1000003
```

factorisation (*x, *args, **kwds*)
Return the prime factorization of x.

EXAMPLES:

```
sage: factor(factorial(10))
2^8 * 3^4 * 5^2 * 7
sage: n = next_prime(10^6); n
1000003
sage: factor(n)
1000003
```

factorization (*x, *args, **kwds*)
Return the prime factorization of x.

EXAMPLES:

```
sage: factor(factorial(10))
2^8 * 3^4 * 5^2 * 7
sage: n = next_prime(10^6); n
1000003
sage: factor(n)
1000003
```

fcp (*x, var='x'*)
Return the factorization of the characteristic polynomial of x.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: fcp(A, 'x')
x * (x^2 - 15*x - 18)
```

gen (*x*)
Return the generator of x.

gens (*x*)
Return the generators of x.

hecke_operator (*x, n*)
Return the n-th Hecke operator T_n acting on x.

EXAMPLES:

```
sage: M = ModularSymbols(1,12)
sage: hecke_operator(M,5)
Hecke operator T_5 on Modular Symbols space of dimension 3 for Gamma_0(1) of weight 12 with sign
```

image (*x*)
Return the image of x.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: image(A)
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
```

integral (*x, *args, **kwds*)
Return an indefinite integral of an object x.

First call x.integrate() and if that fails make an object and integrate it using maxima, maple, etc, as specified by algorithm.

EXAMPLES:

```
sage: f = cyclotomic_polynomial(10)
sage: integral(f)
1/5*x^5 - 1/4*x^4 + 1/3*x^3 - 1/2*x^2 + x
sage: integral(sin(x),x)
-cos(x)
sage: y = var('y')
sage: integral(sin(x),y)
y*sin(x)
sage: integral(sin(x), x, 0, pi/2)
1
sage: sin(x).integral(x, 0,pi/2)
1
```

integral_closure (*x*)

integrate(*x, *args, **kwds*)

Return an indefinite integral of an object x.

First call x.integrate() and if that fails make an object and integrate it using maxima, maple, etc, as specified by algorithm.

EXAMPLES:

```
sage: f = cyclotomic_polynomial(10)
sage: integral(f)
1/5*x^5 - 1/4*x^4 + 1/3*x^3 - 1/2*x^2 + x
sage: integral(sin(x),x)
-cos(x)
sage: y = var('y')
sage: integral(sin(x),y)
y*sin(x)
sage: integral(sin(x), x, 0, pi/2)
1
sage: sin(x).integral(x, 0,pi/2)
1
```

interval(*a, b*)

Integers between a and b *inclusive* (a and b integers).

EXAMPLES:

```
sage: I = interval(1,3)
sage: 2 in I
True
sage: 1 in I
True
sage: 4 in I
False
```

is_commutative(*x*)

EXAMPLES:

```
sage: R = PolynomialRing(QQ, 'x')
sage: is_commutative(R)
True
```

is_even(*x*)

Return whether or not an integer x is even, e.g., divisible by 2.

EXAMPLES:

```
sage: is_even(-1)
False
sage: is_even(4)
True
sage: is_even(-2)
True
```

is_field(*x*)

EXAMPLES:

```
sage: R = PolynomialRing(QQ, 'x')
sage: F = FractionField(R)
sage: is_field(F)
True
```

is_integrally_closed(*x*)

is_noetherian(*x*)

is_odd(*x*)
Return whether or not x is odd. This is by definition the complement of is_even.

EXAMPLES:

```
sage: is_odd(-2)
False
sage: is_odd(-3)
True
sage: is_odd(0)
False
sage: is_odd(1)
True
```

isqrt(*x*)
Return an integer square root, i.e., the floor of a square root.

EXAMPLES:

```
sage: isqrt(10)
3
sage: isqrt(10r)
3
```

kernel(*x*)
Return the left kernel of x.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,2)
sage: A = M([1,2,3,4,5,6])
sage: kernel(A)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
sage: kernel(A.transpose())
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

Here are two corner cases: sage: M=MatrixSpace(QQ,0,3) sage: A=M([]) sage: kernel(A) Vector space of degree 0 and dimension 0 over Rational Field Basis matrix: [] sage: kernel(A.transpose()).basis() [(1, 0, 0), (0, 1, 0), (0, 0, 1)]

krull_dimension(*x*)

lift(*x*)
Lift an object of a quotient ring R/I to R.

EXAMPLES: We lift an integer modulo 3.

```
sage: Mod(2,3).lift()
2
```

We lift an element of a quotient polynomial ring.

```
sage: R.<x> = QQ['x']
sage: S.<xmod> = R.quo(x^2 + 1)
sage: lift(xmod-7)
x - 7
```

log (*x, b=None*)

Return the log of x to the base b. The default base is e.

INPUT:

- x - number
- b - base (default: None, which means natural log)

OUTPUT: number

Note: In Magma, the order of arguments is reversed from in Sage, i.e., the base is given first. We use the opposite ordering, so the base can be viewed as an optional second argument.

minimal_polynomial (*x, var='x'*)

Return the minimal polynomial of x.

EXAMPLES:

```
sage: a = matrix(ZZ, 2, [1..4])
sage: minpoly(a)
x^2 - 5*x - 2
sage: minpoly(a,'t')
t^2 - 5*t - 2
sage: minimal_polynomial(a)
x^2 - 5*x - 2
sage: minimal_polynomial(a,'theta')
theta^2 - 5*theta - 2
```

minpoly (*x, var='x'*)

Return the minimal polynomial of x.

EXAMPLES:

```
sage: a = matrix(ZZ, 2, [1..4])
sage: minpoly(a)
x^2 - 5*x - 2
sage: minpoly(a,'t')
t^2 - 5*t - 2
sage: minimal_polynomial(a)
x^2 - 5*x - 2
sage: minimal_polynomial(a,'theta')
theta^2 - 5*theta - 2
```

multiplicative_order (*x*)

Return the multiplicative order of self, if self is a unit, or raise `ArithmeticError` otherwise.

n (*x, prec=None, digits=None*)

Return a numerical approximation of x with at least prec bits of precision.

Note: Both upper case N and lower case n are aliases for `numerical_approx()`.

INPUT:

- x - an object that has a numerical_approx method, or can be coerced into a real or complex field
- `prec (optional)` - an integer (bits of precision)
- `digits (optional)` - an integer (digits of precision)

If neither the prec or digits are specified, the default is 53 bits of precision.

EXAMPLES:

```
sage: numerical_approx(pi, 10)
3.1
sage: numerical_approx(pi, digits=10)
3.141592654
sage: numerical_approx(pi^2 + e, digits=20)
12.587886229548403854
sage: n(pi^2 + e)
12.5878862295484
sage: N(pi^2 + e)
12.5878862295484
sage: n(pi^2 + e, digits=50)
12.587886229548403854194778471228813633070946500941
sage: a = CC(-5).n(prec=100)
sage: b = ComplexField(100)(-5)
sage: a == b
True
sage: type(a) == type(b)
True
```

You can also usually use method notation:

```
sage: (pi^2 + e).n()
12.5878862295484
```

TESTS:

```
sage: numerical_approx(I)
1.00000000000000*I
sage: x = QQ['x'].gen()
sage: F.<k> = NumberField(x^2+2, embedding=sqrt(CC(2))*CC.0)
sage: numerical_approx(k)
1.41421356237309*I

sage: type(numerical_approx(CC(1/2)))
<type 'sage.rings.complex_number.ComplexNumber'>
```

ngens(*x*)

Return the number of generators of x.

norm(*x*)

Return the norm of x.

EXAMPLES:

```
sage: z = 1+2*I
sage: norm(z)
5
sage: norm(CDF(z))
5.0
sage: norm(CC(z))
5.00000000000000
```

numerator(*x*)

Return the numerator of x.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: F = FractionField(R)
sage: r = (x+1)/(x-1)
sage: numerator(r)
x + 1
sage: numerator(17/11111)
17
```

numerical_approx(*x, prec=None, digits=None*)

Return a numerical approximation of x with at least prec bits of precision.

Note: Both upper case N and lower case n are aliases for `numerical_approx()`.

INPUT:

- `x` - an object that has a numerical_approx method, or can be coerced into a real or complex field
- `prec (optional)` - an integer (bits of precision)
- `digits (optional)` - an integer (digits of precision)

If neither the prec or digits are specified, the default is 53 bits of precision.

EXAMPLES:

```
sage: numerical_approx(pi, 10)
3.1
sage: numerical_approx(pi, digits=10)
3.141592654
sage: numerical_approx(pi^2 + e, digits=20)
12.587886229548403854
sage: n(pi^2 + e)
12.5878862295484
sage: N(pi^2 + e)
12.5878862295484
sage: n(pi^2 + e, digits=50)
12.587886229548403854194778471228813633070946500941
sage: a = CC(-5).n(prec=100)
sage: b = ComplexField(100)(-5)
sage: a == b
True
sage: type(a) == type(b)
True
```

You can also usually use method notation:

```
sage: (pi^2 + e).n()
12.5878862295484
```

TESTS:

```
sage: numerical_approx(I)
1.00000000000000*I
sage: x = QQ['x'].gen()
sage: F.<k> = NumberField(x^2+2, embedding=sqrt(CC(2))*CC.0)
sage: numerical_approx(k)
1.41421356237309*I

sage: type(numerical_approx(CC(1/2)))
<type 'sage.rings.complex_number.ComplexNumber'>
```

objgen(*x*)
EXAMPLES:

```
sage: R, x = objgen(FractionField(QQ['x']))
sage: R
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: x
x
```

objgens(*x*)
EXAMPLES:

```
sage: R, x = objgens(PolynomialRing(QQ,3, 'x'))
sage: R
Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
sage: x
(x0, x1, x2)
```

one(*R*)
Return the one element of the ring R.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: one(R)*x == x
True
sage: one(R) in R
True
```

order(*x*)
Return the order of x. If x is a ring or module element, this is the additive order of x.

EXAMPLES:

```
sage: C = CyclicPermutationGroup(10)
sage: order(C)
10
sage: F = GF(7)
sage: order(F)
7
```

parent(*x*)
Return x.parent() if defined, or type(x) if not.

EXAMPLE:

```
sage: Z = parent(int(5))
sage: Z(17)
17
sage: Z
<type 'int'>
```

quo(*x, y, *args, **kwds*)
Return the quotient object x/y, e.g., a quotient of numbers or of a polynomial ring x by the ideal generated by y, etc.

quotient(*x, y, *args, **kwds*)
Return the quotient object x/y, e.g., a quotient of numbers or of a polynomial ring x by the ideal generated by y, etc.

rank(*x*)
Return the rank of x.

EXAMPLES: We compute the rank of a matrix:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: rank(A)
2
```

We compute the rank of an elliptic curve:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: rank(E)
1
```

regulator(*x*)
Return the regulator of x.

round(*x, ndigits=0*)
round(number[, ndigits]) - double-precision real number

Round a number to a given precision in decimal digits (default 0 digits). This always returns a real double field element.

EXAMPLES:

```
sage: round(sqrt(2),2)
1.41
sage: round(sqrt(2),5)
1.41421
sage: round(pi)
3.0
sage: b = 5.4999999999999999
sage: round(b)
5.0
```

Since we use floating-point with a limited range, some roundings can't be performed:

```
sage: round(sqrt(Integer('1'*1000)))
+infinity
```

IMPLEMENTATION: If ndigits is specified, it calls Python's builtin round function, and converts the result to a real double field element. Otherwise, it tries the argument's .round() method, and if that fails, it falls back to the builtin round function.

Note: This is currently slower than the builtin round function, since it does more work - i.e., allocating an RDF element and initializing it. To access the builtin version do `import __builtin__; __builtin__.round`.

show(*x, *args, **kwds*)
Show a graphics object x.

For additional ways to show objects in the notebook, look at the methods on the html object. For example, html.table will produce an HTML table from a nested list.

OPTIONAL INPUT:

- `filename` - (default: None) string

SOME OF THESE MAY APPLY:

- `dpi` - dots per inch

- `figsize` - [width, height] (same for square aspect)
- `axes` - (default: True)
- `fontsize` - positive integer
- `frame` - (default: False) draw a MATLAB-like frame around the image

EXAMPLES:

```
sage: show(graphs(3))
sage: show(list(graphs(3)))
```

sqrt (*x*)

Return a square root of x.

This function (`numerical_sqrt`) is deprecated. Use `sqrt(x, prec=n)` instead.

EXAMPLES:

```
sage: numerical_sqrt(10.1)
doctest:1: DeprecationWarning: numerical_sqrt is deprecated, use sqrt(x, prec=n) instead
3.17804971641414
sage: numerical_sqrt(9)
3
```

squarefree_part (*x*)

Return the square free part of x, i.e., a divisor z such that $x = zy^2$, for a perfect square y^2.

EXAMPLES:

```
sage: squarefree_part(100)
1
sage: squarefree_part(12)
3
sage: squarefree_part(10)
10
```

```
sage: x = QQ['x'].0
sage: S = squarefree_part(-9*x*(x-6)^7*(x-3)^2); S
-9*x^2 + 54*x
sage: S.factor()
(-9) * (x - 6) * x
```

```
sage: f = (x^3 + x + 1)^3*(x-1); f
x^10 - x^9 + 3*x^8 + 3*x^5 - 2*x^4 - x^3 - 2*x - 1
sage: g = squarefree_part(f); g
x^4 - x^3 + x^2 - 1
sage: g.factor()
(x - 1) * (x^3 + x + 1)
```

transpose (*x*)

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,2,3,4,5,6,7,8,9])
sage: transpose(A)
[1 4 7]
[2 5 8]
[3 6 9]
```

xinterval(*a, b*)
Iterator over the integers between a and b, *inclusive*.

zero(*R*)
Return the zero element of the ring R.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: zero(R) in R
True
sage: zero(R)*x == zero(R)
True
```

11.9 LaTeX printing support

In order to support latex formating, an object should define a special method _latex_(self) that returns a string.

class JSMath()
A simple object for rendering LaTeX input using JSMath.

eval(*x, globals=None, locals=None, mode='display'*)

class JSMathExpr(*y*)
An arbitrary JSMath expression that can be nicely concatenated.

class Latex(*debug=False, slide=False, density=150, pdflatex=None*)
Enter, e.g.,

```
%latex
The equation $y^2 = x^3 + x$ defines an elliptic curve.
We have $2006 = Sage{factor(2006)}$.
```

in an input cell to get a typeset version. Use `%latex_debug` to get debugging output.

Use `latex(...)` to typeset a Sage object.

Use `%slide` instead to typeset slides.

> **Warning:** You must have dvipng (or dvips and convert) installed on your operating system, or this command won't work.

add_macro(*macro*)
Append to the string of extra LaTeX macros, for use with %latex, %html, and %jsmath.

INPUT: `macro` - string

EXAMPLES:: sage: latex.extra_macros() '' sage: latex.add_macro("newcommand{foo}{bar}") sage: latex.extra_macros() 'newcommand{foo}{bar}' sage: latex.extra_macros("") # restore to default

add_to_preamble(*s*)
Append to the string of extra LaTeX macros, for use with %latex. Anything in this string won't be processed by %jsmath.

EXAMPLES:

```
sage: latex.extra_preamble()
''
sage: latex.add_to_preamble("\\DeclareMathOperator{\\Ext}{Ext}")
```

At this point, a notebook cell containing

```
%latex
$\Ext_A^*(\GF{2}, \GF{2}) \Rightarrow \pi_*^s*(S^0)$
```

will be typeset correctly.

```
sage: latex.add_to_preamble("\\usepackage{xypic}")
sage: latex.extra_preamble()
'\\DeclareMathOperator{\\Ext}{Ext}\\usepackage{xypic}'
```

Now one can put various xypic diagrams into a %latex cell, such as

```
%latex
\[ \xymatrix{ \circ \ar `r[d]^{a} `[rr]^{b} `/4pt[rr]^{c} `[rrr]^{d}
`_dl[drrr]^{e} [drrr]^{f} & \circ & \circ & \circ \\ \circ & \circ &
\circ & \circ } \]
```

Reset the preamble to its default, the empty string:

```
sage: latex.extra_preamble('')
sage: latex.extra_preamble()
''
```

blackboard_bold(*t=None*)

Controls whether Sage uses blackboard bold or ordinary bold face for typesetting ZZ, RR, etc.

INPUT:

- t – boolean or None

OUTPUT: if t is None, return the current setting (True or False).

If t == True, use blackboard bold (mathbb); otherwise use boldface (mathbf).

EXAMPLES:

```
sage: latex.blackboard_bold()
False
sage: latex.blackboard_bold(True)
sage: latex.blackboard_bold()
True
sage: latex.blackboard_bold(False)
```

eval(*x, globals, strip=False, filename=None, debug=None, density=None, pdflatex=None, locals={}*)

INPUT: globals – a globals dictionary

- x - string to evaluate.
- strip - ignored
- filename - output filename
- debug - whether to print verbose debugging output
- density - how big output image is.
- pdflatex - whether to use pdflatex.
- locals - extra local variables used when evaluating Sage.. code in x.

> **Warning:** When using latex (the default), you must have 'dvipng' (or 'dvips' and 'convert') installed on your operating system, or this command won't work. When using pdflatex, you must have 'convert' installed.

extra_macros(*macros=None*)

String containing extra LaTeX macros to use with %latex, %html, and %jsmath.

INPUT: macros - string

If macros is None, return the current string. Otherwise, set it to macros. If you want to *append* to the string of macros instead of replacing it, using latex.add_macro.

EXAMPLES:

```
sage: latex.extra_macros("\newcommand{\foo}{bar}")
sage: latex.extra_macros()
'\newcommand{\foo}{bar}'
sage: latex.extra_macros("")
sage: latex.extra_macros()
''
```

extra_preamble(*s=None*)

String containing extra preamble to be used with %latex. Anything in this string won't be processed by %jsmath.

INPUT: `s` - string or `None`

If `s` is None, return the current preamble. Otherwise, set it to `s`. If you want to *append* to the current extra preamble instead of replacing it, using `latex.add_to_preamble`.

EXAMPLES:

```
sage: latex.extra_preamble("\DeclareMathOperator{\Ext}{Ext}")
sage: latex.extra_preamble()
'\DeclareMathOperator{\Ext}{Ext}'
sage: latex.extra_preamble("")
sage: latex.extra_preamble()
''
```

matrix_delimiters(*left=None, right=None*)

Change the left and right delimiters for the LaTeX representation of matrices

INPUT:

- `left`, `right` - strings or None

If both `left` and `right` are `None`, then return the current delimiters. Otherwise, set the left and/or right delimiters, whichever are specified.

Good choices for `left` and `right` are any delimiters which LaTeX understands and knows how to resize; some examples are:

- parentheses: '(', ')'
- brackets: '[', ']'
- braces: '\{', '\}'
- vertical lines: '|'
- angle brackets: '\langle', '\rangle'

Note: Putting aside aesthetics, you may combine these in any way imaginable; for example, you could set `left` to be a right-hand bracket ']' and `right` to be a right-hand brace '\}', and it will be typeset correctly.

EXAMPLES:

```
sage: a = matrix(1, 1, [17])
sage: latex(a)
\left(\begin{array}{r}
17
\end{array}\right)
sage: latex.matrix_delimiters("[", "]")
sage: latex(a)
\left[\begin{array}{r}
17
\end{array}\right]
sage: latex.matrix_delimiters(left="\\{")
sage: latex(a)
\left\{\begin{array}{r}
17
```

```
\end{array}\right]
sage: latex.matrix_delimiters()
['\\{', ']']
```

Restore defaults:

```
sage: latex.matrix_delimiters("(", ")")
```

pdflatex(*t=None*)

Controls whether Sage uses PDFLaTeX or LaTeX when typesetting with `view`, in `%latex` cells, etc.

INPUT:

- `t` – boolean or None

OUTPUT: if t is None, return the current setting (True or False).

If t == True, use PDFLaTeX; otherwise use LaTeX.

EXAMPLES:

```
sage: latex.pdflatex()
False
sage: latex.pdflatex(True)
sage: latex.pdflatex()
True
sage: latex.pdflatex(False)
```

vector_delimiters(*left=None, right=None*)

Change the left and right delimiters for the LaTeX representation of vectors

INPUT:

- `left`, `right` - strings or None

If both `left` and `right` are `None`, then return the current delimiters. Otherwise, set the left and/or right delimiters, whichever are specified.

Good choices for `left` and `right` are any delimiters which LaTeX understands and knows how to resize; some examples are:

- parentheses: '(', ')'
- brackets: '[', ']'
- braces: '\{', '\}'
- vertical lines: '|'
- angle brackets: '\langle', '\rangle'

Note: Putting aside aesthetics, you may combine these in any way imaginable; for example, you could set `left` to be a right-hand bracket ']' and `right` to be a right-hand brace '\}', and it will be typeset correctly.

EXAMPLES:

```
sage: a = vector(QQ, [1,2,3])
sage: latex(a)
\left(1,2,3\right)
sage: latex.vector_delimiters("[", "]")
sage: latex(a)
\left[1,2,3\right]
sage: latex.vector_delimiters(right="\\}")
sage: latex(a)
\left[1,2,3\right\}
sage: latex.vector_delimiters()
['[', '\\}']
```

Restore defaults:

```
sage: latex.vector_delimiters("(", ")")
```

class LatexExpr(*x*)

bool_function(*x*)

Returns the LaTeX code for a tuple x.

INPUT: x - boolean

EXAMPLES:

```
sage: from sage.misc.latex import bool_function
sage: bool_function(2==3)
'\\mbox{\\rm False}'
sage: bool_function(3==(2+1))
'\\mbox{\\rm True}'
```

coeff_repr(*c*)

have_convert()

Return True if this computer has the program convert.

The first time it is run, this function caches its result in the variable `_have_convert`, and any subsequent time, it just checks the value of the variable.

If this computer doesn't have convert installed, you may obtain it (along with the rest of the ImageMagick suite) from http://www.imagemagick.org

EXAMPLES:

```
sage: from sage.misc.latex import have_convert
sage: sage.misc.latex._have_convert is None
True
sage: have_convert() # random
True
sage: sage.misc.latex._have_convert is None
False
sage: sage.misc.latex._have_convert == have_convert()
True
```

have_dvipng()

Return True if this computer has the program dvipng.

The first time it is run, this function caches its result in the variable `_have_dvipng`, and any subsequent time, it just checks the value of the variable.

If this computer doesn't have dvipng installed, you may obtain it from http://sourceforge.net/projects/dvipng/

EXAMPLES:

```
sage: from sage.misc.latex import have_dvipng
sage: sage.misc.latex._have_dvipng is None
True
sage: have_dvipng() # random
True
sage: sage.misc.latex._have_dvipng is None
False
sage: sage.misc.latex._have_dvipng == have_dvipng()
True
```

jsmath(*x, mode='display'*)

Attempt to nicely render an arbitrary SAGE object wih jsmath typesetting. Tries to call ._latex_() on x. If that fails, it will render a string representation of x.

Warning: 2009-04: This function is deprecated; use `html` instead: replace `jsmath('MATH', mode='display')` with `html('$$MATH$$')`, and replace `jsmath('MATH', mode='inline')` with `html('$MATH$')`.

INPUT: x – the object to render mode – 'display' for displaymath or 'inline' for inline math

OUTPUT: A string of html that contains the LaTeX represntation of x. In the notebook this gets embedded into the cell.

EXAMPLES:

```
sage: from sage.misc.latex import jsmath
sage: f = maxima('1/(x^2+1)')
sage: g = f.integrate()
sage: jsmath(f)
... DeprecationWarning: The jsmath function is deprecated.  Use html('$math$') for inline mode c
# -*- coding: utf-8 -*-
<html><font color='black'><div class="math">{{1}\over{x^2+1}}</div></font></html>
sage: jsmath(g, 'inline')
<html><font color='black'><span class="math">\tan^{-1} x</span></font></html>
sage: jsmath('\int' + latex(f) + '\ dx=' + latex(g))
<html><font color='black'><div class="math">\int{{1}\over{x^2+1}}\ dx=\tan^{-1} x</div></font></
```

AUTHORS:

- William Stein (2006-10): general layout (2006-10)
- Bobby Moretti (2006-10): improvements, comments, documentation

latex_extra_preamble()

Return the string containing the user-configured preamble, `sage_latex_macros`, and any user-configured macros. This is used in the `eval` method for the `Latex` class, and in `_latex_file`; it follows either `LATEX_HEADER` or `SLIDE_HEADER` (defined at the top of this file) which is a string containing the documentclass and standard usepackage commands.

EXAMPLES:

```
sage: from sage.misc.latex import latex_extra_preamble
sage: latex_extra_preamble()
'\n\\newcommand{\\ZZ}{\\Bold{Z}}\n\\newcommand{\\RR}{\\Bold{R}}\n\\newcommand{\\CC}{\\Bold{C}}\n
```

latex_variable_name(*x*)

Return latex version of a variable name.

Here are some guiding principles for usage of this function:

1. If the variable is a single letter, that is the latex version.
2. If the variable name is suffixed by a number, we put the number in the subscript.
3. If the variable name contains an '_' we start the subscript at the underscore. Note that #3 trumps rule #2.
4. If a component of the variable is a greek letter, escape it properly.
5. Recurse nicely with subscripts.

Refer to the examples section for how these rules might play out in practice.

EXAMPLES:

```
sage: from sage.misc.latex import latex_variable_name
sage: latex_variable_name('a')
'a'
sage: latex_variable_name('abc')
'\\mbox{abc}'
sage: latex_variable_name('sigma')
'\\sigma'
sage: latex_variable_name('sigma_k')
'\\sigma_{k}'
sage: latex_variable_name('sigma389')
'\\sigma_{389}'
sage: latex_variable_name('beta_00')
'\\beta_{00}'
sage: latex_variable_name('Omega84')
'\\Omega_{84}'
sage: latex_variable_name('sigma_alpha')
'\\sigma_{\\alpha}'
sage: latex_variable_name('nothing1')
'\\mbox{nothing}_{1}'
sage: latex_variable_name('nothing_abc')
'\\mbox{nothing}_{\\mbox{abc}}'
sage: latex_variable_name('alpha_beta_gamma12')
'\\alpha_{\\beta_{\\gamma_{12}}}'
```

AUTHORS:

- Joel B. Mohler: drastic rewrite and many doc-tests

latex_varify(*a*)

list_function(*x*)

Returns the LaTeX code for a list x.

INPUT: x - a list

EXAMPLES:

```
sage: from sage.misc.latex import list_function
sage: list_function([1,2,3])
'\\left[1, \n 2, \n 3\\right]'
sage: latex([1,2,3])  # indirect doctest
\left[1,
2,
3\right]
sage: latex([Matrix(ZZ,3,range(9)), Matrix(ZZ,3,range(9))]) # indirect doctest
\left[\left(\begin{array}{rrr}
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8
\end{array}\right),
\\\left(\begin{array}{rrr}
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8
\end{array}\right)\right]
```

png(*x, filename, density=150, debug=False, do_in_background=True, tiny=False*)

Create a png image representation of x and save to the given filename.

INPUT:

- `x` - object to be displayed
- `filename` - file in which to save the image
- `density` - integer (default: 150)
- `debug` - bool (default: False): print verbose output
- `do_in_background` - bool (default: True): create the file in the background
- `tiny` - bool (default: False): use 'tiny' font

pretty_print (*object*)
Try to pretty print the object in an intelligent way. For many things, this will convert the object to latex inside of html and rely on a latex-aware front end (like jsMath) to render the text

pretty_print_default (*enable=True*)
Enable or disable default pretty printing. Pretty printing means rendering things so that jsMath or some other latex-aware front end can render real math.

print_or_typeset (*object*)
'view' or 'print' the object depending on the situation.

In particular, if in notebook mode with the typeset box checked, view the object. Otherwise, print the object.

INPUT: object: anything

EXAMPLES:

```
sage: sage.misc.latex.print_or_typeset(3)
3
sage: sage.misc.latex.EMBEDDED_MODE=True
sage: sage.misc.latex.print_or_typeset(3)
3
sage: TEMP = sys.displayhook
sage: sys.displayhook = sage.misc.latex.pretty_print
sage: sage.misc.latex.print_or_typeset(3)
<html><span class="math">\newcommand{\Bold}[1]{\mathbf{#1}}3</span></html>
sage: sage.misc.latex.EMBEDDED_MODE=False
sage: sys.displayhook = TEMP
```

repr_lincomb (*symbols, coeffs*)
Compute a latex representation of a linear combination of some formal symbols.

INPUT:

- `symbols` - list of symbols
- `coeffs` - list of coefficients of the symbols

OUTPUT:

- `str` - a string

EXAMPLES:

```
sage: t = PolynomialRing(QQ, 't').0
sage: from sage.misc.latex import repr_lincomb
sage: repr_lincomb(['a', 's', ''], [-t, t - 2, t^12 + 2])
'-t\text{a} + \left(t - 2\right)\text{s} + \left(t^{12} + 2\right)\text{}'
sage: repr_lincomb(['a', 'b'], [1,1])
'\text{a} + \text{b}'
```

Verify that a certain corner case works (see trac 5707 and 5766):

```
sage: repr_lincomb([1,5,-3],[2,8/9,7])
'2\cdot 1 + \frac{8}{9}\cdot 5 + 7\cdot -3'
```

str_function(*x*)

Returns the LaTeX code for a string x.

INPUT: x - a string

EXAMPLES:

```
sage: from sage.misc.latex import str_function
sage: str_function('hello world')
'\\text{hello world}'
```

tuple_function(*x*)

Returns the LaTeX code for a tuple x.

INPUT: x - a tuple

EXAMPLES:

```
sage: from sage.misc.latex import tuple_function
sage: tuple_function((1,2,3))
'\\left(1, \n 2, \n 3\\right)'
```

typeset(*x*)

view(*objects, title='SAGE', debug=False, sep='', tiny=False, pdflatex=None, **kwds*)

Compute a latex representation of each object in objects, compile, and display typeset. If used from the command line, this requires that latex be installed.

INPUT:

- `objects` - list (or object)
- `title` - string (default: 'Sage'): title for the document
- `debug` - bool (default: False): print verbose output
- `sep` - string (default: ''): separator between math objects
- `tiny` - bool (default: False): use tiny font.
- `pdflatex` - bool (default: False): use pdflatex.

OUTPUT: Display typeset objects.

This function behaves differently depending on whether in notebook mode or not.

If not in notebook mode, this opens up a window displaying a dvi (or pdf) file, displaying the following: the title string is printed, centered, at the top. Beneath that, each object in objects is typeset on its own line, with the string sep inserted between these lines.

The value of `sep` is inserted between each element of the list `objects`; you can, for example, add vertical space between objects with `sep='\\vspace{15mm}'`, while `sep='\\hrule'` adds a horizontal line between objects, and `sep='\\newpage'` inserts a page break between objects.

If in notebook mode, this uses jmath to display the output in the notebook. Only the first argument, objects, is relevant; the others are ignored. If objects is a list, the result is typeset as a Python list, e.g. [12, -3.431] - each object in the list is not printed on its own line.

EXAMPLES:

```
sage: sage.misc.latex.EMBEDDED_MODE = True
sage: view(3)
<html><span class="math">\newcommand{\Bold}[1]{\mathbf{#1}}3</span></html>
sage: sage.misc.latex.EMBEDDED_MODE = False
```

11.10 LaTeX macros

AUTHORS:

- John H. Palmieri (2009-03)

The code here sets up LaTeX macro definitions for use in the documentation. To add a macro, modify the list `macros`, near the end of this file, and then run 'sage -b'. The entries in this list are used to produce `sage_latex_macros`, a list of strings of the form '\newcommand...', and `sage_js_macros`, a list of strings of the form 'jsMath.Macro...'. The LaTeX macros are produced using the `_latex_` method for each Sage object listed in `macros`, and the jsMath macros are produced from the LaTeX macros. The list of LaTeX macros is used in the file `SAGE_ROOT/devel/sage/doc/common/conf.py` to add to the preambles of both the LaTeX file used to build the PDF version of the documentation and the LaTeX file used to build the html version. The list of jsMath macros is used in the file `sage/server/notebook/notebook.py` to define jsMath macros for use in the live documentation (and also in the notebook).

Any macro defined here may be used in docstrings or in the tutorial (or other pieces of documentation). In a docstring, for example, "ZZ" in backquotes (demarking math mode) will appear as "ZZ" in interactive help, but will be typeset as "\Bold{Z}" in the reference manual.

More details on the list `macros`: the entries are lists or tuples of the form `[name]` or `[name, arguments]`, where `name` is a string and `arguments` consists of valid arguments for the Sage object named `name`. For example, `["ZZ"]` and `["GF", 2]` produce the LaTeX macros '\newcommand{\ZZ}{\Bold{Z}}' and '\newcommand{\GF}[1]{\Bold{F}_{#1}}', respectively. (For the second of these, `latex(GF(2))` is called and the string '2' gets replaced by '#1', so `["GF", 17]` would have worked just as well. `["GF", p]` would have raised an error, though, because `p` is not defined, and `["GF", 4]` would have raised an error, because to define the field with four elements in Sage, you also need to specify the name of a generator.)

To see evidence of the results of the code here, run `sage -docbuild tutorial latex` (for example), and look at the resulting LaTeX file in `SAGE_ROOT/sage/doc/output/latex/en/tutorial/`. The preamble should contain 'newcommand' lines for each of the entries in `macros`.

convert_latex_macro_to_jsmath(*macro*)

This converts a LaTeX macro definition (newcommand...) to a jsMath macro definition (jsMath.Macro...).

INPUT:

- `macro` - LaTeX macro definition

See the web page http://www.math.union.edu/~dpvc/jsMath/authors/macros.html for a description of the format for jsMath macros.

EXAMPLES:

```
sage: from sage.misc.latex_macros import convert_latex_macro_to_jsmath
sage: convert_latex_macro_to_jsmath('\\newcommand{\\ZZ}{\\Bold{Z}}')
"jsMath.Macro('ZZ','\\\\Bold{Z}')"
sage: convert_latex_macro_to_jsmath('\\newcommand{\\GF}[1]{\\Bold{F}_{#1}}')
"jsMath.Macro('GF','\\\\Bold{F}_{#1}',1)"
```

produce_latex_macro(*name, *sample_args*)

Produce a string defining a LaTeX macro.

INPUT:

- `name` - name of macro to be defined, also name of corresponding Sage object
- `sample_args` - (optional) sample arguments for this Sage object

EXAMPLES:

```
sage: from sage.misc.latex_macros import produce_latex_macro
sage: produce_latex_macro('ZZ')
'\\newcommand{\\ZZ}{\\Bold{Z}}'
```

If the Sage object takes arguments, then the LaTeX macro will accept arguments as well. You must pass valid arguments, which will then be converted to #1, #2, etc. in the macro definition. The following allows the use of "GF{p^n}", for example:

```
sage: produce_latex_macro('GF', 37)
'\\newcommand{\\GF}[1]{\\Bold{F}_{#1}}'
```

If the Sage object is not in the global name space, describe it like so:

```
sage: produce_latex_macro('sage.rings.finite_field.FiniteField', 3)
'\\newcommand{\\FiniteField}[1]{\\Bold{F}_{#1}}'
```

11.11 Lazy attributes

class **lazy_attribute**(*f*)

A lazy attribute for an object is like a usual attribute, except that, instead of being computed when the object is constructed (i.e. in __init__), it is computed on the fly the first time it is accessed.

For constant values attached to an object, lazy attributes provide a shorter syntax and automatic caching (unlike methods), while playing well with inheritance (like methods): a subclass can easily override a given attribute; you don't need to call the super class constructor, etc.

Technically, a lazy_attribute is a non-data descriptor (see Invoking Descriptors in the Python reference manual).

EXAMPLES:

We create a class whose instances have a lazy attribute x:

```
sage: class A(object):
...       def __init__(self):
...           self.a=2 # just to have some data to calculate from
...
...       @lazy_attribute
...       def x(self):
...           print "calculating x in A"
...           return self.a + 1
...
```

For an instance a of A, a.x is calculated the first time it is accessed, and then stored as a usual attribute:

```
sage: a = A()
sage: a.x
calculating x in A
3
sage: a.x
3
```

Implementation details

We redo the same example, but opening the hood to see what happens to the internal dictionary of the object:

```
sage: a = A()
sage: a.__dict__
{'a': 2}
sage: a.x
calculating x in A
3
sage: a.__dict__
{'a': 2, 'x': 3}
sage: a.x
3
sage: timeit('a.x') # random
625 loops, best of 3: 89.6 ns per loop
```

This shows that, after the first calculation, the attribute x becomes a usual attribute; in particular, there is no time penalty to access it.

A lazy attribute may be set as usual, even before its first access, in which case the lazy calculation is completely ignored:

```
sage: a = A()
sage: a.x = 4
sage: a.x
4
sage: a.__dict__
{'a': 2, 'x': 4}
```

Class binding results in the lazy attribute itself:

```
sage: A.x
<sage.misc.lazy_attribute.lazy_attribute object at ...>
```

Conditional definitions

The function calculating the attribute may return NotImplemented to declare that, after all, it is not able to do it. In that case, the attribute lookup proceeds in the super class hierarchy:

```
sage: class B(A):
...       @lazy_attribute
...       def x(self):
...           if hasattr(self, "y"):
...               print "calculating x from y in B"
...               return self.y
...           else:
...               print "y not there; B does not define x"
...               return NotImplemented
...
sage: b = B()
sage: b.x
y not there; B does not define x
calculating x in A
3
sage: b = B()
sage: b.y = 1
sage: b.x
calculating x from y in B
1
```

Attribute existence testing

Testing for the existence of an attribute with hasattr currently always triggers its full calculation, which may not be desirable when the calculation is expensive:

```
sage: a = A()
sage: hasattr(a, "x")
calculating x in A
True
```

It would be great if we could take over the control somehow, if at all possible without a special implementation of hasattr, so as to allow for something like:

```
sage: class A (object):
...       @lazy_attribute
...       def x(self, existence_only=False):
...           if existence_only:
...               print "testing for x existence"
...               return True
...           else:
...               print "calculating x in A"
...               return 3
...
sage: a = A()
sage: hasattr(a, "x") # todo: not implemented
testing for x existence
sage: a.x
calculating x in A
3
sage: a.x
3
```

Here is a full featured example, with both conditional definition and existence testing:

```
sage: class B(A):
...       @lazy_attribute
...       def x(self, existence_only=False):
...           if hasattr(self, "y"):
...               if existence_only:
...                   print "testing for x existence in B"
...                   return True
...               else:
...                   print "calculating x from y in B"
...                   return self.y
...           else:
...               print "y not there; B does not define x"
...               return NotImplemented
...
sage: b = B()
sage: hasattr(b, "x") # todo: not implemented
y not there; B does not define x
testing for x existence
True
sage: b.x
y not there; B does not define x
calculating x in A
3
sage: b = B()
```

```
sage: b.y = 1
sage: hasattr(b, "x") # todo: not implemented
testing for x existence in B
True
sage: b.x
calculating x from y in B
1
```

lazy attributes and introspection

TODO: make the following work nicely:

```
sage:  b.x?                # todo: not implemented
sage:  b.x??               # todo: not implemented
```

Right now, the first one includes the doc of this class, and the second one brings up the code of this class, both being not very useful.

TESTS:

```
.. rubric:: Partial support for old style classes
```

Old style and new style classes play a bit differently with @property and attribute setting:

```
sage: class A:
...       @property
...       def x(self):
...           print "calculating x"
...           return 3
...
sage: a = A()
sage: a.x = 4
sage: a.__dict__
{'x': 4}
sage: a.x
4
sage: a.__dict__['x']=5
sage: a.x
5

sage: class A (object):
...       @property
...       def x(self):
...           print "calculating x"
...           return 3
...
sage: a = A()
sage: a.x = 4
...
AttributeError: can't set attribute
sage: a.__dict__
{}
sage: a.x
calculating x
3
sage: a.__dict__['x']=5
sage: a.x
```

```
calculating x
3
```

In particular, lazy_attributes need to be implemented as non-data descriptors for new style classes, so as to leave access to setattr. We now check that this implementation also works for old style classes (conditional definition does not work yet):

```
sage: class A:
...       def __init__(self):
...           self.a=2 # just to have some data to calculate from
...
...       @lazy_attribute
...       def x(self):
...           print "calculating x"
...           return self.a + 1
...
sage: a = A()
sage: a.__dict__
{'a': 2}
sage: a.x
calculating x
3
sage: a.__dict__
{'a': 2, 'x': 3}
sage: a.x
3
sage: timeit('a.x') # random
625 loops, best of 3: 115 ns per loop

sage: a = A()
sage: a.x = 4
sage: a.x
4
sage: a.__dict__
{'a': 2, 'x': 4}

sage: class B(A):
...       @lazy_attribute
...       def x(self):
...           if hasattr(self, "y"):
...               print "calculating x from y in B"
...               return self.y
...           else:
...               print "y not there; B does not define x"
...               return NotImplemented
...
sage: b = B()
sage: b.x                          # todo: not implemented
y not there; B does not define x
calculating x in A
3
sage: b = B()
sage: b.y = 1
sage: b.x
calculating x from y in B
1
```

lazy_attributes and cpdef functions

This attempts to check that lazy_attributes work with builtin functions like cpdef methods:

```
sage: class A:
...       def __len__(x):
...           return int(5)
...       len = lazy_attribute(len)
...
sage: A().len
5
```

About descriptor specifications

The specifications of descriptors (see 3.4.2.3 Invoking Descriptors in the Python reference manual) are incomplete w.r.t. inheritence, and maybe even ill-implemented. We illustrate this on a simple class hierarchy, with an instrumented descriptor:

```
sage: class descriptor(object):
...       def __get__(self, obj, cls):
...           print cls
...           return 1
...
sage: class A(object):
...       x = descriptor()
...
sage: class B(A):
...       pass
...
```

This is fine:

```
sage: A.x
<class '__main__.A'>
1
```

The behaviour for the following case is not specified (see Instance Binding) when x is not in the dictionary of B but in that of some super category:

```
sage: B().x
<class '__main__.B'>
1
```

It would seem more natural (and practical!) to get A rather than B.

From the specifications for Super Binding, it would be expected to get A and not B as cls parameter:

```
sage: super(B, B()).x
<class '__main__.B'>
1
```

Due to this, the natural implementation runs into an infinite loop in the following example:

```
sage: class A(object):
...       @lazy_attribute
...       def unimplemented_A(self):
...           return NotImplemented
```

```
...       @lazy_attribute
...       def unimplemented_AB(self):
...           return NotImplemented
...       @lazy_attribute
...       def unimplemented_B_implemented_A(self):
...           return 1
...
sage: class B(A):
...       @lazy_attribute
...       def unimplemented_B(self):
...           return NotImplemented
...       @lazy_attribute
...       def unimplemented_AB(self):
...           return NotImplemented
...       @lazy_attribute
...       def unimplemented_B_implemented_A(self):
...           return NotImplemented
...
sage: class C(B):
...       pass
...
```

This is the simplest case where, without workaround, we get an infinite loop:

```
sage: hasattr(B(), "unimplemented_A") # todo: not implemented
False
```

TODO: improve the error message:

```
sage: B().unimplemented_A # todo: not implemented
...
AttributeError: 'super' object has no attribute 'unimplemented_A'
```

We now make some systematic checks:

```
sage: B().unimplemented_A
...
AttributeError: '...' object has no attribute 'unimplemented_A'
sage: B().unimplemented_B
...
AttributeError: '...' object has no attribute 'unimplemented_B'
sage: B().unimplemented_AB
...
AttributeError: '...' object has no attribute 'unimplemented_AB'
sage: B().unimplemented_B_implemented_A
1

sage: C().unimplemented_A()
...
AttributeError: '...' object has no attribute 'unimplemented_A'
sage: C().unimplemented_B()
...
AttributeError: '...' object has no attribute 'unimplemented_B'
sage: C().unimplemented_AB()
...
AttributeError: '...' object has no attribute 'unimplemented_AB'
sage: C().unimplemented_B_implemented_A # todo: not implemented
1
```

11.12 Logging of Sage sessions.

TODO: Pressing "control-D" can mess up the I/O sequence because of a known bug.

You can create a log of your Sage session as a web page and/or as a latex document. Just type `log_html()` to create an HTML log, or `log_dvi()` to create a dvi (LaTeX) log. Your complete session so far up until when you type the above command will be logged, along with any future input. Thus you can view the log system as a way to print or view your entire session so far, along with a way to see nicely typeset incremental updates as you work.

If `L=log_dvi()` or `L=log_html()` is a logger, you can type `L.stop()` and `L.start()` to stop and start logging.

The environment variables `BROWSER` and `DVI_VIEWER` determine which web browser or dvi viewer is used to display your running log.

For both log systems you must have a TeX system installed on your computer. For HTML logging, you must have the convert command, which comes with the free ImageMagick tools.

Note: The HTML output is done via LaTeX and png images right now, sort of like how latex2html works. Obviously it would be interesting to do something using MathML in the long run.

AUTHORS:

- William Stein (2006-02): initial version
- William Stein (2006-02-27): changed html generation so log directory is relocatable (no hardcoded paths).
- William Stein (2006-03-04): changed environment variable to BROWSER.
- Didier Deshommes (2006-05-06): added MathML support; refactored code.
- Dan Drake (2008-03-27): fix bit rotting so that optional directories work, dvi logging works, viewer() command works, remove no-longer-working MathML logger; fix off-by-one problems with IPython history; add text logger; improve documentation about viewers.

class Log (*dir=None, debug=False, viewer=None*)

This is the base logger class. The two classes that you actually instantiate are derived from this one.

dir()

Return the directory that contains the log files.

start()

Start the logger. To stop use the stop function.

stop()

Stop the logger. To restart use the start function.

class log_dvi (*dir=None, debug=False, viewer=None*)

Create a running log of your Sage session as a nicely typeset dvi file.

Easy usage: `log_dvi()`

TODO: Pressing "control-D" can mess up the I/O sequence because of a known bug.

Use `L=log_dvi([optional directory])` to create a dvi log. Your complete session so far up until when you type the above command will be logged, along with any future input. Thus you can view the log system as a way to print or view your entire session so far, along with a way to see nicely typeset incremental updates as you work.

If L is a logger, you can type `L.stop()` and `L.start()` to stop and start logging.

The environment variable `DVI_VIEWER` determines which web browser or dvi viewer is used to display your running log. You can also specify a viewer when you start the logger with something like `log_dvi([opt. dir], viewer='xdvi')`.

You must have a LaTeX system installed on your computer and a dvi viewer.

view()

class log_html(*dir=None, debug=False, viewer=None*)
Create a running log of your Sage session as a web page.

Easy usage: `log_html()`

TODO: Pressing "control-D" can mess up the I/O sequence because of a known bug.

Use `L=log_html([optional directory])` to create an HTML log. Your complete session so far up until when you type the above command will be logged, along with any future input. Thus you can view the log system as a way to print or view your entire session so far, along with a way to see nicely typeset incremental updates as you work.

If L is a logger, you can type `L.stop()` and `L.start()` to stop and start logging.

The environment variable `WEB_BROWSER` determines which web browser or dvi viewer is used to display your running log. You can also specify a viewer when you start the logger with something like `log_html([opt. dir], viewer='firefox')`.

You must have a TeX system installed on your computer, and you must have the convert command, which comes with the free ImageMagick tools.

view()

class log_text(*dir=None, debug=False, viewer=None*)
Create a running log of your Sage session as a plain text file.

Easy usage: `log_text()`

TODO: Pressing "control-D" can mess up the I/O sequence because of a known bug.

Use `L=log_text([optional directory])` to create a text log. Your complete session so far up until when you type the above command will be logged, along with any future input. Thus you can view the log system as a way to print or view your entire session so far, along with a way to see incremental updates as you work.

Unlike the html and dvi loggers, this one does not automatically start a viewer unless you specify one; you can do that when you start the logger with something like `log_text([opt. dir], viewer='xterm -e tail -f')`.

If L is a logger, you can type `L.stop()` and `L.start()` to stop and start logging.

view()

update()

11.13 Object persistence

You can load and save most Sage object to disk using the load and save member functions and commands.

Note: It is impossible to save certain Sage objects to disk. For example, if x is a MAGMA object, i.e., a wrapper around an object that is defined in MAGMA, there is no way to save x it to disk, since MAGMA doesn't support saving of individual objects to disk.

- Versions: Loading and saving of objects is guaranteed to work even if the version of Python changes. Saved objects can be loaded in future versions of Python. However, if the data structure that defines the object, e.g., in Sage code, changes drastically (or changes name or disappears), then the object might not load correctly or work correctly.
- Objects are zlib compressed for space efficiency.

db(*name*)
Load object with given name from the Sage database. Use x.db(name) or db_save(x, name) to save objects to the database.

The database directory is `$HOME/.sage/db`.

db_save(*x, name=None*)
Save x to the Sage database.

The database directory is `$HOME/.sage/db`.

load_sage_element(*cls, parent, dic_pic*)

load_sage_object(*cls, dic*)

11.14 Support for persistent functions in .sage files.

Persistent functions are functions whose values are stored on disk so they do not have to be recomputed.

The inputs to the function must be hashable (so lists are not allowed). Though a hash is used, in the incredibly unlikely event that a hash collision occurs, your function will not return an incorrect result because of this (though the cache might not be used either).

This is meant to be used from `.sage` files, not from library `.py` files.

To use this disk caching mechanism, just put `@func_persist` right before your function definition. For example,

```
@func_persist
def bern(n):
    "Return the n-th Bernoulli number, caching the result to disk."
    return bernoulli(n)
```

You can then use the function `bern` as usual, except it will almost instantly return values that have already been computed, even if you quit and restart.

The disk cache files are stored by default in the subdirectory `func_persist` of the current working directory, with one file for each evaluation of the function.

class func_persist(*f, dir='func_persist'*)
Put `@func_persist` right before your function definition to cache values it computes to disk.

11.15 Evaluating a String in Sage

sage_eval(*source, locals=None, cmds='', preparse=True*)
Obtain a Sage object from the input string by evaluating it using Sage. This means calling eval after preparsing and with globals equal to everything included in the scope of `from sage.all import *`.).

INPUT:

- `source` - a string or object with a _sage_ method
- `locals` - evaluate in namespace of sage.all plus the locals dictionary
- `cmds` - string; sequence of commands to be run before source is evaluated.
- `preparse` - (default: True) if True, preparse the string expression.

EXAMPLES: This example illustrates that preparsing is applied.

```
sage: eval('2^3')
1
sage: sage_eval('2^3')
8
```

However, preparsing can be turned off.

```
sage: sage_eval('2^3', preparse=False)
1
```

Note that you can explicitly define variables and pass them as the second option:

```
sage: x = PolynomialRing(RationalField(),"x").gen()
sage: sage_eval('x^2+1', locals={'x':x})
x^2 + 1
```

This example illustrates that evaluation occurs in the context of `from sage.all import *`. Even though bernoulli has been redefined in the local scope, when calling `sage_eval` the default value meaning of bernoulli is used. Likewise for QQ below.

```
sage: bernoulli = lambda x : x^2
sage: bernoulli(6)
36
sage: eval('bernoulli(6)')
36
sage: sage_eval('bernoulli(6)')
1/42
```

```
sage: QQ = lambda x : x^2
sage: QQ(2)
4
sage: sage_eval('QQ(2)')
2
sage: parent(sage_eval('QQ(2)'))
Rational Field
```

This example illustrates setting a variable for use in evaluation.

```
sage: x = 5
sage: eval('4/3 + x', {'x':25})
26
sage: sage_eval('4/3 + x',  locals={'x':25})
79/3
```

You can also specify a sequence of commands to be run before the expression is evaluated:

```
sage: sage_eval('p', cmds='K.<x> = QQ[]\np = x^2 + 1')
x^2 + 1
```

If you give commands to execute and a dictionary of variables, then the dictionary will be modified by assignments in the commands:

```
sage: vars = {}
sage: sage_eval('None', cmds='y = 3', locals=vars)
sage: vars['y'], parent(vars['y'])
(3, Integer Ring)
```

You can also specify the object to evaluate as a tuple. A 2-tuple is assumed to be a pair of a command sequence and an expression; a 3-tuple is assumed to be a triple of a command sequence, an expression, and a dictionary holding local variables. (In this case, the given dictionary will not be modified by assignments in the commands.)

```
sage: sage_eval(('f(x) = x^2', 'f(3)'))
9
sage: vars = {'rt2': sqrt(2.0)}
sage: sage_eval(('rt2 += 1', 'rt2', vars))
2.41421356237309
sage: vars['rt2']
1.41421356237310
```

This example illustrates how `sage_eval` can be useful when evaluating the output of other computer algebra systems.

```
sage: R.<x> = PolynomialRing(RationalField())
sage: gap.eval('R:=PolynomialRing(Rationals,["x"]);')
'PolynomialRing(..., [ x ])'
sage: ff = gap.eval('x:=IndeterminatesOfPolynomialRing(R);; f:=x^2+1;'); ff
'x^2+1'
sage: sage_eval(ff, locals={'x':x})
x^2 + 1
sage: eval(ff)
...
RuntimeError: Use ** for exponentiation, not '^', which means xor
in Python, and has the wrong precedence.
```

Here you can see eval simply will not work but `sage_eval` will.

TESTS:

We get a nice minimal error message for syntax errors, that still points to the location of the error (in the input string):

```
sage: sage_eval('RR(22/7]')
...
 File "<string>", line 1
    RR(Integer(22)/Integer(7)]
                             ^
SyntaxError: unexpected EOF while parsing


sage: sage_eval('None', cmds='$x = $y[3] # Does Perl syntax work?')
...
 File "<string>", line 1
    $x = $y[Integer(3)] # Does Perl syntax work?
    ^
SyntaxError: invalid syntax
```

sageobj(*x, vars=None*)

Return a native Sage object associated to x, if possible and implemented.

If the object has an _sage_ method it is called and the value is returned. Otherwise str is called on the object, and all preparsing is applied and the resulting expression is evaluated in the context of `from sage.all import *`. To evaluate the expression with certain variables set, use the vars argument, which should be a dictionary.

EXAMPLES:

```
sage: type(sageobj(gp('34/56')))
<type 'sage.rings.rational.Rational'>
sage: n = 5/2
sage: sageobj(n) is n
True
sage: k = sageobj('Z(8^3/1)', {'Z':ZZ}); k
512
sage: type(k)
<type 'sage.rings.integer.Integer'>
```

This illustrates interfaces:

```
sage: f = gp('2/3')
sage: type(f)
<class 'sage.interfaces.gp.GpElement'>
sage: f._sage_()
2/3
sage: type(f._sage_())
<type 'sage.rings.rational.Rational'>
sage: a = gap(939393/2433)
sage: a._sage_()
313131/811
sage: type(a._sage_())
<type 'sage.rings.rational.Rational'>
```

11.16 Random testing.

Some Sage modules do random testing in their doctests; that is, they construct test cases using a random number generator. To get the broadest possible test coverage, we want everybody who runs the doctests to use a different random seed; but we also want to be able to reproduce the problems when debugging. This module provides a decorator to help write random testers that meet these goals.

random_testing(*fn*)

This decorator helps create random testers. These can be run as part of the standard Sage test suite; everybody who runs the test will use a different random number seed, so many different random tests will eventually be run.

INPUT:

- `fn` - The function that we are wrapping for random testing.

The resulting function will take two additional arguments, *seed* (default `None`) and *print_seed* (default `False`). The result will set the random number seed to the given seed value (or to a truly random value, if *seed* is not specified), then call the original function. If *print_seed* is true, then the seed will be printed before calling the original function. If the original function raises an exception, then the random seed that was used will be displayed, along with a message entreating the user to submit a bug report. All other arguments will be passed through to the original function.

Here is a set of recommendations for using this wrapper.

The function to be tested should take arguments specifying the difficulty of the test (size of the test cases, number of iterations, etc.), as well as an argument *verbose* (defaulting to false). With *verbose* true, it should print the values being tested. Suppose `test_foo()` takes an argument for number of iterations. Then the doctests could be:

```
test_foo(2, verbose=True, seed=0)
test_foo(10)
test_foo(100) # long time
```

The first doctest, with the specified seed and `verbose=True`, simply verifies that the tests really are reproducible (that `test_foo` is correctly using the `randstate` framework). The next two tests use truly random seeds, and will print out the seed used if the test fails (raises an exception).

If you want a very long-running test using this setup, you should do something like:

```
for _ in xrange(10^10): test_foo(100)
```

instead of:

```
test_foo(10^12)
```

If the test fails after several hours, the latter snippet would make you rerun the test for several hours while reproducing and debugging the problem. With the former snippet, you only need to rerun `test_foo(100)` with a known-failing random seed.

See `sage.misc.random_testing.test_add_commutes()` for a simple example using this decorator, and `sage.rings.tests` for realistic uses.

Setting *print_seed* to true is useless in doctests, because the random seed printed will never match the expected doctest result (and using `# random` means the doctest framework will never report an error even if one happens). However, it is useful if you have a random test that sometimes segfaults. The normal print-the-random-seed-on-exceptions won't work then, so you can run:

```
while True: test_foo(print_seed=True)
```

and look at the last seed that was printed before it crashed.

TESTS:

```
sage: from sage.misc.random_testing import random_testing
sage: def foo(verbose=False):
...       'oh look, a docstring'
...       n = ZZ.random_element(2^50)
...       if verbose:
...           print "Random value: %s" % n
...       assert(n == 49681376900427)
sage: foo = random_testing(foo)
sage: foo(seed=0, verbose=True)
Random value: 49681376900427
sage: foo(seed=15, verbose=True)
Random value: 1049538412064764
Random testing has revealed a problem in foo
Please report this bug!  You may be the first
person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 15
AssertionError()
sage: foo() # random
Random testing has revealed a problem in foo
Please report this bug!  You may be the first
person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 272500700755151445506092479579811710040
AssertionError()
```

```
sage: foo.__doc__
'oh look, a docstring'
sage: foo.__name__
'foo'
sage: def bar(): pass
sage: bar = random_testing(bar)
sage: bar(print_seed=True) # random
Random seed: 26284109189015634692353976554381414605l
```

test_add_commutes(**args, **kwargs*)

This is a simple demonstration of the `random_testing()` decorator and its recommended usage.

We test that addition is commutative over rationals.

EXAMPLES:

```
sage: from sage.misc.random_testing import test_add_commutes
sage: test_add_commutes(2, verbose=True, seed=0)
a == -4, b == 0 ...
Passes!
a == -1/2, b == -1/95 ...
Passes!
sage: test_add_commutes(10)
sage: test_add_commutes(1000) # long time
```

test_add_is_mul(**args, **kwargs*)

This example demonstrates a failing `random_testing()` test, and shows how to reproduce the error.

DO NOT USE THIS AS AN EXAMPLE OF HOW TO USE `random_testing()`! Instead, look at `sage.misc.random_testing.test_add_commutes()`.

We test that `a+b == a*b`, for *a*, *b* rational. This is of course false, so the test will almost always fail.

EXAMPLES:

```
sage: from sage.misc.random_testing import test_add_is_mul
```

We start by testing that we get reproducible results when setting *seed* to 0.

```
sage: test_add_is_mul(2, verbose=True, seed=0)
a == -4, b == 0 ...
Random testing has revealed a problem in test_add_is_mul
Please report this bug!  You may be the first
person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 0
AssertionError()
```

Normally in a `@random_testing` doctest, we would leave off the `verbose=True` and the `# random`. We put it in here so that we can verify that we are seeing the exact same error when we reproduce the error below.

```
sage: test_add_is_mul(10, verbose=True) # random
a == -2/7, b == 1 ...
Random testing has revealed a problem in test_add_is_mul
Please report this bug!  You may be the first
person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 216390410596009428782506007128692114173
AssertionError()
```

OK, now assume that some user has reported a `test_add_is_mul()` failure. We can specify the same *random_seed* that was found in the bug report, and we will get the exact same failure so that we can debug the "problem".

```
sage: test_add_is_mul(10, verbose=True, seed=216390410596009428782506007128692114173)
a == -2/7, b == 1 ...
Random testing has revealed a problem in test_add_is_mul
Please report this bug!  You may be the first
person in the world to have seen this problem.
Please include this random seed in your bug report:
Random seed: 216390410596009428782506007128692114173
AssertionError()
```

11.17 Miscellaneous arithmetic functions

CRT (*a, b, m, n*)

Use the Chinese Remainder Theorem to find some integer x such that x=a (mod m) and x=b (mod n). Note that x is only well-defined modulo m*n.

EXAMPLES:

```
sage: crt(2, 1, 3, 5)
-4
sage: crt(13,20,100,301)
-2087
```

You can also use upper case:

```
sage: c = CRT(2,3, 3, 5); c
8
sage: c % 3 == 2
True
sage: c % 5 == 3
True
```

CRT_basis (*moduli*)

Return a list of integers a[i] such that CRT to the given moduli of numbers x[0],...,x[n-1] is a[0]*x0 + ... + a[n-1]*x[n-1].

INPUT:

- `list` - list of integers

CRT_list (*v, moduli*)

Given a list v of integers and a list of corresponding moduli, find a single integer that reduces to each element of v modulo the corresponding moduli.

EXAMPLES:

```
sage: CRT_list([2,3,2], [3,5,7])
23
```

CRT_vectors (*X, moduli*)

INPUT:

- `X` - list of lists of the same length
- `moduli` - list of len(X) moduli

OUTPUT:

- `list` - application of CRT componentwise.

class Euler_Phi()

Return the value of the Euler phi function on the integer n. We defined this to be the number of positive integers <= n that are relatively prime to n. Thus if n<=0 then `euler_phi(n)` is defined and equals 0.

INPUT:

- `n` - an integer

EXAMPLES:

```
sage: euler_phi(1)
1
sage: euler_phi(2)
1
sage: euler_phi(3)
2
sage: euler_phi(12)
4
sage: euler_phi(37)
36
```

Notice that euler_phi is defined to be 0 on negative numbers and 0.

```
sage: euler_phi(-1)
0
sage: euler_phi(0)
0
sage: type(euler_phi(0))
<type 'sage.rings.integer.Integer'>
```

We verify directly that the phi function is correct for 21.

```
sage: euler_phi(21)
12
sage: [i for i in range(21) if gcd(21,i) == 1]
[1, 2, 4, 5, 8, 10, 11, 13, 16, 17, 19, 20]
```

The length of the list of integers 'i' in range(n) such that the gcd(i,n) == 1 equals euler_phi(n).

```
sage: len([i for i in range(21) if gcd(21,i) == 1]) == euler_phi(21)
True
```

The phi function also has a special plotting method.

```
sage: P = plot(euler_phi, -3, 71)
```

AUTHORS:

- William Stein
- Alex Clemesha (2006-01-10): some examples

plot(*xmin=1, xmax=50, pointsize=30, rgbcolor=(0, 0, 1), join=True, **kwds*)

Plot the Euler phi function.

INPUT:

- `xmin` - default: 1

- `xmax` - default: 50
- `pointsize` - default: 30
- `rgbcolor` - default: (0,0,1)
- `join` - default: True; whether to join the points.
- `**kwds` - passed on

GCD(*a, b=None, **kwargs*)

The greatest common divisor of a and b, or if a is a list and b is omitted the greatest common divisor of all elements of a.

INPUT:

- `a,b` - two elements of a ring with gcd or
- `a` - a list or tuple of elements of a ring with gcd

Additional keyword arguments are passed to the respectively called methods.

EXAMPLES:

```
sage: GCD(97,100)
1
sage: GCD(97*10^15, 19^20*97^2)
97
sage: GCD(2/3, 4/3)
1
sage: GCD([2,4,6,8])
2
sage: GCD(srange(0,10000,10))  # fast  !!
10
```

Note that to take the gcd of n elements for $n \neq 2$ you must put the elements into a list by enclosing them in `[..]`. Before #4988 the following wrongly returned 3 since the third parameter was just ignored:

```
sage: gcd(3,6,2)
...
TypeError: gcd() takes at most 2 arguments (3 given)
sage: gcd([3,6,2])
1
```

Similarly, giving just one element (which is not a list) gives an error:

```
sage: gcd(3)
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
```

By convention, the gcd of the empty list is (the integer) 0:

```
sage: gcd([])
0
sage: type(gcd([]))
<type 'sage.rings.integer.Integer'>
```

LCM(*a, b=None*)

The least common multiple of a and b, or if a is a list and b is omitted the least common multiple of all elements of a.

Note that LCM is an alias for lcm.

INPUT:

- a, b - two elements of a ring with lcm or
- a - a list or tuple of elements of a ring with lcm

EXAMPLES:

```
sage: lcm(97,100)
9700
sage: LCM(97,100)
9700
sage: LCM(0,2)
0
sage: LCM(-3,-5)
15
sage: LCM([1,2,3,4,5])
60
sage: v = LCM(range(1,10000))    # *very* fast!
sage: len(str(v))
4349
```

class Moebius()

Returns the value of the Moebius function of abs(n), where n is an integer.

DEFINITION: $\mu(n)$ is 0 if n is not square free, and otherwise equals $(-1)^r$, where n has r distinct prime factors.

For simplicity, if $n = 0$ we define $\mu(n) = 0$.

IMPLEMENTATION: Factors or - for integers - uses the PARI C library.

INPUT:

- n - anything that can be factored.

OUTPUT: 0, 1, or -1

EXAMPLES:

```
sage: moebius(-5)
-1
sage: moebius(9)
0
sage: moebius(12)
0
sage: moebius(-35)
1
sage: moebius(-1)
1
sage: moebius(7)
-1


sage: moebius(0)    # potentially nonstandard!
0
```

The moebius function even makes sense for non-integer inputs.

```
sage: x = GF(7)['x'].0
sage: moebius(x+2)
-1
```

plot(*xmin=0, xmax=50, pointsize=30, rgbcolor=(0, 0, 1), join=True, **kwds*)

Plot the Moebius function.

INPUT:

- xmin - default: 0
- xmax - default: 50
- pointsize - default: 30
- rgbcolor - default: (0,0,1)
- join - default: True; whether to join the points (very helpful in seeing their order).
- **kwds - passed on

range(*start, stop=None, step=None*)

Return the Moebius function evaluated at the given range of values, i.e., the image of the list range(start, stop, step) under the Mobius function.

This is much faster than directly computing all these values with a list comprehension.

EXAMPLES:

```
sage: v = moebius.range(-10,10); v
[1, 0, 0, -1, 1, -1, 0, -1, -1, 1, 0, 1, -1, -1, 0, -1, 1, -1, 0, 0]
sage: v == [moebius(n) for n in range(-10,10)]
True
sage: v = moebius.range(-1000, 2000, 4)
sage: v == [moebius(n) for n in range(-1000,2000, 4)]
True
```

class Sigma()

Return the sum of the k-th powers of the divisors of n.

INPUT:

- n - integer
- k - integer (default: 1)

OUTPUT: integer

EXAMPLES:

```
sage: sigma(5)
6
sage: sigma(5,2)
26
```

The sigma function also has a special plotting method.

```
sage: P = plot(sigma, 1, 100)
```

This method also works with k-th powers.

```
sage: P = plot(sigma, 1, 100, k=2)
```

AUTHORS:

- William Stein: original implementation
- Craig Citro (2007-06-01): rewrote for huge speedup

TESTS:

```
sage: sigma(100,4)
106811523
sage: sigma(factorial(100),3).mod(144169)
3672
sage: sigma(factorial(150),12).mod(691)
176
```

```
sage: RR(sigma(factorial(133),20))
2.80414775675747e4523
sage: sigma(factorial(100),0)
39001250856960000
sage: sigma(factorial(41),1)
2291995322730299887677338587007329065117587079168 00
```

plot (*xmin=1, xmax=50, k=1, pointsize=30, rgbcolor=(0, 0, 1), join=True, **kwds*)

Plot the sigma (sum of k-th powers of divisors) function.

INPUT:

- `xmin` - default: 1
- `xmax` - default: 50
- `k` - default: 1
- `pointsize` - default: 30
- `rgbcolor` - default: (0,0,1)
- `join` - default: True; whether to join the points.
- `**kwds` - passed on

XGCD (*a, b*)

Returns triple (g,s,t) such that $g = s * a + t * b = gcd(a, b)$.

INPUT:

- `a, b` - integers or univariate polynomials (or any type with an xgcd method).

OUTPUT:

- `g, s, t` - such that g = s*a + t*b

Note: There is no guarantee that the returned cofactors (s and t) are minimal. In the integer case, see Integer._xgcd() for minimal cofactors.

EXAMPLES:

```
sage: xgcd(56, 44)
(4, 4, -5)
sage: 4*56 + (-5)*44
4
sage: g, a, b = xgcd(5/1, 7/1); g, a, b
(1, 3, -2)
sage: a*(5/1) + b*(7/1) == g
True
sage: x = polygen(QQ)
sage: xgcd(x^3 - 1, x^2 - 1)
(x - 1, 1, -x)
sage: K.<g> = NumberField(x^2-3)
sage: R.<a,b> = K[]
sage: S.<y> = R.fraction_field()[]
sage: xgcd(y^2, a*y+b)
(b^2/a^2, 1, ((-1)/a)*y + b/a^2)
sage: xgcd((b+g)*y^2, (a+g)*y+b)
((b^3 + (g)*b^2)/(a^2 + (2*g)*a + 3), 1, ((-b + (-g))/(a + (g)))*y + (b^2 + (g)*b)/(a^2 + (2*g)*
```

algdep (*z, n, known_bits=None, use_bits=None, known_digits=None, use_digits=None*)

Returns a polynomial of degree at most n which is approximately satisfied by the number z. Note that the returned polynomial need not be irreducible, and indeed usually won't be if z is a good approximation to an algebraic number of degree less than n.

You can specify the number of known bits or digits with known_bits=k or known_digits=k; Pari is then told to compute the result using .8*k of these bits/digits. (The Pari documentation recommends using a factor between .6 and .9, but internally defaults to .8.) Or, you can specify the precision to use directly with use_bits=k or use_digits=k. If none of these are specified, then the precision is taken from the input value.

ALGORITHM: Uses the PARI C-library algdep command.

INPUT:

- z - real, complex, or p-adic number
- n - an integer

EXAMPLES:

```
sage: algdep(1.888888888888888, 1)
9*x - 17
sage: algdep(0.12121212121212,1)
33*x - 4
sage: algdep(sqrt(2),2)
x^2 - 2
```

This example involves a complex number.

```
sage: z = (1/2)*(1 + RDF(sqrt(3)) *CC.0); z
0.500000000000000 + 0.866025403784439*I
sage: p = algdep(z, 6); p
x^5 + x^2
sage: p.factor()
(x + 1) * x^2 * (x^2 - x + 1)
sage: z^2 - z + 1
1.11022302462516e-16
```

This example involves a p-adic number.

```
sage: K = Qp(3, print_mode = 'series')
sage: a = K(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12 + 2*3^15 + 2*3^16 + 3^17 + 2*3^1
sage: algdep(a, 1)
19*x - 7
```

These examples show the importance of proper precision control. We compute a 200-bit approximation to sqrt(2) which is wrong in the 33'rd bit.

```
sage: z = sqrt(RealField(200)(2)) + (1/2)^33
sage: p = algdep(z, 4); p
177858662573*x^4 + 59566570004*x^3 - 221308611561*x^2 - 84791308378*x - 317384111411
sage: factor(p)
177858662573*x^4 + 59566570004*x^3 - 221308611561*x^2 - 84791308378*x - 317384111411
sage: algdep(z, 4, known_bits=32)
x^2 - 2
sage: algdep(z, 4, known_digits=10)
x^2 - 2
sage: algdep(z, 4, use_bits=25)
x^2 - 2
sage: algdep(z, 4, use_digits=8)
x^2 - 2
```

algebraic_dependency(*z, n, known_bits=None, use_bits=None, known_digits=None, use_digits=None*)

Returns a polynomial of degree at most n which is approximately satisfied by the number z. Note that the

returned polynomial need not be irreducible, and indeed usually won't be if z is a good approximation to an algebraic number of degree less than n.

You can specify the number of known bits or digits with known_bits=k or known_digits=k; Pari is then told to compute the result using .8*k of these bits/digits. (The Pari documentation recommends using a factor between .6 and .9, but internally defaults to .8.) Or, you can specify the precision to use directly with use_bits=k or use_digits=k. If none of these are specified, then the precision is taken from the input value.

ALGORITHM: Uses the PARI C-library algdep command.

INPUT:

- z - real, complex, or p-adic number
- n - an integer

EXAMPLES:

```
sage: algdep(1.888888888888888, 1)
9*x - 17
sage: algdep(0.12121212121212,1)
33*x - 4
sage: algdep(sqrt(2),2)
x^2 - 2
```

This example involves a complex number.

```
sage: z = (1/2)*(1 + RDF(sqrt(3)) *CC.0); z
0.500000000000000 + 0.866025403784439*I
sage: p = algdep(z, 6); p
x^5 + x^2
sage: p.factor()
(x + 1) * x^2 * (x^2 - x + 1)
sage: z^2 - z + 1
1.11022302462516e-16
```

This example involves a p-adic number.

```
sage: K = Qp(3, print_mode = 'series')
sage: a = K(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12 + 2*3^15 + 2*3^16 + 3^17 + 2*3^1
sage: algdep(a, 1)
19*x - 7
```

These examples show the importance of proper precision control. We compute a 200-bit approximation to sqrt(2) which is wrong in the 33'rd bit.

```
sage: z = sqrt(RealField(200)(2)) + (1/2)^33
sage: p = algdep(z, 4); p
177858662573*x^4 + 59566570004*x^3 - 221308611561*x^2 - 84791308378*x - 317384111411
sage: factor(p)
177858662573*x^4 + 59566570004*x^3 - 221308611561*x^2 - 84791308378*x - 317384111411
sage: algdep(z, 4, known_bits=32)
x^2 - 2
sage: algdep(z, 4, known_digits=10)
x^2 - 2
sage: algdep(z, 4, use_bits=25)
x^2 - 2
sage: algdep(z, 4, use_digits=8)
x^2 - 2
```

bernoulli(*n, algorithm='default', num_threads=1*)

Return the n-th Bernoulli number, as a rational number.

INPUT:

- `n` - an integer
- `algorithm`:
  - `'default'` - (default) use 'pari' for n <= 30000, and 'bernmm' for n > 30000 (this is just a heuristic, and not guaranteed to be optimal on all hardware)
  - `'pari'` - use the PARI C library
  - `'gap'` - use GAP
  - `'gp'` - use PARI/GP interpreter
  - `'magma'` - use MAGMA (optional)
  - `'bernmm'` - use bernmm package (a multimodular algorithm)
- `num_threads` - positive integer, number of threads to use (only used for bernmm algorithm)

EXAMPLES:

```
sage: bernoulli(12)
-691/2730
sage: bernoulli(50)
495057205241079648212477525/66
```

We demonstrate each of the alternative algorithms:

```
sage: bernoulli(12, algorithm='gap')
-691/2730
sage: bernoulli(12, algorithm='gp')
-691/2730
sage: bernoulli(12, algorithm='magma')           # optional - magma
-691/2730
sage: bernoulli(12, algorithm='pari')
-691/2730
sage: bernoulli(12, algorithm='bernmm')
-691/2730
sage: bernoulli(12, algorithm='bernmm', num_threads=4)
-691/2730
```

TESTS:

```
sage: algs = ['gap','gp','pari','bernmm'] #long time
sage: vals = [[bernoulli(i,algorithm = j) for j in algs] for i in range(2,2255)] #long time
sage: union([len(union(x))==1 for x in vals]) #long time
[True]
sage: algs = ['gp','pari','bernmm'] #long time
sage: vals = [[bernoulli(i,algorithm = j) for j in algs] for i in range(2256,5000)] #long time
sage: union([len(union(x))==1 for x in vals]) #long time
[True]
```

Note: If $n > 50000$ then algorithm = 'gp' is used instead of algorithm = 'pari', since the C-library interface to PARI is limited in memory for individual operations.

AUTHOR:

- David Joyner and William Stein

binomial(*x*, *m*)

Return the binomial coefficient

$$x(x-1)\cdots(x-m+1)/m!$$

which is defined for $m \in \mathbf{Z}$ and any x. We extend this definition to include cases when $x - m$ is an integer but m is not by

binomial(x,m)= binomial(x,x-m)

If $m < 0$ return 0.

INPUT:

- `x, m` - numbers or symbolic expressions. Either m or x-m must be an integer.

OUTPUT: number or symbolic expression (if input is symbolic)

EXAMPLES:

```
sage: binomial(5,2)
10
sage: binomial(2,0)
1
sage: binomial(1/2, 0)
1
sage: binomial(3,-1)
0
sage: binomial(20,10)
184756
sage: binomial(-2, 5)
-6
sage: binomial(RealField()('2.5'), 2)
1.87500000000000
sage: n=var('n'); binomial(n,2)
1/2*(n - 1)*n
sage: n=var('n'); binomial(n,n)
1
sage: n=var('n'); binomial(n,n-1)
n
sage: binomial(2^100, 2^100)
1

sage: k, i = var('k,i',ns=1)
sage: binomial(k,i)
binomial(k,i)
```

TESTS:

We test that certain binomials are very fast (this should be instant) – see trac 3309:

```
sage: a = binomial(RR(1140000.78), 42000000)
```

binomial_coefficients(*n*)

Return a dictionary containing pairs $\{(k_1, k_2) : C_{k,n}\}$ where C_{k_n} are binomial coefficients and $n = k_1 + k_2$.

INPUT:

- `n` - an integer

OUTPUT: dict

EXAMPLES:

```
sage: sorted(binomial_coefficients(3).items())
[((0, 3), 1), ((1, 2), 3), ((2, 1), 3), ((3, 0), 1)]
```

Notice the coefficients above are the same as below:

```
sage: R.<x,y> = QQ[]
sage: (x+y)^3
x^3 + 3*x^2*y + 3*x*y^2 + y^3
```

AUTHORS:

- Fredrik Johansson

continuant(*v, n=None*)

Function returns the continuant of the sequence v (list or tuple).

Definition: see Graham, Knuth and Patashnik: Concrete Mathematics: 6.7 Continuants:

$K_0() = 1$

$K_1(x_1) = x_1$

$K_n(x_1, \cdots, x_n) = K_{n-1}(x_n, \cdots x_{n-1})x_n + K_{n-2}(x_1, \cdots, x_{n-2})$

If $n = None$ or $n > len(v)$ the default $n = len(v)$ is used.

INPUT:

- v - list or tuple of elements of a ring
- n - optional integer

OUTPUT: element of ring (integer, polynomial, etcetera).

EXAMPLES:

```
sage: continuant([1,2,3])
10
sage: p = continuant([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10])
sage: q = continuant([1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10])
sage: p/q
517656/190435
sage: convergent([2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10],14)
517656/190435
sage: x = PolynomialRing(RationalField(),'x',5).gens()
sage: continuant(x)
x0*x1*x2*x3*x4 + x0*x1*x2 + x0*x1*x4 + x0*x3*x4 + x2*x3*x4 + x0 + x2 + x4
sage: continuant(x, 3)
x0*x1*x2 + x0 + x2
sage: continuant(x,2)
x0*x1 + 1
```

$\binom{K_n(z,z,\cdots,z)=sum_{k=0}^{n} n-k}{kz^{n-2k}}$:

```
sage: z = QQ['z'].0
sage: continuant((z,z,z,z,z,z,z,z,z,z,z,z,z,z,z,z),6)
z^6 + 5*z^4 + 6*z^2 + 1
sage: continuant(9)
...
TypeError: object of type 'sage.rings.integer.Integer' has no len()
```

AUTHORS:

- Jaap Spies (2007-02-06)

continued_fraction_list(*x, partial_convergents=False, bits=None*)

Returns the continued fraction of x as a list.

This may be slow since it's implemented in pure Python for real input. For rational number input the PARI C library is used.

EXAMPLES:

```
sage: continued_fraction_list(45/17)
[2, 1, 1, 1, 5]
sage: continued_fraction_list(e, bits=20)
[2, 1, 2, 1, 1, 4, 1, 1, 6]
sage: continued_fraction_list(e, bits=30)
[2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1]
sage: continued_fraction_list(sqrt(2))
[1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1]
sage: continued_fraction_list(sqrt(4/19))
[0, 2, 5, 1, 1, 2, 1, 16, 1, 2, 1, 1, 5, 4, 5, 1, 1, 2, 1, 15, 2]
sage: continued_fraction_list(RR(pi), partial_convergents=True)
([3, 7, 15, 1, 292, 1, 1, 1, 2, 1, 3, 1, 14, 3],
 [(3, 1),
  (22, 7),
  (333, 106),
  (355, 113),
  (103993, 33102),
  (104348, 33215),
  (208341, 66317),
  (312689, 99532),
  (833719, 265381),
  (1146408, 364913),
  (4272943, 1360120),
  (5419351, 1725033),
  (80143857, 25510582),
  (245850922, 78256779)])
sage: continued_fraction_list(e)
[2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1, 11]
sage: continued_fraction_list(RR(e))
[2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1, 11]
sage: print continued_fraction_list(RealField(200)(e))
[2, 1, 2, 1, 1, 4, 1, 1, 6, 1, 1, 8, 1, 1, 10, 1, 1, 12, 1, 1, 14, 1, 1, 16, 1, 1, 18, 1, 1, 20,
```

convergent(*v, n*)

Return the n-th continued fraction convergent of the continued fraction defined by the sequence of integers v. We assume $n \geq 0$.

INPUT:

- `v` - list of integers
- `n` - integer

OUTPUT: a rational number

If the continued fraction integers are

$$v = [a_0, a_1, a_2, \ldots, a_k]$$

then `convergent(v,2)` is the rational number

$$a_0 + 1/a_1$$

and `convergent(v,k)` is the rational number

$$a1 + 1/(a2 + 1/(...)...)$$

represented by the continued fraction.

EXAMPLES:

```
sage: convergent([2, 1, 2, 1, 1, 4, 1, 1], 7)
193/71
```

convergents (*v*)

Return all the partial convergents of a continued fraction defined by the sequence of integers v.

If v is not a list, compute the continued fraction of v and return its convergents (this is potentially much faster than calling continued_fraction first, since continued fractions are implemented using PARI and there is overhead moving the answer back from PARI).

INPUT:

- `v` - list of integers or a rational number

OUTPUT:

- `list` - of partial convergents, as rational numbers

EXAMPLES:

```
sage: convergents([2, 1, 2, 1, 1, 4, 1, 1])
[2, 3, 8/3, 11/4, 19/7, 87/32, 106/39, 193/71]
```

crt (*a, b, m, n*)

Use the Chinese Remainder Theorem to find some integer x such that x=a (mod m) and x=b (mod n). Note that x is only well-defined modulo m*n.

EXAMPLES:

```
sage: crt(2, 1, 3, 5)
-4
sage: crt(13,20,100,301)
-2087
```

You can also use upper case:

```
sage: c = CRT(2,3, 3, 5); c
8
sage: c % 3 == 2
True
sage: c % 5 == 3
True
```

differences (*lis, n=1*)

Returns the n successive differences of the elements in lis.

EXAMPLES:

```
sage: differences(prime_range(50))
[1, 2, 2, 4, 2, 4, 2, 4, 6, 2, 6, 4, 2, 4]
sage: differences([i^2 for i in range(1,11)])
[3, 5, 7, 9, 11, 13, 15, 17, 19]
sage: differences([i^3 + 3*i for i in range(1,21)])
[10, 22, 40, 64, 94, 130, 172, 220, 274, 334, 400, 472, 550, 634, 724, 820, 922, 1030, 1144]
```

```
sage: differences([i^3 - i^2 for i in range(1,21)], 2)
[10, 16, 22, 28, 34, 40, 46, 52, 58, 64, 70, 76, 82, 88, 94, 100, 106, 112]
sage: differences([p - i^2 for i, p in enumerate(prime_range(50))], 3)
[-1, 2, -4, 4, -4, 4, 0, -6, 8, -6, 0, 4]
```

AUTHORS:

- Timothy Clemans (2008-03-09)

divisors(*n*)

Returns a list of all positive integer divisors of the nonzero integer n.

INPUT:

- n - the element

EXAMPLES:

```
sage: divisors(-3)
[1, 3]
sage: divisors(6)
[1, 2, 3, 6]
sage: divisors(28)
[1, 2, 4, 7, 14, 28]
sage: divisors(2^5)
[1, 2, 4, 8, 16, 32]
sage: divisors(100)
[1, 2, 4, 5, 10, 20, 25, 50, 100]
sage: divisors(1)
[1]
sage: divisors(0)
...
ValueError: n must be nonzero
sage: divisors(2^3 * 3^2 * 17)
[1, 2, 3, 4, 6, 8, 9, 12, 17, 18, 24, 34, 36, 51, 68, 72, 102, 136, 153, 204, 306, 408, 612, 122
```

This function works whenever one has unique factorization:

```
sage: K.<a> = QuadraticField(7)
sage: divisors(K.ideal(7))
[Fractional ideal (1), Fractional ideal (-a), Fractional ideal (7)]
sage: divisors(K.ideal(3))
[Fractional ideal (1), Fractional ideal (3), Fractional ideal (a - 2), Fractional ideal (-a - 2)
sage: divisors(K.ideal(35))
[Fractional ideal (1), Fractional ideal (35), Fractional ideal (-5*a), Fractional ideal (5), Fra
```

TESTS:

```
sage: divisors(int(300))
[1, 2, 3, 4, 5, 6, 10, 12, 15, 20, 25, 30, 50, 60, 75, 100, 150, 300]
```

eratosthenes(*n*)

Return a list of the primes $\leq n$.

This is extremely slow and is for educational purposes only.

factor(*n, proof=None, int_=False, algorithm='pari', verbose=0, **kwds*)

Returns the factorization of n. The result depends on the type of n.

If n is an integer, factor returns the factorization of the integer n as an object of type Factorization.

If n is not an integer, `n.factor(proof=proof, **kwds)` gets called. See `n.factor??` for more documentation in this case.

Warning: This means that applying factor to an integer result of a symbolic computation will not factor the integer, because it is considered as an element of a larger symbolic ring.
EXAMPLE:

```
sage: f(n)=n^2
sage: is_prime(f(3))
False
sage: factor(f(3))
9
```

INPUT:

- `n` - an nonzero integer
- `proof` - bool or None (default: None)
- `int_` - bool (default: False) whether to return answers as Python ints
- `algorithm` - string
  - `'pari'` - (default) use the PARI c library
  - `'kash'` - use KASH computer algebra system (requires the optional kash package be installed)
  - `'magma'` - use Magma (requires magma be installed)
- `verbose` - integer (default 0); pari's debug variable is set to this; e.g., set to 4 or 8 to see lots of output during factorization.

OUTPUT: factorization of n

The qsieve and ecm commands give access to highly optimized implementations of algorithms for doing certain integer factorization problems. These implementations are not used by the generic factor command, which currently just calls PARI (note that PARI also implements sieve and ecm algorithms, but they aren't as optimized). Thus you might consider using them instead for certain numbers.

The factorization returned is an element of the class Factorization; see Factorization?? for more details, and examples below for usage. A Factorization contains both the unit factor (+1 or -1) and a sorted list of (prime, exponent) pairs.

The factorization displays in pretty-print format but it is easy to obtain access to the (prime,exponent) pairs and the unit, to recover the number from its factorization, and even to multiply two factorizations. See examples below.

EXAMPLES:

```
sage: factor(500)
2^2 * 5^3
sage: factor(-20)
-1 * 2^2 * 5
sage: f=factor(-20)
sage: list(f)
[(2, 2), (5, 1)]
sage: f.unit()
-1
sage: f.value()
-20
```

```
sage: factor(-500, algorithm='kash')       # optional - kash
-1 * 2^2 * 5^3
```

```
sage: factor(-500, algorithm='magma')      # optional - magma
-1 * 2^2 * 5^3

sage: factor(0)
...
ArithmeticError: Prime factorization of 0 not defined.
sage: factor(1)
1
sage: factor(-1)
-1
sage: factor(2^(2^7)+1)
59649589127497217 * 5704689200685129054721
```

Sage calls PARI's factor, which has proof False by default. Sage has a global proof flag, set to True by default (see sage.structure.proof, or proof.[tab]). To override the default, call this function with proof=False.

```
sage: factor(3^89-1, proof=False)
2 * 179 * 1611479891519807 * 504293943956596049162197

sage: factor(2^197 + 1)       # takes a long time (e.g., 3 seconds!)
3 * 197002597249 * 1348959352853811313 * 251951573867253012259144010843
```

To access the data in a factorization:

```
sage: f = factor(420); f
2^2 * 3 * 5 * 7
sage: [x for x in f]
[(2, 2), (3, 1), (5, 1), (7, 1)]
sage: [p for p,e in f]
[2, 3, 5, 7]
sage: [e for p,e in f]
[2, 1, 1, 1]
sage: [p^e for p,e in f]
[4, 3, 5, 7]
```

factorial(*n, algorithm='gmp'*)

Compute the factorial of n, which is the product $1 \cdot 2 \cdot 3 \cdots (n-1)n$.

INPUT:

- `n` - an integer
- `algorithm` - string (default: 'gmp')
  - `'gmp'` - use the GMP C-library factorial function
  - `'pari'` - use PARI's factorial function

OUTPUT: an integer

EXAMPLES:

```
sage: factorial(0)
1
sage: factorial(4)
24
sage: factorial(10)
3628800
sage: factorial(1) == factorial(0)
True
```

```
sage: factorial(6) == 6*5*4*3*2
True
sage: factorial(1) == factorial(0)
True
sage: factorial(71) == 71* factorial(70)
True
sage: factorial(-32)
...
ValueError: factorial -- must be nonnegative
```

PERFORMANCE: This discussion is valid as of April 2006. All timings below are on a Pentium Core Duo 2Ghz MacBook Pro running Linux with a 2.6.16.1 kernel.

- It takes less than a minute to compute the factorial of 10^7 using the GMP algorithm, and the factorial of 10^6 takes less than 4 seconds.
- The GMP algorithm is faster and more memory efficient than the PARI algorithm. E.g., PARI computes 10^7 factorial in 100 seconds on the core duo 2Ghz.
- For comparison, computation in Magma $\leq$ 2.12-10 of $n!$ is best done using `*[1..n]`. It takes 113 seconds to compute the factorial of 10^7 and 6 seconds to compute the factorial of 10^6. Mathematica V5.2 compute the factorial of 10^7 in 136 seconds and the factorial of 10^6 in 7 seconds. (Mathematica is notably very efficient at memory usage when doing factorial calculations.)

falling_factorial(*x*, *a*)

Returns the falling factorial $(x)_a$.

The notation in the literature is a mess: often $(x)_a$, but there are many other notations: GKP: Concrete Mathematics uses $x^{\underline{a}}$.

Definition: for integer $a \geq 0$ we have $x(x-1)\cdots(x-a+1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+1)}{\Gamma(x-a+1)}$.

INPUT:

- `x` - element of a ring
- `a` - a non-negative integer or

OR

- `x and a` - any numbers

OUTPUT: the falling factorial

EXAMPLES:

```
sage: falling_factorial(10, 3)
720
sage: falling_factorial(10, RR('3.0'))
720.000000000000
sage: falling_factorial(10, RR('3.3'))
1310.11633396601
sage: falling_factorial(10, 10)
3628800
sage: factorial(10)
3628800
sage: a = falling_factorial(1+I, I); a
gamma(I + 2)
sage: CC(a)
0.652965496420167 + 0.343065839816545*I
sage: falling_factorial(1+I, 4)
```

```
4*I + 2
sage: falling_factorial(I, 4)
-10

sage: M = MatrixSpace(ZZ, 4, 4)
sage: A = M([1,0,1,0,1,0,1,0,1,0,10,10,1,0,1,1])
sage: falling_factorial(A, 2) # A(A - I)
[  1   0  10  10]
[  1   0  10  10]
[ 20   0 101 100]
[  2   0  11  10]

sage: x = ZZ['x'].0
sage: falling_factorial(x, 4)
x^4 - 6*x^3 + 11*x^2 - 6*x
```

AUTHORS:

- Jaap Spies (2006-03-05)

farey (*v, lim*)

Return the Farey sequence associated to the floating point number v.

INPUT:

- `v` - float (automatically converted to a float)
- `lim` - maximum denominator.

OUTPUT: Results are (numerator, denominator); (1, 0) is"infinity".

AUTHORS:

- Scott David Daniels: Python Cookbook, 2nd Ed., Recipe 18.13

fundamental_discriminant (*D*)

Return the discriminant of the quadratic extension $K = Q(\sqrt{D})$, i.e. an integer d congruent to either 0 or 1, mod 4, and such that, at most, the only square dividing it is 4.

gaussian_binomial (*n, k, q=None*)

Return the gaussian binomial

$$\binom{n}{k}_q = \frac{(1-q^n)(1-q^{n-1})\cdots(1-q^{n-k+1})}{(1-q)(1-q^2)\cdots(1-q^k)}.$$

EXAMPLES:

```
sage: gaussian_binomial(5,1)
q^4 + q^3 + q^2 + q + 1
sage: gaussian_binomial(5,1).subs(q=2)
31
sage: gaussian_binomial(5,1,2)
31
```

AUTHORS:

- David Joyner and William Stein

gcd(*a, b=None, **kwargs*)

The greatest common divisor of a and b, or if a is a list and b is omitted the greatest common divisor of all elements of a.

INPUT:

- `a, b` - two elements of a ring with gcd or
- `a` - a list or tuple of elements of a ring with gcd

Additional keyword arguments are passed to the respectively called methods.

EXAMPLES:

```
sage: GCD(97,100)
1
sage: GCD(97*10^15, 19^20*97^2)
97
sage: GCD(2/3, 4/3)
1
sage: GCD([2,4,6,8])
2
sage: GCD(srange(0,10000,10))  # fast  !!
10
```

Note that to take the gcd of n elements for $n \neq 2$ you must put the elements into a list by enclosing them in `[..]`. Before #4988 the following wrongly returned 3 since the third parameter was just ignored:

```
sage: gcd(3,6,2)
...
TypeError: gcd() takes at most 2 arguments (3 given)
sage: gcd([3,6,2])
1
```

Similarly, giving just one element (which is not a list) gives an error:

```
sage: gcd(3)
...
TypeError: 'sage.rings.integer.Integer' object is not iterable
```

By convention, the gcd of the empty list is (the integer) 0:

```
sage: gcd([])
0
sage: type(gcd([]))
<type 'sage.rings.integer.Integer'>
```

get_gcd(*order*)

Return the fastest gcd function for integers of size no larger than order.

EXAMPLES:

```
sage: sage.rings.arith.get_gcd(4000)
<built-in method gcd_int of sage.rings.fast_arith.arith_int object at ...>
sage: sage.rings.arith.get_gcd(400000)
<built-in method gcd_longlong of sage.rings.fast_arith.arith_llong object at ...>
sage: sage.rings.arith.get_gcd(4000000000)
<function gcd at ...>
```

get_inverse_mod(*order*)

Return the fastest inverse_mod function for integers of size no larger than order.

EXAMPLES:

```
sage: sage.rings.arith.get_inverse_mod(6000)
<built-in method inverse_mod_int of sage.rings.fast_arith.arith_int object at ...>
sage: sage.rings.arith.get_inverse_mod(600000)
<built-in method inverse_mod_longlong of sage.rings.fast_arith.arith_llong object at ...>
sage: sage.rings.arith.get_inverse_mod(6000000000)
<function inverse_mod at ...>
```

hilbert_conductor(*a, b*)

This is the product of all (finite) primes where the hilbert symbol is -1. What is the same, this is the (reduced) discriminant of the quaternion algebra (a,b) over Q.

INPUT: a, b – integers

OUTPUT: squarefree positive integer

EXAMPLES: sage: hilbert_conductor(-1, -1) 2 sage: hilbert_conductor(-1, -11) 11 sage: hilbert_conductor(-2, -5) 5 sage: hilbert_conductor(-3, -17) 17

AUTHOR: – Gonzalo Tornaria (2009-03-02)

hilbert_conductor_inverse(*d*)

Finds a pair of integers (a,b) such that hilbert_conductor(a,b) == d. The quaternion algebra (a,b) over Q will have (reduced) discriminant d.

INPUT:

- d – square-free positive integer

OUTPUT: pair of integers

EXAMPLES:

```
sage: hilbert_conductor_inverse(2)
(-1, -1)
sage: hilbert_conductor_inverse(3)
(-1, -3)
sage: hilbert_conductor_inverse(6)
(-1, 3)
sage: hilbert_conductor_inverse(30)
(-3, -10)
sage: hilbert_conductor_inverse(4)
...
ValueError: d needs to be squarefree
sage: hilbert_conductor_inverse(-1)
...
ValueError: d needs to be positive
```

AUTHOR:

- Gonzalo Tornaria (2009-03-02)

TESTS:

```
sage: for i in xrange(100):
...       d = ZZ.random_element(2**32).squarefree_part()
...       if hilbert_conductor(*hilbert_conductor_inverse(d)) != d:
...           print "hilbert_conductor_inverse failed for d =", d
```

hilbert_symbol(*a, b, p, algorithm='pari'*)

Returns 1 if $ax^2 + by^2$ p-adically represents a nonzero square, otherwise returns -1. If either a or b is 0, returns 0.

INPUT:

- `a, b` - integers
- `p` - integer; either prime or -1 (which represents the archimedean place)
- `algorithm` - string
  - `'pari'` - (default) use the PARI C library
  - `'direct'` - use a Python implementation
  - `'all'` - use both PARI and direct and check that the results agree, then return the common answer

OUTPUT: integer (0, -1, or 1)

EXAMPLES:

```
sage: hilbert_symbol (-1, -1, -1, algorithm='all')
-1
sage: hilbert_symbol (2,3, 5, algorithm='all')
1
sage: hilbert_symbol (4, 3, 5, algorithm='all')
1
sage: hilbert_symbol (0, 3, 5, algorithm='all')
0
sage: hilbert_symbol (-1, -1, 2, algorithm='all')
-1
sage: hilbert_symbol (1, -1, 2, algorithm='all')
1
sage: hilbert_symbol (3, -1, 2, algorithm='all')
-1

sage: hilbert_symbol(QQ(-1)/QQ(4), -1, 2) == -1
True
sage: hilbert_symbol(QQ(-1)/QQ(4), -1, 3) == 1
True
```

AUTHORS:

- William Stein and David Kohel (2006-01-05)

integer_ceil(*x*)

Return the ceiling of x.

EXAMPLES:

```
sage: integer_ceil(5.4)
6
```

integer_floor(*x*)

Return the largest integer $\leq x$.

INPUT:

- `x` - an object that has a floor method or is coercible to int

OUTPUT: an Integer

EXAMPLES:

```
sage: integer_floor(5.4)
5
sage: integer_floor(float(5.4))
5
sage: integer_floor(-5/2)
-3
sage: integer_floor(RDF(-5/2))
-3
```

inverse_mod(*a*, *m*)

The inverse of the ring element a modulo m.

If no special inverse_mod is defined for the elements, it tries to coerce them into integers and perform the inversion there

```
sage: inverse_mod(7,1)
0
sage: inverse_mod(5,14)
3
sage: inverse_mod(3,-5)
2
```

is_power_of_two(*n*)

This function returns True if and only if n is a power of 2

INPUT:

- `n` - integer

OUTPUT:

- `True` - if n is a power of 2
- `False` - if not

EXAMPLES:

```
sage: is_power_of_two(1024)
True

sage: is_power_of_two(1)
True

sage: is_power_of_two(24)
False

sage: is_power_of_two(0)
False

sage: is_power_of_two(-4)
False
```

AUTHORS:

- Jaap Spies (2006-12-09)

is_prime(*n, flag=0*)

Returns True if x is prime, and False otherwise. The result is proven correct - *this is NOT a pseudo-primality test!*.

INPUT:

- `flag` - int
  - `0` (default) - use a combination of algorithms.
  - `1` - certify primality using the Pocklington-Lehmer Test.
  - `2` - certify primality using the APRCL test.

OUTPUT:

- `bool` - True or False

Note: We do not consider negatives of prime numbers as prime.

EXAMPLES::

```
sage: is_prime(389)
True
sage: is_prime(2000)
False
sage: is_prime(2)
True
sage: is_prime(-1)
False
sage: factor(-6)
-1 * 2 * 3
sage: is_prime(1)
False
sage: is_prime(-2)
False
```

IMPLEMENTATION: Calls the PARI isprime function.

is_prime_power(*n, flag=0*)

Returns True if x is a prime power, and False otherwise. The result is proven correct - *this is NOT a pseudo-primality test!*.

INPUT:

- `n` - an integer
- `flag (for primality testing)` - int
  - `0` (default): use a combination of algorithms.
  - `1`: certify primality using the Pocklington-Lehmer Test.
  - `2`: certify primality using the APRCL test.

EXAMPLES::

```
sage: is_prime_power(389)
True
sage: is_prime_power(2000)
False
sage: is_prime_power(2)
True
sage: is_prime_power(1024)
True
sage: is_prime_power(-1)
```

```
False
sage: is_prime_power(1)
True
sage: is_prime_power(997^100)
True
```

is_pseudoprime(*n, flag=0*)

Returns True if x is a pseudo-prime, and False otherwise. The result is *NOT* proven correct - *this is a pseudo-primality test!*.

INPUT:

- `flag` - int
  - `0` (default): checks whether x is a Baillie-Pomerance- Selfridge-Wagstaff pseudo prime (strong Rabin-Miller pseudo prime for base 2, followed by strong Lucas test for the sequence (P,-1), P smallest positive integer such that $P^2 - 4$ is not a square mod x).
  - `>0`: checks whether x is a strong Miller-Rabin pseudo prime for flag randomly chosen bases (with end-matching to catch square roots of -1).

OUTPUT:

- `bool` - True or False

Note: We do not consider negatives of prime numbers as prime.

EXAMPLES::

```
sage: is_pseudoprime(389)
True
sage: is_pseudoprime(2000)
False
sage: is_pseudoprime(2)
True
sage: is_pseudoprime(-1)
False
sage: factor(-6)
-1 * 2 * 3
sage: is_pseudoprime(1)
False
sage: is_pseudoprime(-2)
False
```

IMPLEMENTATION: Calls the PARI ispseudoprime function.

is_square(*n, root=False*)

Returns whether or not n is square, and if n is a square also returns the square root. If n is not square, also returns None.

INPUT:

- `n` - an integer
- `root` - whether or not to also return a square root (default: False)

OUTPUT:

- `bool` - whether or not a square
- `object` - (optional) an actual square if found, and None otherwise.

EXAMPLES:

```
sage: is_square(2)
False
sage: is_square(4)
True
sage: is_square(2.2)
True
sage: is_square(-2.2)
False
sage: is_square(CDF(-2.2))
True
sage: is_square((x-1)^2)
True

sage: is_square(4, True)
(True, 2)
```

is_squarefree(*n*)
Returns True if and only if n is not divisible by the square of an integer > 1.

kronecker(*x, y*)
Synonym for `kronecker_symbol()`.

kronecker_symbol(*x, y*)
The Kronecker symbol $(x|y)$.

INPUT:

- `x` - integer
- `y` - integer

EXAMPLES:

```
sage: kronecker(3,5)
-1
sage: kronecker(3,15)
0
sage: kronecker(2,15)
1
sage: kronecker(-2,15)
-1
sage: kronecker(2/3,5)
1
```

IMPLEMENTATION: Using GMP.

lcm(*a, b=None*)
The least common multiple of a and b, or if a is a list and b is omitted the least common multiple of all elements of a.

Note that LCM is an alias for lcm.

INPUT:

- `a,b` - two elements of a ring with lcm or
- `a` - a list or tuple of elements of a ring with lcm

EXAMPLES:

```
sage: lcm(97,100)
9700
sage: LCM(97,100)
9700
sage: LCM(0,2)
0
sage: LCM(-3,-5)
15
sage: LCM([1,2,3,4,5])
60
sage: v = LCM(range(1,10000))    # *very* fast!
sage: len(str(v))
4349
```

legendre_symbol(*x, p*)

The Legendre symbol $(x|p)$, for p prime.

Note: The `kronecker_symbol()` command extends the Legendre symbol to composite moduli and $p = 2$.

INPUT:

- `x` - integer
- `p` - an odd prime number

EXAMPLES:

```
sage: legendre_symbol(2,3)
-1
sage: legendre_symbol(1,3)
1
sage: legendre_symbol(1,2)
...
ValueError: p must be odd
sage: legendre_symbol(2,15)
...
ValueError: p must be a prime
sage: kronecker_symbol(2,15)
1
sage: legendre_symbol(2/3,7)
-1
```

mqrr_rational_reconstruction(*u, m, T*)

Maximal Quotient Rational Reconstruction.

FOR research purposes only - this is pure Python, so slow.

INPUT:

- `u, m, T` - integers such that $m > u \geq 0, T > 0$.

OUTPUT:

Either integers n, d such that $d > 0$, $gcd(n, d) = 1$, $n/d = u(modm)$, and $T * d * |n| < m$, or `None`.

Reference: Monagan, Maximal Quotient Rational Reconstruction: An Almost Optimal Algorithm for Rational Reconstruction (page 11)

This algorithm is probabilistic.

multinomial(*ks)

Return the multinomial coefficient

$$\binom{k_1 + \cdots + k_n}{k_1, \cdots, k_n} = \frac{(\sum_{i=1}^n k_i)!}{\prod_{i=1}^n k_i!} = \prod_{i=1}^n \binom{\sum_{j=1}^i k_j}{k_i}$$

EXAMPLES:

```
sage: multinomial(0, 0, 2, 1, 0, 0)
3
sage: multinomial(3, 2)
10
sage: multinomial(2^30, 2, 1)
618970023101454657175683075
```

AUTHORS:

- Gabriel Ebner

multinomial_coefficients(*m, n, _tuple=<type 'tuple'>, _zip=<built-in function zip>*)

Return a dictionary containing pairs $\{(k_1, k_2, ..., k_m) : C_{k,n}\}$ where $C_{k,n}$ are multinomial coefficients such that $n = k_1 + k_2 + ... + k_m$.

INPUT:

- `m` - integer
- `n` - integer
- `_tuple, _zip` - hacks for speed; don't set these as a user.

OUTPUT: dict

EXAMPLES:

```
sage: sorted(multinomial_coefficients(2,5).items())
[((0, 5), 1), ((1, 4), 5), ((2, 3), 10), ((3, 2), 10), ((4, 1), 5), ((5, 0), 1)]
```

Notice that these are the coefficients of $(x + y)^5$:

```
sage: R.<x,y> = QQ[]
sage: (x+y)^5
x^5 + 5*x^4*y + 10*x^3*y^2 + 10*x^2*y^3 + 5*x*y^4 + y^5
```

```
sage: sorted(multinomial_coefficients(3,2).items())
[((0, 0, 2), 1), ((0, 1, 1), 2), ((0, 2, 0), 1), ((1, 0, 1), 2), ((1, 1, 0), 2), ((2, 0, 0), 1)]
```

ALGORITHM: The algorithm we implement for computing the multinomial coefficients is based on the following result:

Consider a polynomial and its n-th exponent:

$$P(x) = \sum_{i=0}^{m} p_i x^k$$

$$P(x)^n = \sum_{k=0}^{mn} a(n,k) x^k$$

We compute the coefficients $a(n,k)$ using the J.C.P. Miller Pure Recurrence [see D.E.Knuth, Seminumerical Algorithms, The art of Computer Programming v.2, Addison Wesley, Reading, 1981].

$$a(n,k) = 1/(kp_0)\sum_{i=1}^{m} p_i((n+1)i-k)a(n,k-i),$$

where $a(n,0) = p_0^n$.

AUTHORS:

- Pearu Peterson

next_prime(*n, proof=None*)

The next prime greater than the integer n. If n is prime, then this function does not return n, but the next prime after n. If the optional argument proof is False, this function only returns a pseudo-prime, as defined by the PARI nextprime function. If it is None, uses the global default (see sage.structure.proof)

INPUT:

- `n` - integer
- `proof` - bool or None (default: None)

EXAMPLES:

```
sage: next_prime(-100)
2
sage: next_prime(1)
2
sage: next_prime(2)
3
sage: next_prime(3)
5
sage: next_prime(4)
5
```

Notice that the next_prime(5) is not 5 but 7.

```
sage: next_prime(5)
7
sage: next_prime(2004)
2011
```

next_prime_power(*n*)

The next prime power greater than the integer n. If n is a prime power, then this function does not return n, but the next prime power after n.

EXAMPLES:

```
sage: next_prime_power(-10)
1
sage: is_prime_power(1)
True
sage: next_prime_power(0)
1
sage: next_prime_power(1)
2
sage: next_prime_power(2)
3
sage: next_prime_power(10)
11
```

```
sage: next_prime_power(7)
8
sage: next_prime_power(99)
101
```

next_probable_prime(*n*)

Returns the next probable prime after self, as determined by PARI.

INPUT:

- n - an integer

EXAMPLES:

```
sage: next_probable_prime(-100)
2
sage: next_probable_prime(19)
23
sage: next_probable_prime(int(999999999))
1000000007
sage: next_probable_prime(2^768)
155251809230070893514897948846250255525688601711669661113905203802605095268637688633087840882864
```

nth_prime(*n*)

EXAMPLES:

```
sage: nth_prime(3)
5
sage: nth_prime(10)
29


sage: nth_prime(0)
...
ValueError: nth prime meaningless for non-positive n (=0)
```

number_of_divisors(*n*)

Return the number of divisors of the integer n.

odd_part(*n*)

The odd part of the integer n. This is $n/2^v$, where $v = \text{valuation}(n, 2)$.

EXAMPLES:

```
sage: odd_part(5)
5
sage: odd_part(4)
1
sage: odd_part(factorial(31))
122529844256906551386796875
```

power_mod(*a, n, m*)

The n-th power of a modulo the integer m.

EXAMPLES:

```
sage: power_mod(0,0,5)
...
ArithmeticError: 0^0 is undefined.
sage: power_mod(2,390,391)
```

```
285
sage: power_mod(2,-1,7)
4
sage: power_mod(11,1,7)
4
sage: R.<x> = ZZ[]
sage: power_mod(3*x, 10, 7)
4*x^10

sage: power_mod(11,1,0)
...
ZeroDivisionError: modulus must be nonzero.
```

previous_prime(*n*)

The largest prime < n. The result is provably correct. If n <= 1, this function raises a ValueError.

EXAMPLES:

```
sage: previous_prime(10)
7
sage: previous_prime(7)
5
sage: previous_prime(8)
7
sage: previous_prime(7)
5
sage: previous_prime(5)
3
sage: previous_prime(3)
2
sage: previous_prime(2)
...
ValueError: no previous prime
sage: previous_prime(1)
...
ValueError: no previous prime
sage: previous_prime(-20)
...
ValueError: no previous prime
```

previous_prime_power(*n*)

The largest prime power $< n$. The result is provably correct. If $n \leq 2$, this function returns $-x$, where x is prime power and $-x < n$ and no larger negative of a prime power has this property.

EXAMPLES:

```
sage: previous_prime_power(2)
1
sage: previous_prime_power(10)
9
sage: previous_prime_power(7)
5
sage: previous_prime_power(127)
125

sage: previous_prime_power(0)
...
ValueError: no previous prime power
```

```
sage: previous_prime_power(1)
...
ValueError: no previous prime power

sage: n = previous_prime_power(2^16 - 1)
sage: while is_prime(n):
...     n = previous_prime_power(n)
sage: factor(n)
251^2
```

prime_divisors(*n*)

The prime divisors of the integer n, sorted in increasing order. If n is negative, we do *not* include -1 among the prime divisors, since -1 is not a prime number.

EXAMPLES:

```
sage: prime_divisors(1)
[]
sage: prime_divisors(100)
[2, 5]
sage: prime_divisors(-100)
[2, 5]
sage: prime_divisors(2004)
[2, 3, 167]
```

prime_factors(*n*)

The prime divisors of the integer n, sorted in increasing order. If n is negative, we do *not* include -1 among the prime divisors, since -1 is not a prime number.

EXAMPLES:

```
sage: prime_divisors(1)
[]
sage: prime_divisors(100)
[2, 5]
sage: prime_divisors(-100)
[2, 5]
sage: prime_divisors(2004)
[2, 3, 167]
```

prime_powers(*start, stop=None*)

List of all positive primes powers between start and stop-1, inclusive. If the second argument is omitted, returns the primes up to the first argument.

EXAMPLES:

```
sage: prime_powers(20)
[1, 2, 3, 4, 5, 7, 8, 9, 11, 13, 16, 17, 19]
sage: len(prime_powers(1000))
194
sage: len(prime_range(1000))
168
sage: a = [z for z in range(95,1234) if is_prime_power(z)]
sage: b = prime_powers(95,1234)
sage: len(b)
194
sage: len(a)
194
```

```
sage: a[:10]
[97, 101, 103, 107, 109, 113, 121, 125, 127, 128]
sage: b[:10]
[97, 101, 103, 107, 109, 113, 121, 125, 127, 128]
sage: a == b
True
```

TESTS:

```
sage: v = prime_powers(10)
sage: type(v[0])      # trac #922
<type 'sage.rings.integer.Integer'>
```

prime_to_m_part(*n, m*)

Returns the prime-to-m part of n, i.e., the largest divisor of n that is coprime to m.

INPUT:

- n - Integer (nonzero)
- m - Integer

OUTPUT: Integer

EXAMPLES:

```
sage: z = 43434
sage: z.prime_to_m_part(20)
21717
```

primes(*start, stop=None*)

Returns an iterator over all primes between start and stop-1, inclusive. This is much slower than prime_range, but potentially uses less memory.

This command is like the xrange command, except it only iterates over primes. In some cases it is better to use primes than prime_range, because primes does not build a list of all primes in the range in memory all at once. However it is potentially much slower since it simply calls the next_prime function repeatedly, and next_prime is slow, partly because it proves correctness.

EXAMPLES:

```
sage: for p in primes(5,10):
...     print p
...
5
7
sage: list(primes(11))
[2, 3, 5, 7]
sage: list(primes(10000000000, 10000000100))
[10000000019, 10000000033, 10000000061, 10000000069, 10000000097]
```

primes_first_n(*n, leave_pari=False*)

Return the first n primes.

INPUT:

- n - a nonnegative integer

OUTPUT:

- a list of the first n prime numbers.

EXAMPLES:

```
sage: primes_first_n(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: len(primes_first_n(1000))
1000
sage: primes_first_n(0)
[]
```

primitive_root(*n*)

Return a generator for the multiplicative group of integers modulo n, if one exists.

EXAMPLES:

```
sage: primitive_root(23)
5
sage: print [primitive_root(p) for p in primes(100)]
[1, 2, 2, 3, 2, 2, 3, 2, 5, 2, 3, 2, 6, 3, 5, 2, 2, 2, 2, 7, 5, 3, 2, 3, 5]
```

quadratic_residues(*n*)

Return a sorted list of all squares modulo the integer n in the range $0 \leq x < |n|$.

EXAMPLES:

```
sage: quadratic_residues(11)
[0, 1, 3, 4, 5, 9]
sage: quadratic_residues(1)
[0]
sage: quadratic_residues(2)
[0, 1]
sage: quadratic_residues(8)
[0, 1, 4]
sage: quadratic_residues(-10)
[0, 1, 4, 5, 6, 9]
sage: v = quadratic_residues(1000); len(v);
159
```

random_prime(*n, proof=None*)

Returns a random prime p between 2 and n (i.e. 2 <= p <= n). The returned prime is chosen uniformly at random from the set of prime numbers less than or equal to n.

INPUT:

- `n` - an integer >= 2.
- `proof` - bool or None (default: None) If False, the function uses a pseudo-primality test, which is much faster for really big numbers but does not provide a proof of primality. If None, uses the global default (see sage.structure.proof)

EXAMPLES:

```
sage: random_prime(100000)
88237
sage: random_prime(2)
2
```

TESTS:

```
sage: type(random_prime(2))
<type 'sage.rings.integer.Integer'>
sage: type(random_prime(100))
<type 'sage.rings.integer.Integer'>
```

AUTHORS:

- Jon Hanke (2006-08-08): with standard Stein cleanup
- Jonathan Bober (2007-03-17)

rational_reconstruction(*a, m, algorithm='fast'*)

This function tries to compute x/y, where x/y is a rational number in lowest terms such that the reduction of x/y modulo m is equal to a and the absolute values of x and y are both $\leq \sqrt{m/2}$. If such x/y exists, that pair is unique and this function returns it. If no such pair exists, this function raises ZeroDivisionError.

An efficient algorithm for computing rational reconstruction is very similar to the extended Euclidean algorithm. For more details, see Knuth, Vol 2, 3rd ed, pages 656-657.

INPUT:

- `a` - an integer
- `m` - a modulus
- `algorithm` - (default: 'fast')
  - `'fast'` - a fast compiled implementation
  - `'python'` - a slow pure python implementation

OUTPUT:

Numerator and denominator n, d of the unique rational number $r = n/d$, if it exists, with n and $|d| \leq \sqrt{N/2}$. Return $(0, 0)$ if no such number exists.

The algorithm for rational reconstruction is described (with a complete nontrivial proof) on pages 656-657 of Knuth, Vol 2, 3rd ed. as the solution to exercise 51 on page 379. See in particular the conclusion paragraph right in the middle of page 657, which describes the algorithm thus:

> This discussion proves that the problem can be solved efficiently by applying Algorithm 4.5.2X with $u = m$ and $v = a$, but with the following replacement for step X2: If $v3 \leq \sqrt{m/2}$, the algorithm terminates. The pair $(x, y) = (|v2|, v3 * \text{sign}(v2))$ is then the unique solution, provided that x and y are coprime and $x \leq \sqrt{m/2}$; otherwise there is no solution. (Alg 4.5.2X is the extended Euclidean algorithm.)

Knuth remarks that this algorithm is due to Wang, Kornerup, and Gregory from around 1983.

EXAMPLES:

```
sage: m = 100000
sage: (119*inverse_mod(53,m))%m
11323
sage: rational_reconstruction(11323,m)
119/53
```

```
sage: rational_reconstruction(400,1000)
...
ValueError: Rational reconstruction of 400 (mod 1000) does not exist.
```

```
sage: rational_reconstruction(3,292393, algorithm='python')
3
sage: a = Integers(292393)(45/97); a
204977
sage: rational_reconstruction(a,292393, algorithm='python')
45/97
sage: a = Integers(292393)(45/97); a
204977
sage: rational_reconstruction(a,292393, algorithm='fast')
45/97
sage: rational_reconstruction(293048,292393, algorithm='fast')
...
ValueError: Rational reconstruction of 655 (mod 292393) does not exist.
sage: rational_reconstruction(293048,292393, algorithm='python')
...
ValueError: Rational reconstruction of 655 (mod 292393) does not exist.
```

rising_factorial(*x*, *a*)

Returns the rising factorial $(x)^a$.

The notation in the literature is a mess: often $(x)^a$, but there are many other notations: GKP: Concrete Mathematics uses $x^{\overline{a}}$.

The rising factorial is also known as the Pochhammer symbol, see Maple and Mathematica.

Definition: for integer $a \geq 0$ we have $x(x+1)\cdots(x+a-1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+a)}{\Gamma(x)}$.

INPUT:

- `x` - element of a ring
- `a` - a non-negative integer or
- `x and a` - any numbers

OUTPUT: the rising factorial

EXAMPLES:

```
sage: rising_factorial(10,3)
1320

sage: rising_factorial(10,RR('3.0'))
1320.00000000000

sage: rising_factorial(10,RR('3.3'))
2826.38895824964

sage: a = rising_factorial(1+I, I); a
gamma(2*I + 1)/gamma(I + 1)
sage: CC(a)
0.266816390637832 + 0.122783354006372*I

sage: a = rising_factorial(I, 4); a
-10
```

See falling_factorial(I, 4).

```
sage: x = polygen(ZZ)
sage: rising_factorial(x, 4)
x^4 + 6*x^3 + 11*x^2 + 6*x
```

AUTHORS:

- Jaap Spies (2006-03-05)

sort_complex_numbers_for_display(*nums*)

Given a list of complex numbers (or a list of tuples, where the first element of each tuple is a complex number), we sort the list in a "pretty" order. First come the real numbers (with zero imaginary part), then the complex numbers sorted according to their real part. If two complex numbers have a real part which is sufficiently close, then they are sorted according to their imaginary part.

This is not a useful function mathematically (not least because there's no principled way to determine whether the real components should be treated as equal or not). It is called by various polynomial root-finders; its purpose is to make doctest printing more reproducible.

We deliberately choose a cumbersome name for this function to discourage use, since it is mathematically meaningless.

EXAMPLES:

```
sage: import sage.rings.arith
sage: sort_c = sort_complex_numbers_for_display
sage: nums = [CDF(i) for i in range(3)]
sage: for i in range(3):
...       nums.append(CDF(i + RDF.random_element(-3e-11, 3e-11),
...                       RDF.random_element()))
...       nums.append(CDF(i + RDF.random_element(-3e-11, 3e-11),
...                       RDF.random_element()))
sage: shuffle(nums)
sage: sort_c(nums)
[0, 1.0, 2.0, -2.862406201e-11 - 0.708874026302*I, 2.2108362707e-11 - 0.436810529675*I, 1.000000
```

squarefree_divisors(*x*)

Iterator over the squarefree divisors (up to units) of the element x.

Depends on the output of the prime_divisors function.

INPUT:

```
x -- an element of any ring for which the prime_divisors
     function works.
```

EXAMPLES:

```
sage: list(squarefree_divisors(7))
[1, 7]
sage: list(squarefree_divisors(6))
[1, 2, 3, 6]
sage: list(squarefree_divisors(12))
[1, 2, 3, 6]
```

subfactorial(*n*)

Subfactorial or rencontres numbers, or derangements: number of permutations of n elements with no fixed points.

INPUT:

- n - non negative integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: subfactorial(0)
1
sage: subfactorial(1)
0
sage: subfactorial(8)
14833
```

AUTHORS:

- Jaap Spies (2007-01-23)

trial_division(*n, bound=None*)

Return the smallest prime divisor <= bound of the positive integer n, or n if there is no such prime. If the optional argument bound is omitted, then bound <= n.

INPUT:

- n - a positive integer
- bound - (optional) a positive integer

OUTPUT:

- int - a prime p=bound that divides n, or n if there is no such prime.

EXAMPLES:

```
sage: trial_division(15)
3
sage: trial_division(91)
7
sage: trial_division(11)
11
sage: trial_division(387833, 300)
387833
sage: # 300 is not big enough to split off a
sage: # factor, but 400 is.
sage: trial_division(387833, 400)
389
```

two_squares(*n, algorithm='gap'*)

Write the integer n as a sum of two integer squares if possible; otherwise raise a ValueError.

EXAMPLES:

```
sage: two_squares(389)
(10, 17)
sage: two_squares(7)
...
ValueError: 7 is not a sum of two squares
sage: a,b = two_squares(2009); a,b
(28, 35)
sage: a^2 + b^2
2009
```

TODO: Create an implementation using PARI's continued fraction implementation.

valuation(*m, p*)

The exact power of p that divides m.

m should be an integer or rational (but maybe other types work too.)

This actually just calls the m.valuation() method.

If m is 0, this function returns rings.infinity.

EXAMPLES:

```
sage: valuation(512,2)
9
sage: valuation(1,2)
0
sage: valuation(5/9, 3)
-2
```

Valuation of 0 is defined, but valuation with respect to 0 is not:

```
sage: valuation(0,7)
+Infinity
sage: valuation(3,0)
...
ValueError: You can only compute the valuation with respect to a integer larger than 1.
```

Here are some other examples:

```
sage: valuation(100,10)
2
sage: valuation(200,10)
2
sage: valuation(243,3)
5
sage: valuation(243*10007,3)
5
sage: valuation(243*10007,10007)
1
```

xgcd(*a, b*)

Returns triple (g,s,t) such that $g = s * a + t * b = gcd(a, b)$.

INPUT:

- `a, b` - integers or univariate polynomials (or any type with an xgcd method).

OUTPUT:

- `g, s, t` - such that g = s*a + t*b

Note: There is no guarantee that the returned cofactors (s and t) are minimal. In the integer case, see Integer._xgcd() for minimal cofactors.

EXAMPLES:

```
sage: xgcd(56, 44)
(4, 4, -5)
sage: 4*56 + (-5)*44
4
sage: g, a, b = xgcd(5/1, 7/1); g, a, b
(1, 3, -2)
sage: a*(5/1) + b*(7/1) == g
True
```

```
sage: x = polygen(QQ)
sage: xgcd(x^3 - 1, x^2 - 1)
(x - 1, 1, -x)
sage: K.<g> = NumberField(x^2-3)
sage: R.<a,b> = K[]
sage: S.<y> = R.fraction_field()[]
sage: xgcd(y^2, a*y+b)
(b^2/a^2, 1, ((-1)/a)*y + b/a^2)
sage: xgcd((b+g)*y^2, (a+g)*y+b)
((b^3 + (g)*b^2)/(a^2 + (2*g)*a + 3), 1, ((-b + (-g))/(a + (g)))*y + (b^2 + (g)*b)/(a^2 + (2*g)*
```

xlcm(*m, n*)

Extended lcm function: given two positive integers m,n, returns a triple (l,m1,n1) such that l=lcm(m,n)=m1*n1 where m1|m, n1|n and gcd(m1,n1)=1. All with no factorization.

Used to construct an element of order l from elements of orders m,n in any group: see sage/groups/generic.py for examples.

EXAMPLES:

```
sage: xlcm(120,36)
(360, 40, 9)
```

CHAPTER

TWELVE

DATABASES

There are numerous specific mathematical databases either included in Sage or available as optional packages. Also, Sage includes two powerful general database packages.

Sage includes the ZOPE object oriented database ZODB, which "is a Python object persistence system. It provides transparent object-oriented persistency."

Sage also includes the powerful relational database SQLite, along with a Python interface to SQLite. SQlite is a small C library that implements a self-contained, embeddable, zero-configuration SQL database engine.

- Transactions are atomic, consistent, isolated, and durable (ACID) even after system crashes and power failures.
- Zero-configuration - no setup or administration needed.
- Implements most of SQL92. (Features not supported)
- A complete database is stored in a single disk file.
- Database files can be freely shared between machines with different byte orders.
- Supports databases up to 2 tebibytes (2^41 bytes) in size.
- Strings and BLOBs up to 2 gibibytes (2^31 bytes) in size.
- Small code footprint: less than 250KiB fully configured or less than 150KiB with optional features omitted.
- Faster than popular client/server database engines for most common operations.
- Simple, easy to use API.
- TCL bindings included. Bindings for many other languages available separately.
- Well-commented source code with over 95% test coverage.
- Self-contained: no external dependencies.
- Sources are in the public domain. Use for any purpose.

12.1 Cremona's tables of elliptic curves.

Sage includes John Cremona's tables of elliptic curves in an easy-to-use format. The unique instance of the class CremonaDatabase() gives access to the database.

If the full CremonaDatabase isn't installed, a mini-version is included by default with Sage. It contains Weierstrass equations, rank, and torsion for curves up to conductor 10000.

The large database includes all curves of conductor up to 120,000 (!). It also includes data related to the BSD conjecture and modular degrees for all of these curves, and generators for the Mordell-Weil groups. To install it type the following in Sage: !sage -i database_cremona_ellcurve-2005.11.03

The name of the database may change as it is updated. Type "!sage -optional" to see the latest package names.

CremonaDatabase()

class LargeCremonaDatabase(*read_only=True*)

The Cremona database of elliptic curves.

EXAMPLES:

```
sage: c = CremonaDatabase()
sage: c.allcurves(11)
{'a1': [[0, -1, 1, -10, -20], 0, 5], 'a3': [[0, -1, 1, 0, 0], 0, 5], 'a2': [[0, -1, 1, -7820, -2
```

allbsd(*N*)

Return the allbsd table for conductor N. The entries are:

```
[id, tamagawa_product, Omega_E, L, Reg_E, Sha_an(E)],
```

where id is the isogeny class (letter) followed by a number, e.g., b3, and L is $L^r(E,1)/r!$, where E has rank r.

INPUT:

- N - int, the conductor

OUTPUT: dict

allcurves(*N*)

Returns the allcurves table of curves of conductor N.

INPUT:

- N - int, the conductor

OUTPUT:

- dict - id:[ainvs, rank, tor], ...

allgens(*N*)

Return the allgens table for conductor N.

INPUT:

- N - int, the conductor

OUTPUT:

- dict - id:[points, ...], ...

conductor_range()

Return the range of conductors that are covered by the database.

OUTPUT:

- int - smallest cond
- int - largest conductor plus one

EXAMPLES:

```
sage: CremonaDatabase().conductor_range()      # random -- depends on database installed
(1, 10000)
```

curves(*N*)

Returns the curves table of all *optimal* curves of conductor N.

INPUT:

- N - int, the conductor

OUTPUT:

- dict - id:[ainvs, rank, tor], ...

EXAMPLES:

Optimal curves of conductor 37:

```
sage: CremonaDatabase().curves(37)
{'a1': [[0, 0, 1, -1, 0], 1, 1], 'b1': [[0, 1, 1, -23, -50], 0, 3]}
```

Note the 'h3', which is the unique case in the tables where the optimal curve doesn't have label ending in 1:

```
sage: list(sorted(CremonaDatabase().curves(990).keys()))
['a1', 'b1', 'c1', 'd1', 'e1', 'f1', 'g1', 'h3', 'i1', 'j1', 'k1', 'l1']
```

degphi (*N*)

Return the degphi table for conductor N.

INPUT:

- N - int, the conductor

OUTPUT:

- dict - id:degphi, ...

elliptic_curve (*label*)

Return an elliptic curve with given label with some data about it from the database pre-filled in.

INPUT:

- label - str (Cremona label)

OUTPUT: EllipticCurve

elliptic_curve_from_ainvs (*N, ainvs*)

Returns the elliptic curve in the database of conductor N with minimal ainvs, if it exists, or raises a RuntimeError exception otherwise.

INPUT:

- N - int
- ainvs - list (5-tuple of int's); the minimal Weierstrass model for an elliptic curve of conductor N

OUTPUT: EllipticCurve

isogeny_class (*label*)

Returns the isogeny class of elliptic curves that are isogenous to the curve with given Cremona label.

INPUT:

- label - string

OUTPUT:

- list - list of EllipticCurve objects.

isogeny_classes (*conductor*)

Return the allcurves data (ainvariants, rank and torsion) for the elliptic curves in the database of given conductor as a list of lists, one for each isogeny class. The curve with number 1 is always listed first.

iter (*conductors*)

Returns an iterator through all curves with conductor between Nmin and Nmax-1, inclusive, in the database.

INPUT:

- conductors - list or generator of ints

OUTPUT: generator that iterates over EllipticCurve objects.

iter_optimal(*conductors*)

Returns an iterator through all optimal curves with conductor between Nmin and Nmax-1 in the database.

INPUT:

- `conductors` - list or generator of ints

OUTPUT:

generator that iterates over EllipticCurve objects.

EXAMPLES:

We list optimal curves with conductor up to 20:

```
sage: [e.cremona_label() for e in CremonaDatabase().iter_optimal([11..20])]
['11a1', '14a1', '15a1', '17a1', '19a1', '20a1']
```

Note the unfortunate 990h3 special case:

```
sage: [e.cremona_label() for e in CremonaDatabase().iter_optimal([990])]
['990a1', '990b1', '990c1', '990d1', '990e1', '990f1', '990g1', '990h3', '990i1', '990j1', '
```

largest_conductor()

The largest conductor for which the database is complete.

OUTPUT:

- `int` - largest conductor

EXAMPLES:

```
sage: CremonaDatabase().largest_conductor()    # random -- depends on size of installed datab
9999
```

list(*conductors*)

Returns a list of all curves with conductor between Nmin and Nmax-1, inclusive, in the database.

INPUT:

- `conductors` - list or generator of ints

OUTPUT:

- list of EllipticCurve objects.

list_optimal(*conductors*)

Returns a list of all optimal curves with conductor between Nmin and Nmax-1, inclusive, in the database.

INPUT:

- **`conductors` - list or generator of ints** list of EllipticCurve objects.

OUTPUT:

list of EllipticCurve objects.

EXAMPLES:

```
sage: CremonaDatabase().list_optimal([37])
[Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field,
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 23*x - 50 over Rational Field]
```

number_of_curves(*N=0, i=0*)

Returns the number of curves stored in the database with conductor N. If N = 0, returns the total number of curves in the database.

If i is nonzero, returns the number of curves in the i-th isogeny class. If i is a Cremona letter code, e.g., 'a' or 'bc', it is converted to the corresponding number.

INPUT:

- `N` - int
- `i` - int or str

OUTPUT: int

EXAMPLES:

```
sage: c = CremonaDatabase()
sage: c.number_of_curves(11)
3
sage: c.number_of_curves(37)
4
sage: c.number_of_curves(990)
42
sage: num = c.number_of_curves()
```

number_of_isogeny_classes(*N=0*)

Returns the number of isogeny classes of curves in the database of conductor N. If N is 0, return the total number of isogeny classes of curves in the database.

INPUT:

- N - int

OUTPUT: int

EXAMPLES:

```
sage: c = CremonaDatabase()
sage: c.number_of_isogeny_classes(11)
1
sage: c.number_of_isogeny_classes(37)
2
sage: num = c.number_of_isogeny_classes()
```

random()

Returns a random curve from the database.

EXAMPLES:

```
sage: CremonaDatabase().random() # random -- depends on database installed
Elliptic Curve defined by y^2 + x*y  = x^3 - x^2 - 224*x + 3072 over Rational Field
```

smallest_conductor()

The smallest conductor for which the database is complete. (Always 1.)

OUTPUT:

- int - smallest conductor

EXAMPLES:

```
sage: CremonaDatabase().smallest_conductor()
1
```

class MiniCremonaDatabase(*read_only=True*)

A mini version of the Cremona database that contains only the Weierstrass equations, rank and torsion of elliptic curves of conductor up to 10000 and nothing else.

class_to_int(*k*)

Converts class id string into an integer. Note that this is the inverse of cremona_letter_code.

EXAMPLES:

```
sage: import sage.databases.cremona as cremona
sage: cremona.class_to_int('ba')
26
sage: cremona.class_to_int('cremona')
821863562
sage: cremona_letter_code(821863562)
'cremona'
```

cmp_code(*key1, key2*)

Comparison function for curve id strings.

Note: Not the same as standard lexicographic order!

EXAMPLES:

```
sage: import sage.databases.cremona as cremona
sage: cremona.cmp_code('ba1','z1')
1
```

By contrast:

```
sage: cmp('ba1','z1')
-1
```

cremona_letter_code(*n*)

Returns the Cremona letter code corresponding to an integer. For example, 0 - a 25 - z 26 - ba 51 - bz 52 - ca 53 - cb etc.

Note: This is just the base 26 representation of n, where a=0, b=1, ..., z=25. This extends the old Cremona notation (counting from 0) for the first 26 classes, and is different for classes above 26.

INPUT:

- n - int

OUTPUT: str

EXAMPLES:

```
sage: cremona_letter_code(0)
'a'
sage: cremona_letter_code(26)
'ba'
sage: cremona_letter_code(27)
'bb'
sage: cremona_letter_code(521)
'ub'
sage: cremona_letter_code(53)
'cb'
sage: cremona_letter_code(2005)
'czd'
```

is_optimal_id(*id*)

Returns true if the Cremona id refers to an optimal curve, and false otherwise. The curve is optimal if the id, which is of the form [letter code][number] has number 1.

Note: 990h3 is the optimal curve in that class, so doesn't obey this rule.

INPUT:

- id - str of form letter code followed by an integer, e.g., a3, bb5, etc.

OUTPUT: bool

EXAMPLES:

```
False
sage: is_optimal_id('b1')
True
sage: is_optimal_id('bb1')
True
```

```
sage: is_optimal_id('c1')
True
sage: is_optimal_id('c2')
False
```

old_cremona_letter_code(*n*)

Returns the *old* Cremona letter code corresponding to an integer. integer.

For example,

:: 1 –> A 26 –> Z 27 –> AA 52 –> ZZ 53 –> AAA etc.

INPUT:

- n - int

OUTPUT: str

EXAMPLES:

```
sage: old_cremona_letter_code(1)
'A'
sage: old_cremona_letter_code(26)
'Z'
sage: old_cremona_letter_code(27)
'AA'
sage: old_cremona_letter_code(521)
'AAAAAAAAAAAAAAAAAAAAA'
sage: old_cremona_letter_code(53)
'AAA'
sage: old_cremona_letter_code(2005)
'CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC'
```

parse_cremona_label(*label*)

Given a Cremona label that defines an elliptic curve, e.g., 11A1 or 37B3, parse the label and return the conductor, isogeny class label, and number.

The isogeny number may be omitted, in which case it defaults to 1. If the isogeny number and letter are both omitted, so label is just a string representing a conductor, then the label defaults to 'A' and the number to 1.

INPUT:

- label - str

OUTPUT:

- int - the conductor
- str - the isogeny class label
- int - the number

EXAMPLES:

```
sage: parse_cremona_label('37a2')
(37, 'a', 2)
sage: parse_cremona_label('37b1')
(37, 'b', 1)
sage: parse_cremona_label('10bb2')
(10, 'bb', 2)
```

rebuild(*data_tgz, largest_conductor, decompress=True*)

Rebuild the LargeCremonaDatabase from scratch using the data_tgz tarball.

split_code(*key*)
Splits class+curve id string into its two parts.

EXAMPLES:

```
sage: import sage.databases.cremona as cremona
sage: cremona.split_code('ba2')
('ba', '2')
```

12.2 The Stein-Watkins table of elliptic curves.

Sage gives access to the Stein-Watkins table of elliptic curves, via an optional package that you must install. This is a huge database of elliptic curves. You can download the database as a 2.6GB Sage package from http://modular.ucsd.edu/sagedb/, which you install with the command

```
sage -i stein-watkins-ecdb.spkg
```

You can also download a small version, without having to explicitly download anything from a website, using the command

```
sage -i stein-watkins-ecdb-mini
```

This database covers a wide range of conductors, but unlike CremonaDatabase(), this database need not list all curves of a given conductor. It lists the curves whose coefficients aren't "too large" (see [Stein-Watkins, Ants 5]).

- The command `SteinWatkinsAllData(n)` returns an iterator over the curves in the n^{th} Stein-Watkins table, which contains elliptic curves of conductor between $n10^5$ and $(n+1)10^5$. Here n can be between 0 and 999, inclusive.
- The command `SteinWatkinsPrimeData(n)` returns an iterator over the curves in the n^{th} Stein-Watkins table, which contains prime conductor elliptic curves of conductor between $n10^6$ and $(n+1)10^6$. Here n varies between 0 and 99, inclusive.

EXAMPLES: We obtain the first table of elliptic curves.

```
sage: d = SteinWatkinsAllData(0)
sage: d
Stein-Watkins Database a.0 Iterator
```

We type `d.next()` to get each isogeny class of curves from `d`:

```
sage: C = d.next()                                  # optional - stein_watkins_database
sage: C                                             # optional - stein_watkins_database
Stein-Watkins isogeny class of conductor 11
sage: d.next()                                      # optional - stein_watkins_database
Stein-Watkins isogeny class of conductor 14
sage: d.next()                                      # optional - stein_watkins_database
Stein-Watkins isogeny class of conductor 15
```

An isogeny class has a number of attributes that give data about the isogeny class, such as the rank, equations of curves, conductor, leading coefficient of L-function, etc.

```
sage: C.data                                          # optional - stein_watkins_database
['11', '[11]', '0', '0.253842', '25', '+*1']
sage: C.curves                                        # optional - stein_watkins_database
[[[0, -1, 1, 0, 0], '(1)', '1', '5'],
 [[0, -1, 1, -10, -20], '(5)', '1', '5'],
 [[0, -1, 1, -7820, -263580], '(1)', '1', '1']]
sage: C.conductor                                     # optional - stein_watkins_database
11
sage: C.leading_coefficient                           # optional - stein_watkins_database
'0.253842'
sage: C.modular_degree                                # optional - stein_watkins_database
'+*1'
sage: C.rank                                          # optional - stein_watkins_database
0
sage: C.isogeny_number                                # optional - stein_watkins_database
'25'
```

If we were to continue typing `d.next()` we would iterate over all curves in the Stein-Watkins database up to conductor 10^5. We could also type `for C in d: ...`

To access the data file starting at 10^5 do the following:

```
sage: d = SteinWatkinsAllData(1)                      # optional - stein_watkins_database
sage: C = d.next()                                    # optional - stein_watkins_database
sage: C                                               # optional - stein_watkins_database
Stein-Watkins isogeny class of conductor 100002
sage: C.curves                                        # optional - stein_watkins_database
[[[1, 1, 0, 112, 0], '(8,1,2,1)', 'X', '2'],
 [[1, 1, 0, -448, -560], '[4,2,1,2]', 'X', '2']]
```

Next we access the prime-conductor data:

```
sage: d = SteinWatkinsPrimeData(0)                    # optional - stein_watkins_database
sage: C = d.next()                                    # optional - stein_watkins_database
sage: C                                               # optional - stein_watkins_database
Stein-Watkins isogeny class of conductor 11
```

Each call `d.next()` gives another elliptic curve of prime conductor:

```
sage: C = d.next()                                    # optional - stein_watkins_database
sage: C                                               # optional - stein_watkins_database
Stein-Watkins isogeny class of conductor 17
sage: C.curves                                        # optional - stein_watkins_database
[[[1, -1, 1, -1, 0], '[1]', '1', '4'],
 [[1, -1, 1, -6, -4], '[2]', '1', '2x'],
 [[1, -1, 1, -1, -14], '(4)', '1', '4'],
 [[1, -1, 1, -91, -310], '[1]', '1', '2']]
sage: C = d.next()                                    # optional - stein_watkins_database
sage: C                                               # optional - stein_watkins_database
Stein-Watkins isogeny class of conductor 19
```

class **SteinWatkinsAllData**(*num*)

Class for iterating through one of the Stein-Watkins database files for all conductors.

iter_levels()

Iterate through the curve classes, but grouped into lists by level.

next()

class SteinWatkinsIsogenyClass(*conductor*)

class SteinWatkinsPrimeData(*num*)

ecdb_num_curves(*max_level=200000*)

12.3 John Jones's tables of number fields

In order to use the Jones database, the optional database package must be installed using the Sage command !sage -i database_jones_numfield

This is a table of number fields with bounded ramification and degree ≤ 6. You can query the database for all number fields in Jones's tables with bounded ramification and degree.

EXAMPLES: First load the database:

```
sage: J = JonesDatabase()
sage: J
John Jones's table of number fields with bounded ramification and degree <= 6
```

List the degree and discriminant of all fields in the database that have ramification at most at 2:

```
sage: [(k.degree(), k.disc()) for k in J.unramified_outside([2])]     # optional - jones_database
[(1, 1), (2, 8), (2, -4), (2, -8), (4, 2048), (4, -1024), (4, 512), (4, -2048), (4, 256), (4, 2048),
```

List the discriminants of the fields of degree exactly 2 unramified outside 2:

```
sage: [k.disc() for k in J.unramified_outside([2],2)]                 # optional - jones_database

[8, -4, -8]
```

List the discriminants of cubic field in the database ramified exactly at 3 and 5:

```
sage: [k.disc() for k in J.ramified_at([3,5],3)]                      # optional - jones_database

[-6075, -6075, -675, -135]
sage: factor(6075)
3^5 * 5^2
sage: factor(675)
3^3 * 5^2
sage: factor(135)
3^3 * 5
```

List all fields in the database ramified at 101:

```
sage: J.ramified_at(101)                                              # optional - jones_database
[Number Field in a with defining polynomial x^2 - 101, Number Field in a with defining polynomial x^
```

class JonesDatabase()

get(*S, var='a'*)

ramified_at(*S, d=None, var='a'*)

Return all fields in the database of degree d ramified exactly at the primes in S. INPUT:

- S - list or set of primes

- d - None (default) or an integer

EXAMPLES:

```
sage: J = JonesDatabase()               # requires optional package
sage: J.ramified_at([101,119])          # requires optional package
[]
sage: J.ramified_at([119])              # requires optional package
[]
sage: J.ramified_at(101)                # requires optional package
[Number Field in a with defining polynomial x^2 - 101, Number Field in a with defining polyn
```

unramified_outside (*S, d=None*)

Return iterator over fields in the database of degree d unramified outside S. If d is omitted, return fields of any degree up to 6. INPUT:

- S - list or set of primes
- d - None (default) or an integer

EXAMPLES:

```
sage: J = JonesDatabase()              # requires optional package
sage: J.unramified_outside([101,119]) # requires optional package
[Number Field in a with defining polynomial x - 1, Number Field in a with defining polynomia
```

12.4 Linear codes

Linear codes

parse_bound_html (*text, n, k*)

12.5 Interface to Sloane On-Line Encyclopedia of Integer Sequences

To look up sequence A060843, type one of the following:

```
sage: sloane_sequence(60843)       # optional - internet
Searching Sloane's online database...
[60843, 'Busy Beaver problem: maximal number of steps that an n-state Turing machine can make on an
```

```
sage: sloane_sequence("60843")     # optional - internet
Searching Sloane's online database...
[60843, 'Busy Beaver problem: maximal number of steps that an n-state Turing machine can make on an
```

```
sage: sloane_sequence("060843")    # optional - internet
Searching Sloane's online database...
[60843, 'Busy Beaver problem: maximal number of steps that an n-state Turing machine can make on an
```

Do not prefix an integer with a 0 or it will be interpreted in octal. Results are of the form [number, description, list], and invalid numbers will cause sloane_sequence to raise an ValueError exception:

```
sage: sloane_sequence('sage')     # optional - internet
...
ValueError: sequence 'sage' not found
```

To look up the sequence

```
sage: sloane_find([2,3,5,7], 2)       # optional - internet
Searching Sloane's online database...
[[40, 'The prime numbers.', [2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67,
```

To return no more than 2 results (default is 30), type

```
sage: sloane_find([1,2,3,4,5], 2)      # optional - internet
Searching Sloane's online database...
[[27, 'The natural numbers. Also called the whole numbers, the counting numbers or the positive integ
```

Note that the OEIS (http://www.research.att.com/ njas/sequences/) claims to limit the number of results to 100. Results are lists of the form [[number, description, list]], and invalid input will cause sloane_find to return [].

In some cases, these functions may return [] even though the inputs are legal. These cases correspond to errors from the OEIS server, and calling the functions again may fix the problem.

Alternatively, the SloaneEncyclopedia object provides access to a local copy of the database containing only the sequences. To use this you must install the optional database_sloane_oeis-2005-12 package using `sage -i database_sloane_oeis-2005-12`.

To look up a sequence, type

```
sage: SloaneEncyclopedia[60843]               # optional - sloane_database
[1, 6, 21, 107]
```

To search locally for a particular subsequence, type

```
sage: SloaneEncyclopedia.find([1,2,3,4,5], 1)     # optional - sloane_database
[(15, [1, 2, 3, 4, 5, 7, 7, 8, 9, 11, 11, 13, 13, 16, 16, 16, 17, 19, 19, 23, 23, 23, 23, 25, 25, 27,
```

The default maximum number of results is 30, but to return up to 100, type

```
sage: SloaneEncyclopedia.find([1,2,3,4,5], 100)     # optional - sloane_database
[(15, [1, 2, 3, 4, 5, 7, 7, 8, 9, 11, 11, ...
```

Results in either case are of the form [(number, list)].

TODO:

- When this program gets a sloane sequence from the database it actually downloads a huge amount of information about it, then throws most of it away. Also, it returns the data to the user as a very simple tuple. It would be much better to return an instance of a class:

  ```
  class SloaneSequence: ...
  ```

  and the class should have methods for each of the things that Sloane records about a sequence. Also, when possible, it should be able to compute more terms.

AUTHORS:

- Steven Sivek (2005-12-22): first version
- Steven Sivek (2006-02-07): updated to correctly handle the new search form on the Sloane website, and it's now also smarter about loading the local database in that it doesn't convert a sequence from string form to a list of integers until absolutely necessary. This seems to cut the loading time roughly in half.

class SloaneEncyclopediaClass()

A local copy of the Sloane Online Encyclopedia of Integer Sequences that contains only the sequence numbers and the sequences themselves.

find(*seq, maxresults=30*)

Return a list of all sequences which have seq as a subsequence, up to maxresults results. Sequences are returned in the form (number, list).

INPUT:

- `seq` - list
- `maxresults` - int

OUTPUT: list of 2-tuples (i, v), where v is a sequence with seq as a subsequence.

load()

Load the entire encyclopedia into memory from a file. This is done automatically if the user tries to perform a lookup or a search.

unload()

Remove the database from memory.

parse_sequence(*text*)

sloane_find(*list, nresults=30, verbose=True*)

Searches Sloane's Online Encyclopedia of Integer Sequences for a sequence containing the number provided in `list`.

INPUT:

- `list` - (list) a list of integers to search Sloane's for
- `nresults` - (integer) the maximum number of results to return default: 30
- `verbose` - (boolean) print a string to let the user know that it is working and not hanging. default: True

OUTPUT: A list of matches in Sloane's database. Each match consists of a list of the sequence number, the name of the sequence, and some initial terms of the sequence.

EXAMPLES:

```
sage: sloane_find([1,1,2,3,5,8,13,21], nresults=1) #optional - internet
Searching Sloane's online database...
[[45,
  'Fibonacci numbers: F(n) = F(n-1) + F(n-2), F(0) = 0, F(1) = 1, F(2) = 1, ...',
  [0,
   1,
   1,
   2,
   3,
   5,
   8,
   13,
   21,
   ...
   39088169]]]
```

sloane_sequence(*number*)

Returns a list with the number, name, and values for the sequence `number` in Sloane's online database of integer sequences.

EXAMPLES:

```
sage: sloane_sequence(22) # optional - internet
Searching Sloane's online database...
[22,
 'Number of centered hydrocarbons with n atoms.',
 [0,
  1,
  0,
  1,
  ...
  36201693122]]
```

The input must not be a sequence itself:

```
sage: sloane_sequence(prime_range(100))
...
TypeError: input must be an integer or string that specifies the id of the Sloane sequence to do
```

12.6 Frank Luebeck's tables of Conway polynomials over finite fields.

class ConwayPolynomials (*read_only=True*)

degrees (*p*)

has_polynomial (*p, n*)

polynomial (*p, n*)

primes ()

12.7 Tables of zeros of the Riemann-Zeta function.

zeta_zeros ()

List of the imaginary parts of the first 100,000 nontrivial zeros of the Riemann zeta function. Andrew Odlyzko computed these to precision within $3 \cdot 10^{-9}$.

In order to use `zeta_zeros()`, you will need to install the optional Odlyzko database package: `sage -i database_odlyzko_zeta`. You can see a list of all available optional packages with `sage -optional`.

REFERENCES:

- http://www.dtc.umn.edu/~odlyzko/zeta_tables/

EXAMPLES:

The following example prints the imaginary part of the 13th nontrivial zero of the Riemann zeta function. Note that only the first 9 digits after the decimal come from the database. Subsequent digits are the result of the inherent imprecision of a binary representation of decimal numbers.

```
sage: zz = zeta_zeros() # optional
sage: zz[12]            # optional
59.347044003000001
```

CHAPTER

THIRTEEN

INTERPRETER INTERFACES

Sage provides a unified interface to the best computational software. This is accomplished using both C-libraries (see *C/C++ Library Interfaces*) and interpreter interfaces, which are implemented using pseudo-tty's, system files, etc. This chapter is about these interpreter interfaces.

Note: Each interface requires that the corresponding software is installed on your computer. Sage includes GAP, PARI, Singular, and Maxima, but does not include Octave (very easy to install), MAGMA (non-free), Maple (non-free), or Mathematica (non-free).

There is overhead associated with each call to one of these systems. For example, computing `2+2` thousands of times using the GAP interface will be slower than doing it directly in Sage. In contrast, the C-library interfaces of *C/C++ Library Interfaces* incur less overhead.

In addition to the commands described for each of the interfaces below, you can also type e.g., `%gap`, `%magma`, etc., to directly interact with a given interface in its state. Alternatively, if `X` is an interface object, typing `X.interact()` allows you to interact with it. This is completely different than `X.console()` which starts a complete new copy of whatever program `X` interacts with. Note that the input for `X.interact()` is handled by Sage, so the history buffer is the same as for Sage, tab completion is as for Sage (unfortunately!), and input that spans multiple lines must be indicated using a backslash at the end of each line. You can pull data into an interactive session with `X` using `sage(expression)`.

The console and interact methods of an interface do very different things. For example, using gap as an example:

1. `gap.console()`: You are completely using another program, e.g., gap/magma/gp Here Sage is serving as nothing more than a convenient program launcher, similar to bash.
2. `gap.interact()`: This is a convenient way to interact with a running gap instance that may be "full of" Sage objects. You can import Sage objects into this gap (even from the interactive interface), etc.

The console function is very useful on occasion, since you get the exact actual program available (especially useful for tab completion and testing to make sure nothing funny is going on).

13.1 Common Interface Functionality

See the examples in the other sections for how to use specific interfaces. The interface classes all derive from the generic interface that is described in this section.

AUTHORS:

- William Stein (2005): initial version
- William Stein (2006-03-01): got rid of infinite loop on startup if client system missing

class **AsciiArtString**(*x*)

class **Expect**(*name, prompt, command=None, server=None, server_tmpdir=None, ulimit=None, maxread=100000, script_subdirectory='', restart_on_ctrlc=False, verbose_start=False, init_code=, [], max_startup_time=30, logfile=None, eval_using_file_cutoff=0, do_cleaner=True, remote_cleaner=False, path=None*)

Expect interface object.

call(*function_name, *args, **kwds*)

clear(*var*)

Clear the variable named var.

clear_prompts()

console()

cputime()

CPU time since this process started running.

eval(*code, strip=True, synchronize=False, locals=None, **kwds*)

INPUT:

- `code` - text to evaluate
- `strip` - bool; whether to strip output prompts, etc. (ignored in the base class).
- `locals` - None (ignored); this is used for compatibility with the Sage notebook's generic system interface.
- `**kwds` - All other arguments are passed onto the _eval_line method. An often useful example is reformat=False.

execute(*args, **kwds*)

expect()

function_call(*function, args=None, kwds=None*)

EXAMPLES:

```
sage: maxima.quad_qags(x, x, 0, 1, epsrel=1e-4)
[0.5,5.5511151231257...e-15,21,0]
sage: maxima.function_call('quad_qags', [x, x, 0, 1], {'epsrel':'1e-4'})
[0.5,5.5511151231257...e-15,21,0]
```

get(*var*)

Get the value of the variable var.

get_using_file(*var*)

Return the string representation of the variable var in self, possibly using a file. Use this if var has a huge string representation, since it may be way faster.

Warning: In fact unless a special derived class implements this, it will *not* be any faster. This is the case for this class if you're reading it through introspection and seeing this.

help(*s*)

interact()

This allows you to interactively interact with the child interpreter. Press Ctrl-D or type 'quit' or 'exit' to exit and return to Sage.

Note: This is completely different than the console() member function. The console function opens a new copy of the child interepreter, whereas the interact function gives you interactive access to the interpreter that is being used by Sage. Use sage(xxx) or interpretername(xxx) to pull objects in from sage to the interpreter.

interrupt(*tries=20, timeout=0.29999999999999999, quit_on_fail=True*)

is_local()

is_remote()

is_running()
Return True if self is currently running.

name(*new_name=None*)

new(*code*)

path()

pid()

quit(*verbose=False, timeout=0.25*)
EXAMPLES:

```
sage: a = maxima('y')
sage: maxima.quit()
sage: a._check_valid()
...
ValueError: The maxima session in which this object was defined is no longer running.
```

read(*filename*)
EXAMPLES:

```
sage: filename = tmp_filename()
sage: f = open(filename, 'w')
sage: f.write('x = 2\n')
sage: f.close()
sage: octave.read(filename)  #optional -- requires Octave
sage: octave.get('x')        #optional
' 2'
sage: import os
sage: os.unlink(filename)
```

set(*var, value*)
Set the variable var to the given value.

user_dir()

class ExpectElement(*parent, value, is_name=False, name=None*)
Expect element.

attribute(*attrname*)
If this wraps the object x in the system, this returns the object x.attrname. This is useful for some systems that have object oriented attribute access notation.
EXAMPLES:

```
sage: g = gap('SO(1,4,7)')
sage: k = g.InvariantQuadraticForm()
sage: k.attribute('matrix')
[ [ 0*Z(7), Z(7)^0, 0*Z(7), 0*Z(7) ], [ 0*Z(7), 0*Z(7), 0*Z(7), 0*Z(7) ],
  [ 0*Z(7), 0*Z(7), Z(7), 0*Z(7) ], [ 0*Z(7), 0*Z(7), 0*Z(7), Z(7)^0 ] ]

sage: e = gp('ellinit([0,-1,1,-10,-20])')
sage: e.attribute('j')
-122023936/161051
```

bool()

gen(*n*)

get_using_file()
Return this element's string representation using a file. Use this if self has a huge string representation. It'll be way faster.
EXAMPLES:

```
sage: a = maxima(str(2^1000))
sage: a.get_using_file()
'10715086071862673209484250490600018105614048117055336074437503883703510511249361224931983788
```

hasattr(*attrname*)

Returns whether the given attribute is already defined by this object, and in particular is not dynamically generated.

EXAMPLES:

```
sage: m = maxima('2')
sage: m.hasattr('integral')
True
sage: m.hasattr('gcd')
False
```

name(*new_name=None*)

Returns the name of self. If new_name is passed in, then this function returns a new object identitical to self whose name is new_name.

Note that this can overwrite existing variables in the system.

EXAMPLES:

```
sage: x = r([1,2,3]); x
[1] 1 2 3
sage: x.name()
'sage3'
sage: x = r([1,2,3]).name('x'); x
[1] 1 2 3
sage: x.name()
'x'

sage: s5 = gap.SymmetricGroup(5).name('s5')
sage: s5
SymmetricGroup( [ 1 .. 5 ] )
sage: s5.name()
's5'
```

sage()

Attempt to return a Sage version of this object.

EXAMPLES:

```
sage: gp(1/2).sage()
1/2
sage: _.parent()
Rational Field
```

class **ExpectFunction**(*parent, name*)

Expect function.

class **FunctionElement**(*obj, name*)

Expect function element.

help()

exception **PropTypeError**

console(*cmd*)

class **gc_disabled**()

This is a "with" statement context manager. Garbage collection is disabled within its scope. Nested usage is properly handled.

EXAMPLES:

```
sage: import gc
sage: from sage.interfaces.expect import gc_disabled
sage: gc.isenabled()
True
sage: with gc_disabled():
...       print gc.isenabled()
...       with gc_disabled():
...           print gc.isenabled()
...       print gc.isenabled()
False
False
False
sage: gc.isenabled()
True
```

is_ExpectElement(*x*)

reduce_load(*parent, x*)

tmp_expect_interface_local()

13.2 Interface to Axiom

TODO:

- Evaluation using a file is not done. Any input line with more than a few thousand characters would hang the system, so currently it automatically raises an exception.
- All completions of a given command.
- Interactive help.

Axiom is a free GPL-compatible (modified BSD license) general purpose computer algebra system whose development started in 1973 at IBM. It contains symbolic manipulation algorithms, as well as implementations of special functions, including elliptic functions and generalized hypergeometric functions. Moreover, Axiom has implementations of many functions relating to the invariant theory of the symmetric group S_n. For many links to Axiom documentation see http://wiki.axiom-developer.org.

AUTHORS:

- Bill Page (2006-10): Created this (based on maxima interface)

  Note: Bill Page put a huge amount of effort into the Sage Axiom interface over several days during the Sage Days 2 coding sprint. This is contribution is greatly appreciated.
- William Stein (2006-10): misc touchup.
- Bill Page (2007-08): Minor modifications to support axiom4sage-0.3

Note: The axiom4sage-0.3.spkg is based on an experimental version of the FriCAS fork of the Axiom project by Waldek Hebisch that uses pre-compiled cached Lisp code to build Axiom very quickly with clisp.

If the string "error" (case insensitive) occurs in the output of anything from axiom, a RuntimeError exception is raised.

EXAMPLES: We evaluate a very simple expression in axiom.

```
sage: axiom('3 * 5')                        #optional - axiom
15
sage: a = axiom(3) * axiom(5); a            #optional - axiom
15
```

The type of a is AxiomElement, i.e., an element of the axiom interpreter.

```
sage: type(a)                               #optional - axiom
<class 'sage.interfaces.axiom.AxiomElement'>
sage: parent(a)                             #optional - axiom
Axiom
```

The underlying Axiom type of a is also available, via the type method:

```
sage: a.type()                              #optional - axiom
PositiveInteger
```

We factor $x^5 - y^5$ in Axiom in several different ways. The first way yields a Axiom object.

```
sage: F = axiom.factor('x^5 - y^5'); F      #optional - axiom
           4      3    2 2    3     4
- (y - x)(y  + x y  + x y  + x y + x )
sage: type(F)                               #optional - axiom
<class 'sage.interfaces.axiom.AxiomElement'>
sage: F.type()                              #optional - axiom
Factored Polynomial Integer
```

Note that Axiom objects are normally displayed using "ASCII art".

```
sage: a = axiom(2/3); a          #optional - axiom
  2
  -
  3
sage: a = axiom('x^2 + 3/7'); a      #optional - axiom
   2   3
  x  + -
       7
```

The `axiom.eval` command evaluates an expression in axiom and returns the result as a string. This is exact as if we typed in the given line of code to axiom; the return value is what Axiom would print out.

```
sage: print axiom.eval('factor(x^5 - y^5)')   #optional - axiom
           4      3    2 2    3     4
- (y - x)(y  + x y  + x y  + x y + x )
Type: Factored Polynomial Integer
```

We can create the polynomial f as a Axiom polynomial, then call the factor method on it. Notice that the notation `f.factor()` is consistent with how the rest of Sage works.

```
sage: f = axiom('x^5 - y^5')                  #optional - axiom
sage: f^2                                     #optional - axiom
   10     5 5    10
  y   - 2x y  + x
sage: f.factor()                              #optional - axiom
```

```
           4       3     2 2     3      4
- (y - x)(y  + x y  + x y  + x y + x )
```

Control-C interruption works well with the axiom interface, because of the excellent implementation of axiom. For example, try the following sum but with a much bigger range, and hit control-C.

```
sage:  f = axiom('(x^5 - y^5)^10000')        # not tested
Interrupting Axiom...
...
<type 'exceptions.TypeError'>: Ctrl-c pressed while running Axiom


sage: axiom('1/100 + 1/101')                  #optional - axiom
   201
  -----
  10100
sage: a = axiom('(1 + sqrt(2))^5'); a         #optional - axiom
     +-+
  29\|2  + 41
```

TESTS: We check to make sure the subst method works with keyword arguments.

```
sage: a = axiom(x+2); a  #optional - axiom
x + 2
sage: a.subst(x=3)       #optional - axiom
5
```

We verify that Axiom floating point numbers can be converted to Python floats.

```
sage: float(axiom(2))      #optional - axiom
2.0
```

class **Axiom**(*name='axiom', command='axiom -nox -noclef', script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, init_code=, [')lisp (si::readline-off)']*)

console()

Spawn a new Axiom command-line session.

EXAMPLES:

```
sage: axiom.console() #not tested
                          AXIOM Computer Algebra System
                          Version: Axiom (January 2009)
                 Timestamp: Sunday January 25, 2009 at 07:08:54
------------------------------------------------------------------------------
   Issue )copyright to view copyright notices.
   Issue )summary for a summary of useful system commands.
   Issue )quit to leave AXIOM and return to shell.
------------------------------------------------------------------------------
```

class **AxiomElement**(*parent, value, is_name=False, name=None*)

class **AxiomExpectFunction**(*parent, name*)

class **AxiomFunctionElement**(*object, name*)

class **PanAxiom**(*name='axiom', command='axiom -nox -noclef', script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, init_code=, [')lisp (si::readline-off)']*)

Interface to a PanAxiom interpreter.

get (*var*)

Get the string value of the Axiom variable var.

EXAMPLES:

```
sage: axiom.set('xx', '2')     #optional - axiom
sage: axiom.get('xx')          #optional - axiom
'2'
sage: a = axiom('(1 + sqrt(2))^5') #optional - axiom
sage: axiom.get(a.name())          #optional - axiom
'      +-+\r\r\n  29\\|2  + 41'
```

set (*var, value*)

Set the variable var to the given value.

EXAMPLES:

```
sage: axiom.set('xx', '2')     #optional - axiom
sage: axiom.get('xx')          #optional - axiom
'2'

sage: fricas.set('xx', '2')     #optional - fricas
sage: fricas.get('xx')          #optional - fricas
'2'
```

trait_names (*verbose=True, use_disk_cache=True*)

Returns a list of all the commands defined in Axiom and optionally (per default) store them to disk.

EXAMPLES:

```
sage: c = axiom.trait_names(use_disk_cache=False, verbose=False) #optional - axiom
sage: len(c) > 100  #optional - axiom
True
sage: 'factor' in c  #optional - axiom
True
sage: '**' in c     #optional - axiom
False
sage: 'upperCase?' in c  #optional - axiom
False
sage: 'upperCase_q' in c #optional - axiom
True
sage: 'upperCase_e' in c #optional - axiom
True
```

class **PanAxiomElement** (*parent, value, is_name=False, name=None*)

as_type (*type*)

Returns self as type.

EXAMPLES:

```
sage: a = axiom(1.2); a            #optional - axiom
1.2
sage: a.as_type(axiom.DoubleFloat) #optional - axiom
1.2
sage: _.type()                     #optional - axiom
DoubleFloat

sage: a = fricas(1.2); a            #optional - fricas
1.2
sage: a.as_type(fricas.DoubleFloat) #optional - fricas
1.2
```

```
sage: _.type()                          #optional - fricas
DoubleFloat
```

comma(**args*)

Returns a Axiom tuple from self and args.

EXAMPLES:

```
sage: two = axiom(2)  #optional - axiom
sage: two.comma(3)    #optional - axiom
[2,3]
sage: two.comma(3,4)  #optional - axiom
[2,3,4]
sage: _.type()        #optional - axiom
Tuple PositiveInteger

sage: two = fricas(2)  #optional - fricas
sage: two.comma(3)     #optional - fricas
[2,3]
sage: two.comma(3,4)   #optional - fricas
[2,3,4]
sage: _.type()         #optional - fricas
Tuple PositiveInteger
```

type()

Returns the type of an AxiomElement.

EXAMPLES:

```
sage: axiom(x+2).type()  #optional - axiom
Polynomial Integer
```

unparsed_input_form()

Get the linear string representation of this object, if possible (often it isn't).

EXAMPLES:

```
sage: a = axiom(x^2+1); a     #optional - axiom
   2
  x  + 1
sage: a.unparsed_input_form() #optional - axiom
'x*x+1'

sage: a = fricas(x^2+1)       #optional - fricas
sage: a.unparsed_input_form() #optional - fricas
'x^2+1'
```

class **PanAxiomExpectFunction**(*parent, name*)

class **PanAxiomFunctionElement**(*object, name*)

axiom_console()

Spawn a new Axiom command-line session.

EXAMPLES:

```
sage: axiom_console() #not tested
                        AXIOM Computer Algebra System
                        Version: Axiom (January 2009)
                Timestamp: Sunday January 25, 2009 at 07:08:54
------------------------------------------------------------------------------
   Issue )copyright to view copyright notices.
   Issue )summary for a summary of useful system commands.
```

```
   Issue )quit to leave AXIOM and return to shell.
------------------------------------------------------------------------------
```

is_AxiomElement(*x*)

Returns True of x is of type AxiomElement.

EXAMPLES:

```
sage: from sage.interfaces.axiom import is_AxiomElement
sage: is_AxiomElement(axiom(2)) #optional - axiom
True
sage: is_AxiomElement(2)
False
```

reduce_load_Axiom()

Returns the Axiom interface object defined in sage.interfaces.axiom.

EXAMPLES:

```
sage: from sage.interfaces.axiom import reduce_load_Axiom
sage: reduce_load_Axiom()
Axiom
```

13.3 Interface to GAP

Sage provides an interface to the GAP system. This system provides extensive group theory, combinatorics, etc.

The GAP interface will only work if GAP is installed on your computer; this should be the case, since GAP is included with Sage. The interface offers three pieces of functionality:

1. `gap_console()` - A function that dumps you into an interactive command-line GAP session.
2. `gap(expr)` - Evaluation of arbitrary GAP expressions, with the result returned as a string.
3. `gap.new(expr)` - Creation of a Sage object that wraps a AP object. This provides a Pythonic interface to GAP. For example, if `f=gap.new(10)`, then `f.Factors()` returns the prime factorization of 10 computed using GAP.

13.3.1 First Examples

We factor an integer using GAP:

```
sage: n = gap(20062006); n
20062006
sage: n.parent()
Gap
sage: fac = n.Factors(); fac
[ 2, 17, 59, 73, 137 ]
sage: fac.parent()
Gap
sage: fac[1]
2
```

13.3.2 GAP and Singular

This example illustrates conversion between Singular and GAP via Sage as an intermediate step. First we create and factor a Singular polynomial.

```
sage: singular(389)
389
sage: R1 = singular.ring(0, '(x,y)', 'dp')
sage: f = singular('9*x^16-18*x^13*y^2-9*x^12*y^3+9*x^10*y^4-18*x^11*y^2+36*x^8*y^4+18*x^7*y^5-18*x^5
sage: F = f.factorize()
sage: print F
[1]:
   _[1]=9
   _[2]=x^6-2*x^3*y^2-x^2*y^3+y^4
   _[3]=-x^5+y^2
[2]:
   1,1,2
```

Next we convert the factor $-x^5 + y^2$ to a Sage multivariate polynomial. Note that it is important to let x and y be the generators of a polynomial ring, so the eval command works.

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: s = F[1][3].sage_polystring(); s
'-x**5+y**2'
sage: g = eval(s); g
-x^5 + y^2
```

Next we create a polynomial ring in GAP and obtain its indeterminates:

```
sage: R = gap.PolynomialRing('Rationals', 2); R
PolynomialRing( Rationals, ["x_1", "x_2"] )
sage: I = R.IndeterminatesOfPolynomialRing(); I
[ x_1, x_2 ]
```

In order to eval g in GAP, we need to tell GAP to view the variables `x0` and `x1` as the two generators of R. This is the one tricky part. In the GAP interpreter the object `I` has its own name (which isn't `I`). We can access its name using `I.name()`.

```
sage: _ = gap.eval("x := %s[1];; y := %s[2];;"%(I.name(), I.name()))
```

Now x_0 and x_1 are defined, so we can construct the GAP polynomial f corresponding to g:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: f = gap(str(g)); f
-x_1^5+x_2^2
```

We can call GAP functions on f. For example, we evaluate the GAP `Value` function, which evaluates f at the point $(1, 2)$.

```
sage: f.Value(I, [1,2])
3
sage: g(1,2)          # agrees
3
```

13.3.3 Saving and loading objects

Saving and loading GAP objects (using the dumps method, etc.) is *not* supported, since the output string representation of Gap objects is sometimes not valid input to GAP. Creating classes that wrap GAP objects *is* supported, via simply defining the a _gap_init_ member function that returns a string that when evaluated in GAP constructs the object. See `groups/permutation_group.py` for a nontrivial example of this.

13.3.4 Long Input

The GAP interface reads in even very long input (using files) in a robust manner, as long as you are creating a new object.

Note: Using `gap.eval` for long input is much less robust, and is not recommended.

```
sage: t = '"%s"'%10^10000   # ten thousand character string.
sage: a = gap(t)
```

13.3.5 Changing which GAP is used

Use this code to change which GAP interpreter is run. E.g.,

```
import sage.interfaces.gap
sage.interfaces.gap.gap_cmd = "/usr/local/bin/gap"
```

AUTHORS:

- David Joyner and William Stein: initial version(s)
- William Stein (2006-02-01): modified gap_console command so it uses exactly the same startup command as Gap.__init__.
- William Stein (2006-03-02): added tab completions: gap.[tab], x = gap(...), x.[tab], and docs, e.g., gap.function? and x.function?

class Gap (*max_workspace_size=None, maxread=100000, script_subdirectory=None, use_workspace_cache=True, server=None, server_tmpdir=None, logfile=None*)

Interface to the GAP interpreter.

AUTHORS:

•William Stein and David Joyner

console()

Spawn a new GAP command-line session.

EXAMPLES:

```
sage: gap.console() #not tested
GAP4, Version: 4.4.10 of 02-Oct-2007, x86_64-unknown-linux-gnu-gcc
gap>
```

cputime (*t=None*)

Returns the amount of CPU time that the GAP session has used. If `t` is not None, then it returns the difference between the current CPU time and `t`.

EXAMPLES:

```
sage: t = gap.cputime()
sage: t  #random
0.13600000000000001
sage: gap.Order(gap.SymmetricGroup(5))
120
sage: gap.cputime(t)  #random
0.059999999999999998
```

eval(*x, newlines=False, strip=True, **kwds*)

Send the code in the string s to the GAP interpreter and return the output as a string.

INPUT:

- `s` - string containing GAP code.
- `newlines` - bool (default: True); if False, remove all backslash-newlines inserted by the GAP output formatter.
- `strip` - ignored

EXAMPLES:

```
sage: gap.eval('2+2')
'4'
sage: gap.eval('Print(4); #test\n Print(6);')
'46'
sage: gap.eval('Print("#"); Print(6);')
'#6'
sage: gap.eval('4; \n 6;')
'4\n6'
```

function_call(*function, args=None, kwds=None*)

Calls the GAP function with args and kwds.

EXAMPLES:

```
sage: gap.function_call('SymmetricGroup', [5])
SymmetricGroup( [ 1 .. 5 ] )
```

If the GAP function does not return a value, but prints something to the screen, then a string of the printed output is returned.

```
sage: s = gap.function_call('Display', [gap.SymmetricGroup(5).CharacterTable()])
sage: type(s)
<class 'sage.interfaces.expect.AsciiArtString'>
sage: s.startswith('CT')
True
```

get(*var, use_file=False*)

Get the string representation of the variable var.

EXAMPLES:

```
sage: gap.set('x', '2')
sage: gap.get('x')
'2'
```

help(*s, pager=True*)

Print help on a given topic.

EXAMPLES:

```
sage: print gap.help('SymmetricGroup', pager=False)
Basic Groups ____________________________________________ Group Libraries
...
```

load_package(*pkg, verbose=False*)
Load the Gap package with the given name.
If loading fails, raise a RuntimeError exception.

save_workspace()

set(*var, value*)
Set the variable var to the given value.
EXAMPLES:

```
sage: gap.set('x', '2')
sage: gap.get('x')
'2'
```

trait_names()
EXAMPLES:

```
sage: c = gap.trait_names()
sage: len(c) > 100
True
sage: 'Order' in c
True
```

unbind(*var*)
Clear the variable named var.
EXAMPLES:

```
sage: gap.set('x', '2')
sage: gap.get('x')
'2'
sage: gap.unbind('x')
sage: gap.get('x')
...
RuntimeError: Gap produced error output
Variable: 'x' must have a value
...
```

version()
Returns the version of GAP being used.
EXAMPLES:

```
sage: gap.version()
'4.4.10'
```

class **GapElement**(*parent, value, is_name=False, name=None*)

bool()
EXAMPLES:

```
sage: bool(gap(2))
True
sage: gap(0).bool()
False
sage: gap('false').bool()
False
```

str(*use_file=False*)
EXAMPLES:

```
sage: print gap(2)
2
```

trait_names()
EXAMPLES:

```
sage: s5 = gap.SymmetricGroup(5)
sage: 'Centralizer' in s5.trait_names()
True
```

class GapFunction(*parent, name*)

class GapFunctionElement(*obj, name*)

gap_command(*use_workspace_cache=True, local=True*)

gap_console(*use_workspace_cache=True*)
Spawn a new GAP command-line session.

EXAMPLES:

```
sage: gap.console() #not tested
GAP4, Version: 4.4.10 of 02-Oct-2007, x86_64-unknown-linux-gnu-gcc
gap>
```

gap_reset_workspace(*max_workspace_size=None, verbose=False*)
Call this to completely reset the GAP workspace, which is used by default when Sage first starts GAP.

The first time you start GAP from Sage, it saves the startup state of GAP in the file

```
$HOME/.sage/gap-workspace
```

This is useful, since then subsequent startup of GAP is at least 10 times as fast. Unfortunately, if you install any new code for GAP, it won't be noticed unless you explicitly load it, e.g., with gap.load_package("my_package")

The packages sonata, guava, factint, gapdoc, grape, design, toric, and laguna are loaded in all cases before the workspace is saved, if they are available.

gap_version()
Returns the version of GAP being used.

EXAMPLES:

```
sage: gap_version()
'4.4.10'
```

gfq_gap_to_sage(*x, F*)
INPUT:

- x - gap finite field element
- F - Sage finite field

OUTPUT: element of F

EXAMPLES:

```
sage: x = gap('Z(13)')
sage: F = GF(13, 'a')
sage: F(x)
2
sage: F(gap('0*Z(13)'))
0
sage: F = GF(13^2, 'a')
sage: x = gap('Z(13)')
sage: F(x)
2
```

```
sage: x = gap('Z(13^2)^3')
sage: F(x)
12*a + 11
sage: F.multiplicative_generator()^3
12*a + 11
```

AUTHOR:

- David Joyner and William Stein

is_GapElement(*x*)

Returns True if x is a GapElement.

EXAMPLES:

```
sage: from sage.interfaces.gap import is_GapElement
sage: is_GapElement(gap(2))
True
sage: is_GapElement(2)
False
```

reduce_load()

Returns an invalid GAP element. Note that this is the object returned when a GAP element is unpickled.

EXAMPLES:

```
sage: from sage.interfaces.gap import reduce_load
sage: reduce_load()
(invalid object -- defined in terms of closed session)
sage: loads(dumps(gap(2)))
(invalid object -- defined in terms of closed session)
```

reduce_load_GAP()

Returns the GAP interface object definedin sage.interfaces.gap.

EXAMPLES:

```
sage: from sage.interfaces.gap import reduce_load_GAP
sage: reduce_load_GAP()
Gap
```

13.4 Interface to GP/Pari

Type `gp.[tab]` for a list of all the functions available from your Gp install. Type `gp.[tab]?` for Gp's help about a given function. Type `gp(...)` to create a new Gp object, and `gp.eval(...)` to run a string using Gp (and get the result back as a string).

EXAMPLES: We illustrate objects that wrap GP objects (gp is the PARI interpreter):

```
sage: M = gp('[1,2;3,4]')
sage: M
[1, 2; 3, 4]
sage: M * M
[7, 10; 15, 22]
sage: M + M
[2, 4; 6, 8]
```

```
sage: M.matdet()
-2
```

```
sage: E = gp.ellinit([1,2,3,4,5])
sage: E.ellglobalred()
[10351, [1, -1, 0, -1], 1]
sage: E.ellan(20)
[1, 1, 0, -1, -3, 0, -1, -3, -3, -3, -1, 0, 1, -1, 0, -1, 5, -3, 4, 3]
```

```
sage: primitive_root(7)
3
sage: x = gp("znlog( Mod(2,7), Mod(3,7))")
sage: 3^x % 7
2
```

```
sage: print gp("taylor(sin(x),x)")
x - 1/6*x^3 + 1/120*x^5 - 1/5040*x^7 + 1/362880*x^9 - 1/39916800*x^11 + 1/6227020800*x^13 - 1/130767
```

GP has a powerful very efficient algorithm for numerical computation of integrals.

```
sage: gp("a = intnum(x=0,6,sin(x))")
0.039829713349633979454347702077075594548     # 32-bit
0.039829713349633979454347702077075594548     # 64-bit
sage: gp("a")
0.039829713349633979454347702077075594548     # 32-bit
0.039829713349633979454347702077075594548     # 64-bit
sage: gp.kill("a")
sage: gp("a")
a
```

Note that gp ASCII plots *do* work in Sage, as follows:

```
sage: print gp.eval("plot(x=0,6,sin(x))")
<BLANKLINE>
0.9988963 |''''''''''''_x"...x_''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''|
          |           x"         "x                                                                    |
          |         _"             "_                                                                  |
          |        x                 x                                                                 |
          |       "                   "                                                                |
          |      "                     "                                                               |
          |    _"                       "_                                                             |
          |  _                            _                                                            |
          | _                              _                                                           |
          |_                                _                                                          |
          _,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,,
          |                                  "                                                        |
          |                                   "                                                       |
          |                                    "                                                      |
          |                                     "                                                    "|
          |                                      "_                                               _"  |
          |                                        _                                           _     |
          |                                         _                                       _        |
          |                                          x                                   x          |
          |                                           "_                               _"           |
          |                                             x_                         _x              |
```

```
-0.998955 |.........................................."x____x".........|
          0                                                          6
```

The GP interface reads in even very long input (using files) in a robust manner, as long as you are creating a new object.

```
sage: t = '"%s"'%10^10000   # ten thousand character string.
sage: a = gp.eval(t)
sage: a = gp(t)
```

In Sage, the PARI large galois groups datafiles should be installed by default:

```
sage: f = gp('x^9 - x - 2')
sage: f.polgalois()
[362880, -1, 34, "S9"]
```

AUTHORS:

- William Stein
- David Joyner: some examples
- William Stein (2006-03-01): added tab completion for methods: gp.[tab] and x = gp(blah); x.[tab]
- William Stein (2006-03-01): updated to work with PARI 2.2.12-beta
- William Stein (2006-05-17): updated to work with PARI 2.2.13-beta

class Gp (*stacksize=10000000, maxread=100000, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, init_list_length=1024*)

Interface to the PARI gp interpreter.

Type `gp.[tab]` for a list of all the functions available from your Gp install. Type `gp.[tab]?` for Gp's help about a given function. Type `gp(...)` to create a new Gp object, and `gp.eval(...)` to run a string using Gp (and get the result back as a string).

console()

Spawn a new GP command-line session.

EXAMPLES:

```
sage: gp.console() #not tested
GP/PARI CALCULATOR Version 2.3.3 (released)
amd64 running linux (x86-64/GMP-4.2.1 kernel) 64-bit version
compiled: Feb 22 2008, gcc-4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)
(readline v5.2 enabled, extended help available)
...
```

cputime (*t=None*)

cputime for pari - cputime since the pari process was started.

INPUT:

- `t` - (default: None); if not None, then returns time since t

> **Warning:** If you call gettime explicitly, e.g., gp.eval('gettime'), you will throw off this clock.

EXAMPLES:

```
sage: gp.cputime()          # random output
0.0080000000000000002
sage: gp.factor('2^157-1')
[852133201, 1; 60726444167, 1; 1654058017289, 1; 2134387368610417, 1]
sage: gp.cputime()          # random output
0.26900000000000002
```

get(*var*)

Get the value of the variable var.

EXAMPLES:

```
sage: gp.set('x', '2')
sage: gp.get('x')
'2'
```

get_precision()

Return the current PARI precision for real number computations.

EXAMPLES:

```
sage: gp.get_precision()
28              # 32-bit
38              # 64-bit
```

get_real_precision()

Return the current PARI precision for real number computations.

EXAMPLES:

```
sage: gp.get_precision()
28              # 32-bit
38              # 64-bit
```

help(*command*)

Returns GP's help for `command`.

EXAMPLES:

```
sage: gp.help('gcd')
'gcd(x,{y}): greatest common divisor of x and y.'
```

kill(*var*)

EXAMPLES:

```
sage: gp.set('xx', '22')
sage: gp.get('xx')
'22'
sage: gp.kill('xx')
sage: gp.get('xx')
'xx'
```

new_with_bits_prec(*s, precision=0*)

Creates a GP object from s with `precision` bits of precision. GP actually automatically increases this precision to the nearest word (i.e. the next multiple of 32 on a 32-bit machine, or the next multiple of 64 on a 64-bit machine).

EXAMPLES:

```
sage: pi_def = gp(pi); pi_def
3.1415926535897932384626433383                  # 32-bit
3.1415926535897932384626433832795028842        # 64-bit
sage: pi_def.precision()
28                                              # 32-bit
```

```
38                                                  # 64-bit
sage: pi_150 = gp.new_with_bits_prec(pi, 150)
sage: new_prec = pi_150.precision(); new_prec
48                                                  # 32-bit
57                                                  # 64-bit
sage: old_prec = gp.get_precision()
sage: gp.set_precision(new_prec)
28                                                  # 32-bit
38                                                  # 64-bit
sage: pi_150
3.14159265358979323846264338327950288419716939937  # 32-bit
3.14159265358979323846264338327950288419716939937510582098  # 64-bit
sage: gp.set_precision(old_prec)
48                                                  # 32-bit
57                                                  # 64-bit
```

quit (*verbose=False, timeout=0.25*)

EXAMPLES:

```
sage: a = gp('10'); a
10
sage: gp.quit()
sage: a
(invalid object -- defined in terms of closed session)
sage: gp(pi)
3.14159265358979323846264338327950288420    # 64-bit
3.14159265358979323846264383                # 32-bit
```

set (*var, value*)

Set the variable var to the given value.

EXAMPLES:

```
sage: gp.set('x', '2')
sage: gp.get('x')
'2'
```

set_precision (*prec=None*)

Sets the current PARI precision (in decimal digits) for real number computations, and returns the old one.

EXAMPLES:

```
sage: old_prec = gp.set_precision(53)
sage: gp.get_precision()
57
sage: gp.set_precision(old_prec)
57
sage: gp.get_precision()
28              # 32-bit
38              # 64-bit
```

set_real_precision (*prec=None*)

Sets the current PARI precision (in decimal digits) for real number computations, and returns the old one.

EXAMPLES:

```
sage: old_prec = gp.set_precision(53)
sage: gp.get_precision()
57
sage: gp.set_precision(old_prec)
57
sage: gp.get_precision()
```

```
28              # 32-bit
38              # 64-bit
```

trait_names()
EXAMPLES:

```
sage: c = gp.trait_names()
sage: len(c) > 100
True
sage: 'gcd' in c
True
```

version()
Returns the version of GP being used.
EXAMPLES:

```
sage: gp.version()
((2, 3, 3), 'GP/PARI CALCULATOR Version 2.3.3 (released)')
```

class GpElement(*parent, value, is_name=False, name=None*)
EXAMPLES: This example illustrates dumping and loading GP elements to compressed strings.

```
sage: a = gp(39393)
sage: loads(a.dumps()) == a
True
```

Since dumping and loading uses the string representation of the object, it need not result in an identical object from the point of view of PARI:

```
sage: E = gp('ellinit([1,2,3,4,5])')
sage: loads(E.dumps()) == E
False
sage: loads(E.dumps())
[1, 2, 3, 4, 5, 9, 11, 29, 35, -183, -3429, -10351, 6128487/10351, [-1.6189099322673713423780009
[1, 2, 3, 4, 5, 9, 11, 29, 35, -183, -3429, -10351, 6128487/10351, [-1.6189099322673713423780009
sage: E
[1, 2, 3, 4, 5, 9, 11, 29, 35, -183, -3429, -10351, 6128487/10351, [-1.6189099322673713423780009
[1, 2, 3, 4, 5, 9, 11, 29, 35, -183, -3429, -10351, 6128487/10351, [-1.6189099322673713423780009
```

The two elliptic curves look the same, but internally the floating point numbers are slightly different.

bool()
EXAMPLES:

```
sage: gp(2).bool()
True
sage: bool(gp(2))
True
sage: bool(gp(0))
False
```

trait_names()
EXAMPLES:

```
sage: 'gcd' in gp(2).trait_names()
True
```

class GpFunction(*parent, name*)

class GpFunctionElement(*obj, name*)

gp_console()
Spawn a new GP command-line session.

EXAMPLES:

```
sage: gp.console() #not tested
GP/PARI CALCULATOR Version 2.3.3 (released)
amd64 running linux (x86-64/GMP-4.2.1 kernel) 64-bit version
compiled: Feb 22 2008, gcc-4.1.3 20070929 (prerelease) (Ubuntu 4.1.2-16ubuntu2)
(readline v5.2 enabled, extended help available)
...
```

gp_version()
EXAMPLES:

```
sage: gp.version()       # random output
((2, 3, 1), 'GP/PARI CALCULATOR Version 2.3.1 (0)')
```

is_GpElement(*x*)
Returns True of x is a GpElement.

EXAMPLES:

```
sage: from sage.interfaces.gp import is_GpElement
sage: is_GpElement(gp(2))
True
sage: is_GpElement(2)
False
```

reduce_load_GP()
Returns the GP interface object defined in sage.interfaces.gp.

EXAMPLES:

```
sage: from sage.interfaces.gp import reduce_load_GP
sage: reduce_load_GP()
GP/PARI interpreter
```

13.5 Interface to the Gnuplot interpreter.

class Gnuplot()
Interface to the Gnuplot interpreter.

console()

gnuplot()

interact(*cmd*)

plot(*cmd, file=None, verbose=True, reset=True*)
Draw the plot described by cmd, and possibly also save to an eps or png file.

INPUT:

- `cmd` - string
- `file` - string (default: None), if specified save plot to given file, which may be either an eps (default) or png file.
- `verbose` - print some info
- `reset` - True: reset gnuplot before making graph

OUTPUT: displays graph

Note: Note that ^ s are replaced by ** s before being passed to gnuplot.

plot3d(*f, xmin=-1, xmax=1, ymin=-1, ymax=1, zmin=-1, zmax=1, title=None, samples=25, isosamples=20, xlabel='x', ylabel='y', interact=True*)

plot3d_parametric(*f='cos(u)*(3 + v*cos(u/2)), sin(u)*(3 + v*cos(u/2)), v*sin(u/2)', range1=', [u=-pi:pi], ', range2=', [v=-0.2:0.2], ', samples=50, title=None, interact=True*)

Draw a parametric 3d surface and rotate it interactively.

INPUT:

- `f` - (string) a function of two variables, e.g., 'cos(u)*(3 + v*cos(u/2)), sin(u)*(3 + v*cos(u/2)), v*sin(u/2)'
- `range1` - (string) range of values for one variable, e.g., '[u=-pi:pi]'
- `range2` - (string) range of values for another variable, e.g., '[v=-0.2:0.2]'
- `samples` - (int) number of sample points to use
- `title` - (string) title of the graph.

EXAMPLES:

```
sage: gnuplot.plot3d_parametric('v^2*sin(u), v*cos(u), v*(1-v)')    # optional -- requires gr
```

gnuplot_console()

13.6 Interface to KASH

Sage provides an interface to the KASH computer algebra system, which is a *free* (as in beer!) but *closed source* program for algebraic number theory that shares much common code with Magma. To use KASH, you must install the appropriate optional Sage package by typing something like "sage -i kash3-linux-2005.11.22" or "sage -i kash3_osx-2005.11.22". For a list of optional packages type "sage -optional". If you type one of the above commands, the (about 16MB) package will be downloaded automatically (you don't have to do that).

It is not enough to just have KASH installed on your computer. Note that the KASH Sage package is currently only available for Linux and OSX. If you need Windows, support contact me (wstein@gmail.com).

The KASH interface offers three pieces of functionality:

1. `kash_console()` - A function that dumps you into an interactive command-line KASH session. Alternatively,

   type `!kash` from the Sage prompt.

2. `kash(expr)` - Creation of a Sage object that wraps a KASH object. This provides a Pythonic interface to KASH. For example, if `f=kash.new(10)`, then `f.Factors()` returns the prime factorization of 10 computed using KASH.

3. `kash.function_name(args ...)` - Call the indicated KASH function with the given arguments are return the result as a KASH object.

4. `kash.eval(expr)` - Evaluation of arbitrary KASH expressions, with the result returned as a string.

13.6.1 Issues

For some reason hitting Control-C to interrupt a calculation doesn't work correctly. (TODO)

13.6.2 Tutorial

The examples in this tutorial require that the optional kash package be installed.

Basics

Basic arithmetic is straightforward. First, we obtain the result as a string.

```
sage: kash.eval('(9 - 7) * (5 + 6)')                # optional -- kash
'22'
```

Next we obtain the result as a new KASH object.

```
sage: a = kash('(9 - 7) * (5 + 6)'); a              # optional -- kash

22
sage: a.parent()                                    # optional -- kash
Kash
```

We can do arithmetic and call functions on KASH objects:

```
sage: a*a                                           # optional -- kash
484
sage: a.Factorial()                                 # optional -- kash
1124000727777607680000
```

Integrated Help

Use the `kash.help(name)` command to get help about a given command. This returns a list of help for each of the definitions of `name`. Use `print kash.help(name)` to nicely print out all signatures.

Arithmetic

Using the `kash.new` command we create Kash objects on which one can do arithmetic.

```
sage: a = kash(12345)                               # optional -- kash
sage: b = kash(25)                                  # optional -- kash
sage: a/b                                           # optional -- kash
2469/5
sage: a**b                                          # optional -- kash
193765903041146393565116739165642262657761441158615231767486923346401992277143215887218713760375976
```

Variable assignment

Variable assignment using `kash` is takes place in Sage.

```
sage: a = kash('32233')                             # optional -- kash
sage: a                                             # optional -- kash
32233
```

In particular, `a` is not defined as part of the KASH session itself.

```
sage: kash.eval('a')                                # optional -- kash
"Error, the variable 'a' must have a value"
```

Use `a.name()` to get the name of the KASH variable:

```
sage: a.name()                                  # somewhat random and optional - kash
'sage0'
sage: kash(a.name())                            # optional -- kash
32233
```

Integers and Rationals

We illustrate arithmetic with integers and rationals in KASH.

```
sage: F = kash.Factorization(4352)              # optional -- kash
sage: F[1]                                      # optional -- kash
<2, 8>
sage: F[2]                                      # optional -- kash
<17, 1>
sage: F                                         # optional -- kash
[ <2, 8>, <17, 1> ], extended by:
  ext1 := 1,
  ext2 := Unassign
```

Note: For some very large numbers KASH's integer factorization seems much faster than PARI's (which is the default in Sage).

```
sage: kash.GCD(15,25)                           # optional -- kash
5
sage: kash.LCM(15,25)                           # optional -- kash
75
sage: kash.Div(25,15)                           # optional -- kash
1
sage: kash(17) % kash(5)                        # optional -- kash
2
sage: kash.IsPrime(10007)                       # optional -- kash
TRUE
sage: kash.IsPrime(2005)                        # optional -- kash
FALSE

sage: kash.NextPrime(10007)                     # optional -- kash
10009
```

Real and Complex Numbers

```
sage: kash.Precision()                          # optional -- kash
30
sage: kash('R')                                 # optional -- kash
Real field of precision 30
sage: kash.Precision(40)                        # optional -- kash
40
sage: kash('R')                                 # optional -- kash
Real field of precision 40
sage: z = kash('1 + 2*I')                       # optional -- kash
sage: z                                         # optional -- kash
1.0000000000000000000000000000000000000000 + 2.0000000000000000000000000000000000000000*I
sage: z*z                                       # optional -- kash
-3.0000000000000000000000000000000000000000 + 4.0000000000000000000000000000000000000000*I
```

```
sage: kash.Cos('1.24')                          # optional -- kash
0.32479628443877623657769341569738039969 92
sage: kash('1.24').Cos()                        # optional -- kash
0.32479628443877623657769341569738039969 92

sage: kash.Exp('1.24')                          # optional -- kash
3.45561346476267559805761549412199817540 0

sage: kash.Precision(30)                        # optional -- kash
30
sage: kash.Log('3+4*I')                         # optional -- kash
1.60943791243410037460075933323 + 0.92729521800161223242851246292 2*I
sage: kash.Log('I')                             # optional -- kash
1.57079632679489661923132169164*I

sage: kash.Sqrt(4)                              # optional -- kash
2.00000000000000000000000000000
sage: kash.Sqrt(2)                              # optional -- kash
1.41421356237309504880168872421

sage: kash.Floor('9/5')                         # optional -- kash
1
sage: kash.Floor('3/5')                         # optional -- kash
0

sage: x_c = kash('3+I')                         # optional -- kash
sage: x_c.Argument()                            # optional -- kash
0.32175055439664219340140461435 9
sage: x_c.Imaginary()                           # optional -- kash
1.00000000000000000000000000000
```

Lists

Note that list appends are completely different in KASH than in Python. Use underscore after the function name for the mutation version.

```
sage: v = kash([1,2,3]); v                      # optional -- kash
[ 1, 2, 3 ]
sage: v[1]                                      # optional -- kash
1
sage: v[3]                                      # optional -- kash
3
sage: v.Append([5])                             # optional -- kash
[ 1, 2, 3, 5 ]
sage: v                                         # optional -- kash
[ 1, 2, 3 ]
sage: v.Append_([5, 6])                         # optional -- kash
SUCCESS
sage: v                                         # optional -- kash
[ 1, 2, 3, 5, 6 ]
sage: v.Add(5)                                  # optional -- kash
[ 1, 2, 3, 5, 6, 5 ]
sage: v                                         # optional -- kash
[ 1, 2, 3, 5, 6 ]
sage: v.Add_(5)                                 # optional -- kash
SUCCESS
```

```
sage: v                                          # optional -- kash
[ 1, 2, 3, 5, 6, 5 ]
```

The `Apply` command applies a function to each element of a list.

:: sage: L = kash([1,2,3,4]) # optional – kash sage: L.Apply('i -> 3*i') # optional – kash [3, 6, 9, 12] sage: L # optional – kash [1, 2, 3, 4] sage: L.Apply('IsEven') # optional – kash [FALSE, TRUE, FALSE, TRUE] sage: L # optional – kash [1, 2, 3, 4]

Ranges

the following are examples of ranges.

```
sage: L = kash('[1..10]')                        # optional -- kash
sage: L                                          # optional -- kash
[ 1 .. 10 ]
sage: L = kash('[2,4..100]')                     # optional -- kash
sage: L                                          # optional -- kash
[ 2, 4 .. 100 ]
```

Sequences

Tuples

Polynomials

```
sage: f = kash('X^3 + X + 1')                    # optional -- kash
sage: f + f                                      # optional -- kash
2*X^3 + 2*X + 2
sage: f * f                                      # optional -- kash
X^6 + 2*X^4 + 2*X^3 + X^2 + 2*X + 1
sage: f.Evaluate(10)                             # optional -- kash
1011
sage: Qx = kash.PolynomialAlgebra('Q')           # optional -- kash
sage: Qx.gen(1)**5 + kash('7/3')    # sage1 below somewhat random; optional -- kash
sage1.1^5 + 7/3
```

Number Fields

We create an equation order.

```
sage: f = kash('X^5 + 4*X^4 - 56*X^2 -16*X + 192')     # optional -- kash
sage: OK = f.EquationOrder()                           # optional -- kash
sage: OK                                               # optional -- kash
Equation Order with defining polynomial X^5 + 4*X^4 - 56*X^2 - 16*X + 192 over Z
```

```
sage: f = kash('X^5 + 4*X^4 - 56*X^2 -16*X + 192')     # optional -- kash
sage: O = f.EquationOrder()                            # optional -- kash
sage: a = O.gen(2)                                     # optional -- kash
sage: a                                                # optional -- kash
[0, 1, 0, 0, 0]
```

```
sage: O.Basis()        # output somewhat random; optional -- kash
[
_NG.1,
_NG.2,
_NG.3,
_NG.4,
_NG.5
]
sage: O.Discriminant()              # optional -- kash
1364202618880
sage: O.MaximalOrder()    # name sage2 below somewhat random; optional -- kash
Maximal Order of sage2

sage: O = kash.MaximalOrder('X^3 - 77')                  # optional -- kash
sage: I = O.Ideal(5,[2, 1, 0])                           # optional -- kash
sage: I                    # name sage14 below random; optional -- kash
Ideal of sage14
Two element generators:
[5, 0, 0]
[2, 1, 0]

sage: F = I.Factorisation()                  # optional -- kash
sage: F                    # name sage14 random; optional -- kash
[
<Prime Ideal of sage14
Two element generators:
[5, 0, 0]
[2, 1, 0], 1>
]
```

Determining whether an ideal is principal.

```
sage: I.IsPrincipal()                      # optional -- kash
FALSE, extended by:
ext1 := Unassign
```

Computation of class groups and unit groups:

```
sage: f = kash('X^5 + 4*X^4 - 56*X^2 -16*X + 192')         # optional -- kash
sage: O = kash.EquationOrder(f)                            # optional -- kash
sage: OK = O.MaximalOrder()                                # optional -- kash
sage: OK.ClassGroup()       # name sage32 below random; optional -- kash
Abelian Group isomorphic to Z/6
  Defined on 1 generator
  Relations:
  6*sage32.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: sage32 to ids/ord^num: _AA


sage: U = OK.UnitGroup()                                  # optional -- kash
sage: U        # name sage34 below random; optional -- kash
Abelian Group isomorphic to Z/2 + Z + Z
  Defined on 3 generators
  Relations:
  2*sage34.1 = 0, extended by:
  ext1 := Mapping from: grp^abl: sage34 to ord^num: sage30
```

```
sage: kash.Apply('x->%s.ext1(x)'%U.name(), U.Generators().List())      # optional -- kash
[ [1, -1, 0, 0, 0], [1, 1, 0, 0, 0], [-1, 0, 0, 0, 0] ]
```

Function Fields

```
sage: k = kash.FiniteField(25)                                         # optional -- kash
sage: kT = k.RationalFunctionField()                                   # optional -- kash
sage: kTy = kT.PolynomialAlgebra()                                     # optional -- kash
sage: T = kT.gen(1)                                                    # optional -- kash
sage: y = kTy.gen(1)                                                   # optional -- kash
sage: f = y**3 + T**4 + 1                                              # optional -- kash
```

13.6.3 Long Input

The KASH interface reads in even very long input (using files) in a robust manner, as long as you are creating a new object.

Note: Using `kash.eval` for long input is much less robust, and is not recommended.

```
sage: a = kash(range(10000))                                           # optional -- kash
```

Note that KASH seems to not support string or integer literals with more than 1024 digits, which is why the above example uses a list unlike for the other interfaces.

class Kash (*max_workspace_size=None, maxread=100000, script_subdirectory=None, restart_on_ctrlc=True, logfile=None, server=None, server_tmpdir=None*)

Interface to the Kash interpreter.

AUTHORS:

- William Stein and David Joyner

console()

eval(*x, newlines=False, strip=True, **kwds*)

Send the code in the string s to the Kash interpreter and return the output as a string.

INPUT:

- `s` - string containing Kash code.
- `newlines` - bool (default: True); if False, remove all backslash-newlines inserted by the Kash output formatter.
- `strip` - ignored

get(*var*)

Get the value of the variable var.

help(*name=None*)

Return help on KASH commands.

Returns help on all commands with a given name. If name is None, return the location of the installed Kash html documentation.

EXAMPLES:

```
sage: X = kash.help('IntegerRing')    # optional -- kash
```

There is one entry in X for each item found in the documentation for this function: If you type `print X[0]` you will get help on about the first one, printed nicely to the screen.

AUTHORS:

•Sebastion Pauli (2006-02-04): during Sage coding sprint

help_search(*name*)

set(*var, value*)
Set the variable var to the given value.

version()

class KashDocumentation()

class KashElement(*parent, value, is_name=False, name=None*)

is_KashElement(*x*)

kash_console()

kash_version()

reduce_load_Kash()

13.7 Interface to Magma

Note: You must have `magma` installed on your computer for this interface to work. Magma is not free, so it is not included with Sage, but you can obtain it from http://magma.maths.usyd.edu.au/.

Type `magma.[tab]` for a list of all the functions available from your Magma install. Type `magma.[tab]?` for Magma's help about a given function. Type `magma(...)` to create a new Magma object, and `magma.eval(...)` to run a string using Magma (and get the result back as a string).

Sage provides an interface to the Magma computational algebra system. This system provides extensive functionality for number theory, group theory, combinatorics and algebra.

The Magma interface offers three pieces of functionality:

1. `magma_console()` - A function that dumps you into an interactive command-line Magma session.
2. `magma(expr)` - Evaluation of arbitrary Magma expressions, with the result returned as a string.
3. `magma.new(expr)` - Creation of a Sage object that wraps a Magma object. This provides a Pythonic interface to Magma. For example, if `f=magma.new(10)`, then `f.Factors()` returns the prime factorization of 10 computed using Magma.

13.7.1 Parameters

Some Magma functions have optional "parameters", which are arguments that in Magma go after a colon. In Sage, you pass these using named function arguments. For example,

```
sage: E = magma('EllipticCurve([0,1,1,-1,0])')                    # optional - magma
sage: E.Rank(Bound = 5)                                           # optional - magma
0
```

13.7.2 Multiple Return Values

Some Magma functions return more than one value. You can control how many you get using the `nvals` named parameter to a function call:

```
sage: n = magma(100)                                                    # optional - magma
sage: n.IsSquare(nvals = 1)                                             # optional - magma
true
sage: n.IsSquare(nvals = 2)                                             # optional - magma
(true, 10)
sage: n = magma(-2006)                                                  # optional - magma
sage: n.Factorization()                                                 # optional - magma
[ <2, 1>, <17, 1>, <59, 1> ]
sage: n.Factorization(nvals=2)                                          # optional - magma
([ <2, 1>, <17, 1>, <59, 1> ], -1)
```

We verify that an obviously principal ideal is principal:

```
sage: _ = magma.eval('R<x> := PolynomialRing(RationalField())')     # optional - magma
sage: O = magma.NumberField('x^2+23').MaximalOrder()                # optional - magma
sage: I = magma('ideal<%s|%s.1>'%(O.name(),O.name()))               # optional - magma
sage: I.IsPrincipal(nvals=2)                                        # optional - magma
(true, [1, 0])
```

13.7.3 Long Input

The Magma interface reads in even very long input (using files) in a robust manner.

```
sage: t = '"%s"'%10^10000   # ten thousand character string.        # optional - magma
sage: a = magma.eval(t)                                             # optional - magma
sage: a = magma(t)                                                  # optional - magma
```

13.7.4 Other Examples

We compute a space of modular forms with character.

```
sage: N = 20
sage: D = 20
sage: eps_top = fundamental_discriminant(D)
sage: eps = magma.KroneckerCharacter(eps_top, RationalField())          # optional - magma
sage: M2 = magma.ModularForms(eps)                                      # optional - magma
sage: print M2                                                          # optional - magma
Space of modular forms on Gamma_1(5) ...
sage: print M2.Basis()                                                  # optional - magma
[
1 + 10*q^2 + 20*q^3 + 20*q^5 + 60*q^7 + ...
q + q^2 + 2*q^3 + 3*q^4 + 5*q^5 + 2*q^6 + ...
]
```

In Sage/Python (and sort of C++) coercion of an element x into a structure S is denoted by S(x). This also works for the Magma interface:

```
sage: G = magma.DirichletGroup(20)                                      # optional - magma
sage: G.AssignNames(['a', 'b'])                                         # optional - magma
sage: (G.1).Modulus()                                                   # optional - magma
20
sage: e = magma.DirichletGroup(40)(G.1)                                 # optional - magma
sage: print e                                                           # optional - magma
```

```
$.1
sage: print e.Modulus()                                                    # optional - magma
40
```

We coerce some polynomial rings into Magma:

```
sage: R.<y> = PolynomialRing(QQ)
sage: S = magma(R)                                                         # optional - magma
sage: print S                                                              # optional - magma
Univariate Polynomial Ring in y over Rational Field
sage: S.1                                                                  # optional - magma
y
```

This example illustrates that Sage doesn't magically extend how Magma implicit coercion (what there is, at least) works. The errors below are the result of Magma having a rather limited automatic coercion system compared to Sage's:

```
sage: R.<x> = ZZ[]
sage: x * 5
5*x
sage: x * 1.0
1.00000000000000*x
sage: x * (2/3)
2/3*x
sage: y = magma(x)                                                         # optional - magma
sage: y * 5                                                                # optional - magma
5*x
sage: y * 1.0                                                              # optional - magma
...
TypeError: unsupported operand parent(s) for '*': 'Magma' and 'Real Field with 53 bits of precision'
sage: y * (2/3)                                                            # optional - magma
...
TypeError: unsupported operand parent(s) for '*': 'Magma' and 'Rational Field'
```

AUTHORS:

- William Stein (2005): initial version
- William Stein (2006-02-28): added extensive tab completion and interactive IPython documentation support.
- William Stein (2006-03-09): added nvals argument for magma.functions...

class Magma (*maxread=10000, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None, user_config=False*)

Interface to the Magma interpreter.

Type `magma.[tab]` for a list of all the functions available from your Magma install. Type `magma.[tab]?` for Magma's help about a given function. Type `magma(...)` to create a new Magma object, and `magma.eval(...)` to run a string using Magma (and get the result back as a string).

Note: If you do not own a local copy of Magma, try using the `magma_free` command instead, which uses the free demo web interface to Magma.

EXAMPLES:

You must use nvals = 0 to call a function that doesn't return anything, otherwise you'll get an error. (nvals is the number of return values.)

```
sage: magma.SetDefaultRealFieldPrecision(200, nvals=0)  # magma >= v2.12; optional - magma
sage: magma.eval('1.1')   # optional - magma
'1.1000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
sage: magma.SetDefaultRealFieldPrecision(30, nvals=0)  # optional - magma
```

Attach (*filename*)

Attach the given file to the running instance of Magma.

Attaching a file in Magma makes all intrinsics defined in the file available to the shell. Moreover, if the file doesn't start with the `freeze;` command, then the file is reloaded whenever it is changed. Note that functions and procedures defined in the file are *not* available. For only those, use `magma.load(filename)`.

INPUT:

- `filename` - a string

EXAMPLES: Attaching a file that exists is fine:

```
sage: magma.attach('%s/data/extcode/magma/sage/basic.m'%Sage_ROOT)    # optional - magma
```

Attaching a file that doesn't exist raises an exception:

```
sage: magma.attach('%s/data/extcode/magma/sage/basic2.m'%Sage_ROOT)     # optional - magma
...
RuntimeError: Error evaluating Magma code...
```

AttachSpec (*filename*)

Attach the given spec file to the running instance of Magma.

This can attach numerous other files to the running Magma (see the Magma documentation for more details).

INPUT:

- `filename` - a string

EXAMPLES:

```
sage: magma.attach_spec('%s/data/extcode/magma/spec'%SAGE_ROOT)    # optional - magma
sage: magma.attach_spec('%s/data/extcode/magma/spec2'%SAGE_ROOT)   # optional - magma
...
RuntimeError: Can't open package spec file .../data/extcode/magma/spec2 for reading (No such
```

GetVerbose (*type*)

Get the verbosity level of a given algorithm class etc. in Magma.

INPUT:

- `type` - string (e.g. 'Groebner'), see Magma documentation

Note: This method is provided to be consistent with the Magma naming convention.

EXAMPLES:

```
sage: magma.SetVerbose("Groebner", 2)       # optional - magma
sage: magma.GetVerbose("Groebner")          # optional - magma
2
```

SetVerbose (*type, level*)

Set the verbosity level for a given algorithm class etc. in Magma.

INPUT:

- `type` - string (e.g. 'Groebner'), see Magma documentation
- `level` - integer = 0

Note: This method is provided to be consistent with the Magma naming convention.

```
sage: magma.SetVerbose("Groebner", 2)       # optional - magma
sage: magma.GetVerbose("Groebner")          # optional - magma
2
```

attach (*filename*)

Attach the given file to the running instance of Magma.

Attaching a file in Magma makes all intrinsics defined in the file available to the shell. Moreover, if the file doesn't start with the `freeze;` command, then the file is reloaded whenever it is changed. Note that functions and procedures defined in the file are *not* available. For only those, use `magma.load(filename)`.

INPUT:

- `filename` - a string

EXAMPLES: Attaching a file that exists is fine:

```
sage: magma.attach('%s/data/extcode/magma/sage/basic.m'%Sage_ROOT)     # optional - magma
```

Attaching a file that doesn't exist raises an exception:

```
sage: magma.attach('%s/data/extcode/magma/sage/basic2.m'%Sage_ROOT)      # opticnal - magma
...
RuntimeError: Error evaluating Magma code...
```

attach_spec (*filename*)

Attach the given spec file to the running instance of Magma.

This can attach numerous other files to the running Magma (see the Magma documentation for more details).

INPUT:

- `filename` - a string

EXAMPLES:

```
sage: magma.attach_spec('%s/data/extcode/magma/spec'%SAGE_ROOT)     # optional - magma
sage: magma.attach_spec('%s/data/extcode/magma/spec2'%SAGE_ROOT)    # optional - magma
...
RuntimeError: Can't open package spec file .../data/extcode/magma/spec2 for reading (No such
```

bar_call (*left, name, gens, nvals=1*)

This is a wrapper around the Magma constructor

nameleft gens

returning nvals.

INPUT:

- `left` - something coerceable to a magma object
- `name` - name of the constructor, e.g., sub, quo, ideal, etc.
- `gens` - if a list/tuple, each item is coerced to magma; otherwise gens itself is converted to magma
- `nvals` - positive integer; number of return values

OUTPUT: a single magma object if nvals == 1; otherwise a tuple of nvals magma objects.

EXAMPLES: The bar_call function is used by the sub, quo, and ideal methods of Magma elements. Here we illustrate directly using bar_call to create quotients:

```
sage: V = magma.RModule(ZZ,3)     # optional - magma
sage: V                           # optional - magma
RModule(IntegerRing(), 3)
sage: magma.bar_call(V, 'quo', [[1,2,3]], nvals=1)  # optional - magma
RModule(IntegerRing(), 2)
sage: magma.bar_call(V, 'quo', [[1,2,3]], nvals=2)  # optional - magma
(RModule(IntegerRing(), 2),
```

```
 Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 2))
sage: magma.bar_call(V, 'quo', V, nvals=2)          # optional - magma
(RModule(IntegerRing(), 0),
 Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 0))
```

chdir(*dir*)

Change to the given directory.

INPUT:

- `dir` - string; name of a directory

EXAMPLES:

```
sage: magma.chdir('/')                  # optional - magma
sage: magma.eval('System("pwd")')       # optional - magma
'/'
```

clear(*var*)

Clear the variable named var and make it available to be used again.

INPUT:

- `var` - a string

EXAMPLES:

```
sage: magma = Magma()       # optional - magma
sage: magma.clear('foo')    # sets foo to 0 in magma; optional - magma
sage: magma.eval('foo')     # optional - magma
'0'
```

Because we cleared foo, it is set to be used as a variable name in the future:

```
sage: a = magma('10')       # optional - magma
sage: a.name()              # optional - magma
'foo'
```

The following tests that the whole variable clearing and freeing system is working correctly.

```
sage: magma = Magma()       # optional - magma
sage: a = magma('100')      # optional - magma
sage: a.name()              # optional - magma
'_sage_[1]'
sage: del a                 # optional - magma
sage: b = magma('257')      # optional - magma
sage: b.name()              # optional - magma
'_sage_[1]'
sage: del b                 # optional - magma
sage: magma('_sage_[1]')    # optional - magma
0
```

console()

Run a command line Magma session. This session is completely separate from this Magma interface.

EXAMPLES:

```
sage: magma.console()             # not tested
Magma V2.14-9     Sat Oct 11 2008 06:36:41 on one      [Seed = 1157408761]
Type ? for help.  Type <Ctrl>-D to quit.
>
Total time: 2.820 seconds, Total memory usage: 3.95MB
```

cputime(*t=None*)

Return the CPU time in seconds that has elapsed since this Magma session started. This is a floating point number, computed by Magma.

If t is given, then instead return the floating point time from when t seconds had elapsed. This is useful for computing elapsed times between two points in a running program.

INPUT:

- `t` - float (default: None); if not None, return cputime since t

OUTPUT:

- `float` - seconds

EXAMPLES:

```
sage: type(magma.cputime())          # optional - magma
<type 'float'>
sage: magma.cputime()                 # random, optional - magma
1.9399999999999999
sage: t = magma.cputime()             # optional - magma
sage: magma.cputime(t)                # random, optional - magma
0.02
```

eval(*x, strip=True, **kwds*)

Evaluate the given block x of code in Magma and return the output as a string.

INPUT:

- `x` - string of code
- `strip` - ignored

OUTPUT: string

EXAMPLES: We evaluate a string that involves assigning to a variable and printing.

```
sage: magma.eval("a := 10;print 2+a;")      # optional - magma
'12'
```

We evaluate a large input line (note that no weird output appears and that this works quickly).

```
sage: magma.eval("a := %s;"%(10^10000))    # optional - magma
"
```

function_call(*function, args=, [], params={}, nvals=1*)

Return result of evaluating a Magma function with given input, parameters, and asking for nvals as output.

INPUT:

- `function` - string, a Magma function name
- `args` - list of objects coercible into this magma interface
- `params` - Magma parameters, passed in aftera colon
- `nvals` - number of return values from the function to ask Magma for

OUTPUT: MagmaElement or tuple of nvals MagmaElement's

EXAMPLES:

```
sage: magma.function_call('Factorization', 100)    # optional - magma
[ <2, 2>, <5, 2> ]
sage: magma.function_call('NextPrime', 100, {'Proof':False})    # optional - magma
101
sage: magma.function_call('PolynomialRing', [QQ,2])      # optional - magma
Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: $.1, $.2
```

Next, we illustrate multiple return values:

```
sage: magma.function_call('IsSquare', 100)          # optional - magma
true
sage: magma.function_call('IsSquare', 100, nvals=2)     # optional - magma
```

```
(true, 10)
sage: magma.function_call('IsSquare', 100, nvals=3)      # optional - magma
...
RuntimeError: Error evaluating Magma code...
Runtime error in :=: Expected to assign 3 value(s) but only computed 2 value(s)
```

get(*var*)

Get the value of the variable var.

INPUT:

- `var` - string; name of a variable defined in the Magma session

OUTPUT:

- `string` - string representation of the value of the variable.

EXAMPLES:

```
sage: magma.set('abc', '2 + 3/5')      # optional - magma
sage: magma.get('abc')                 # optional - magma
'13/5'
```

get_verbose(*type*)

Get the verbosity level of a given algorithm class etc. in Magma.

INPUT:

- `type` - string (e.g. 'Groebner'), see Magma documentation

EXAMPLES:

```
sage: magma.set_verbose("Groebner", 2)         # optional - magma
sage: magma.get_verbose("Groebner")            # optional - magma
2
```

help(*s*)

Return Magma help on string s.

This returns what typing ?s would return in Magma.

INPUT:

- `s` - string

OUTPUT: string

EXAMPLES:

```
sage: magma.help("NextPrime")        # optional - magma
================================================================================
PATH: /magma/ring-field-algebra/integer/prime/next-previous/NextPrime
KIND: Intrinsic
================================================================================
NextPrime(n) : RngIntElt -> RngIntElt
NextPrime(n: parameter) : RngIntElt -> RngIntElt
...
```

ideal(*L*)

Return the Magma ideal defined by L.

INPUT:

- `L` - a list of elements of a Sage multivariate polynomial ring.

OUTPUT: The magma ideal generated by the elements of L.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: magma.ideal([x^2, y^3*x])          # optional - magma
Ideal of Polynomial ring of rank 2 over Rational Field
Graded Reverse Lexicographical Order
Variables: x, y
Basis:
[
x^2,
x*y^3
]
```

load (*filename*)

Load the file with given filename using the 'load' command in the Magma shell.

Loading a file in Magma makes all the functions and procedures in the file available. The file should not contain any intrinsics (or you'll get errors). It also runs code in the file, which can produce output.

INPUT:

- `filename` - string

OUTPUT: output printed when loading the file

EXAMPLES:

```
sage: open(SAGE_TMP + 'a.m','w').write('function f(n) return n^2; end function;\nprint "hi";
sage: print magma.load(SAGE_TMP + 'a.m')       # optional - magma
Loading ".../.sage//temp/.../a.m"
hi
sage: magma('f(12)')        # optional - magma
144
```

objgens (*value, gens*)

Create a new object with given value and gens.

INPUT:

- `value` - something coercible to an element of this Magma interface
- `gens` - string; command separated list of variable names

OUTPUT: new Magma element that is equal to value with given gens

EXAMPLES:

```
sage: R = magma.objgens('PolynomialRing(Rationals(),2)', 'alpha,beta')     # optional - magma
sage: R.gens()          # optional - magma
[alpha, beta]
```

Because of how Magma works you can use this to change the variable names of the generators of an object:

```
sage: S = magma.objgens(R, 'X,Y')          # optional - magma
sage: R                                    # optional - magma
Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: X, Y
sage: S                                    # optional - magma
Polynomial ring of rank 2 over Rational Field
Lexicographical Order
Variables: X, Y
```

set (*var, value*)

Set the variable var to the given value in the Magma interpreter.

INPUT:

- `var` - string; a variable name
- `value` - string; what to set var equal to

EXAMPLES:

```
sage: magma.set('abc', '2 + 3/5')        # optional - magma
sage: magma('abc')                       # optional - magma
13/5
```

set_verbose(*type, level*)

Set the verbosity level for a given algorithm, class, etc. in Magma.

INPUT:

- `type` - string (e.g. 'Groebner')
- `level` - integer = 0

EXAMPLES:

```
sage: magma.set_verbose("Groebner", 2)       # optional - magma
sage: magma.get_verbose("Groebner")          # optional - magma
2
```

trait_names(*verbose=True, use_disk_cache=True*)

Return a list of all Magma commands.

This is used as a hook to enable custom command completion.

Magma doesn't provide any fast way to make a list of all commands, which is why caching is done by default. Note that an adverse impact of caching is that *new* commands are not picked up, e.g., user defined variables or functions.

INPUT:

- `verbose` - bool (default: True); whether to verbosely output status info the first time the command list is built
- `use_disk_cache` - bool (default: True); use cached command list, which is saved to disk.

OUTPUT: list of strings

EXAMPLES:

```
sage: len(magma.trait_names(verbose=False))     # random, optional - magma
7261
```

version()

Return the version of Magma that you have in your PATH on your computer.

OUTPUT:

- `numbers` - 3-tuple: major, minor, etc.
- `string` - version as a string

EXAMPLES:

```
sage: magma.version()        # random, optional - magma
((2, 14, 9), 'V2.14-9')
```

class **MagmaElement**(*parent, value, is_name=False, name=None*)

AssignNames(*names*)

EXAMPLES:

```
sage: G = magma.DirichletGroup(20)    # optional - magma
sage: G.AssignNames(['a','b'])        # optional - magma
sage: G.1                             # optional - magma
a
```

```
sage: G.Elements()                    # optional - magma
[
1,
a,
b,
a*b
]
```

assign_names(*names*)

EXAMPLES:

```
sage: G = magma.DirichletGroup(20)    # optional - magma
sage: G.AssignNames(['a','b'])        # optional - magma
sage: G.1                             # optional - magma
a

sage: G.Elements()                    # optional - magma
[
1,
a,
b,
a*b
]
```

eval(**args*)

Evalute self at the inputs.

INPUT:

- `*args` - import arguments

OUTPUT: self(*args)

EXAMPLES:

```
sage: f = magma('Factorization')            # optional - magma
sage: f.evaluate(15)                        # optional - magma
[ <3, 1>, <5, 1> ]
sage: f(15)                                 # optional - magma
[ <3, 1>, <5, 1> ]
sage: f = magma('GCD')                      # optional - magma
sage: f.evaluate(15,20)                     # optional - magma
5
```

evaluate(**args*)

Evalute self at the inputs.

INPUT:

- `*args` - import arguments

OUTPUT: self(*args)

EXAMPLES:

```
sage: f = magma('Factorization')            # optional - magma
sage: f.evaluate(15)                        # optional - magma
[ <3, 1>, <5, 1> ]
sage: f(15)                                 # optional - magma
[ <3, 1>, <5, 1> ]
sage: f = magma('GCD')                      # optional - magma
sage: f.evaluate(15,20)                     # optional - magma
5
```

gen(*n*)

Return the n-th generator of this Magma element. Note that generators are 1-based in Magma rather than 0 based!

INPUT:

- n - a *positive* integer

OUTPUT: MagmaElement

EXAMPLES:

```
sage: k.<a> = GF(9)
sage: magma(k).gen(1)          # optional -- magma
a
sage: R.<s,t,w> = k[]
sage: m = magma(R)             # optional -- magma
sage: m.gen(1)                 # optional -- magma
s
sage: m.gen(2)                 # optional -- magma
t
sage: m.gen(3)                 # optional -- magma
w
sage: m.gen(0)                 # optional -- magma
...
IndexError: index must be positive since Magma indexes are 1-based
sage: m.gen(4)                 # optional -- magma
...
IndexError: list index out of range
```

gen_names()

Return list of Magma variable names of the generators of self.

Note: As illustrated below, these are not the print names of the the generators of the Magma object, but special variable names in the Magma session that reference the generators.

EXAMPLES:

```
sage: R.<x,zw> = QQ[]
sage: S = magma(R)                # optional - magma
sage: S.gen_names()               # optional - magma
('_sage_[...]', '_sage_[...]')
sage: magma(S.gen_names()[1])     # optional - magma
zw
```

gens()

Return generators for self.

If self is named X is Magma, this function evaluates X.1, X.2, etc., in Magma until an error occurs. It then returns a Sage list of the resulting X.i. Note - I don't think there is a Magma command that returns the list of valid X.i. There are numerous ad hoc functions for various classes but nothing systematic. This function gets around that problem. Again, this is something that should probably be reported to the Magma group and fixed there.

AUTHORS:

- William Stein (2006-07-02)

EXAMPLES:

```
sage: magma("VectorSpace(RationalField(),3)").gens()          # optional - magma
[(1 0 0), (0 1 0), (0 0 1)]
sage: magma("AbelianGroup(EllipticCurve([1..5]))").gens()     # optional - magma
[$.1]
```

get_magma_attribute(*attrname*)

Return value of a given Magma attribute. This is like selfattrname in Magma.

OUTPUT: MagmaElement

EXAMPLES:

```
sage: V = magma("VectorSpace(RationalField(),10)")    # optional - magma
sage: V.set_magma_attribute('M','"hello"')            # optional - magma
sage: V.get_magma_attribute('M')                      # optional - magma
hello
sage: V.M                                             # optional - magma
hello
```

ideal (*gens*)

Return the ideal of self with given list of generators.

INPUT:

- gens - object or list/tuple of generators

OUTPUT:

- magma element - a Magma ideal

EXAMPLES:

```
sage: R = magma('PolynomialRing(RationalField())')        # optional - magma
sage: R.assign_names(['x'])                               # optional - magma
sage: x = R.1                                             # optional - magma
sage: R.ideal([x^2 - 1, x^3 - 1])                         # optional - magma
Ideal of Univariate Polynomial Ring in x over Rational Field generated by x - 1
```

list_attributes ()

Return the attributes of self, obtained by calling the ListAttributes function in Magma.

OUTPUT: list of strings

EXAMPLES: We observe that vector spaces in Magma have numerous funny and mysterious attributes.

```
sage: V = magma("VectorSpace(RationalField(),2)")        # optional - magma
sage: V.list_attributes()                                # optional - magma
['Coroots', 'Roots', 'decomp', 'ssbasis', 'M', 'StrLocalData', 'eisen', 'weights', 'RootDatu
```

methods (*any=False*)

Return signatures of all Magma intrinsics that can take self as the first argument, as strings.

INPUT:

- any - (bool: default is False) if True, also include signatures with Any as first argument.

OUTPUT: list of strings

EXAMPLES:

```
sage: v = magma('2/3').methods()          # optional - magma
sage: v[0]                                # optional - magma
"'*'..."
```

quo (*gens*)

Return the quotient of self by the given object or list of generators.

INPUT:

- gens - object or list/tuple of generators

OUTPUT:

- magma element - the quotient object
- magma element - mapping from self to the quotient object

EXAMPLES:

```
sage: V = magma('VectorSpace(RationalField(),3)')       # optional - magma
sage: V.quo([[1,2,3], [1,1,2]])                         # optional - magma
(Full Vector space of degree 1 over Rational Field, Mapping from: Full Vector space of degre
```

We illustrate quotienting out by an object instead of a list of generators:

```
sage: W = V.sub([ [1,2,3], [1,1,2] ])                   # optional - magma
sage: V.quo(W)                                          # optional - magma
(Full Vector space of degree 1 over Rational Field, Mapping from: Full Vector space of degre
```

We quotient a ZZ module out by a submodule.

```
sage: V = magma.RModule(ZZ,3); V # optional - magma
RModule(IntegerRing(), 3)
sage: W, phi = V.quo([[1,2,3]])  # optional - magma
sage: W                          # optional - magma
RModule(IntegerRing(), 2)
sage: phi                        # optional - magma
Mapping from: RModule(IntegerRing(), 3) to RModule(IntegerRing(), 2)
```

set_magma_attribute(*attrname, value*)

INPUTS: attrname - string value - something coercible to a MagmaElement

EXAMPLES:

```
sage: V = magma("VectorSpace(RationalField(),2)")   # optional - magma
sage: V.set_magma_attribute('M',10)                 # optional - magma
sage: V.get_magma_attribute('M')                    # optional - magma
10
sage: V.M                                           # optional - magma
10
```

sub(*gens*)

Return the sub-object of self with given gens.

INPUT:

- `gens` - object or list/tuple of generators

EXAMPLES:

```
sage: V = magma('VectorSpace(RationalField(),3)')       # optional - magma
sage: W = V.sub([ [1,2,3], [1,1,2] ]); W                # optional - magma
Vector space of degree 3, dimension 2 over Rational Field
Generators:
(1 2 3)
(1 1 2)
Echelonized basis:
(1 0 1)
(0 1 1)
```

trait_names()

Return all Magma functions that have this Magma element as first input. This is used for tab completion.

Note: This function can unfortunately be slow if there are a very large number of functions, e.g., when self is an integer. (This could be fixed by the addition of an appropriate function to the Magma kernel, which is something that can only be done by the Magma developers.)

OUTPUT:

- `list` - sorted list of distinct strings

EXAMPLES:

```
sage: v = magma('2/3').trait_names()        # optional - magma
sage: type(v[0])                            # optional - magma
<type 'str'>
```

class MagmaFunction(*parent, name*)

class MagmaFunctionElement(*obj, name*)

extcode_dir()

Return directory that contains all the Magma extcode. This is put in a writable directory owned by the user, since when attached, Magma has to write sig and lck files.

EXAMPLES: sage: sage.interfaces.magma.extcode_dir() '...dir_.../data/'

is_MagmaElement(*x*)

Return True if x is of type MagmaElement, and False otherwise.

INPUT:

- x - any object

OUTPUT: bool

EXAMPLES:

```
sage: from sage.interfaces.magma import is_MagmaElement
sage: is_MagmaElement(2)
False
sage: is_MagmaElement(magma(2))                    # optional - magma
True
```

magma_console()

Run a command line Magma session.

EXAMPLES:

```
sage: magma_console()             # not tested
Magma V2.14-9     Sat Oct 11 2008 06:36:41 on one      [Seed = 1157408761]
Type ? for help.  Type <Ctrl>-D to quit.
>
Total time: 2.820 seconds, Total memory usage: 3.95MB
```

magma_version()

Return the version of Magma that you have in your PATH on your computer.

OUTPUT:

- numbers - 3-tuple: major, minor, etc.
- string - version as a string

EXAMPLES:

```
sage: magma_version()       # random, optional - magma
((2, 14, 9), 'V2.14-9')
```

reduce_load_Magma()

Used in unpickling a Magma interface.

This functions just returns the global default Magma interface.

EXAMPLES:

```
sage: sage.interfaces.magma.reduce_load_Magma()
Magma
```

13.8 Interface to Maple

AUTHORS:

- William Stein (2005): maple interface
- Gregg Musiker (2006-02-02): tutorial
- William Stein (2006-03-05): added tab completion, e.g., maple.[tab], and help, e.g, maple.sin?.

You must have the optional commercial Maple interpreter installed and available as the command `maple` in your PATH in order to use this interface. You do not have to install any optional Sage packages.

Type `maple.[tab]` for a list of all the functions available from your Maple install. Type `maple.[tab]?` for Maple's help about a given function. Type `maple(...)` to create a new Maple object, and `maple.eval(...)` to run a string using Maple (and get the result back as a string).

EXAMPLES:

```
sage: maple('3 * 5')                                  # optional - maple
15
sage: maple.eval('ifactor(2005)')                     # optional - maple
'"(5)*"(401)'
sage: maple.ifactor(2005)                             # optional - maple
"(5)*"(401)
sage: maple.fsolve('x^2=cos(x)+4', 'x=0..5')          # optional - maple
1.914020619
sage: maple.factor('x^5 - y^5')                       # optional - maple
(x-y)*(x^4+x^3*y+x^2*y^2+x*y^3+y^4)
```

If the string "error" (case insensitive) occurs in the output of anything from Maple, a RuntimeError exception is raised.

13.8.1 Tutorial

AUTHORS:

- Gregg Musiker (2006-02-02): initial version.

This tutorial is based on the Maple Tutorial for number theory from http://www.math.mun.ca/~drideout/m3370/numtheory.html.

There are several ways to use the Maple Interface in Sage. We will discuss two of those ways in this tutorial.

1. If you have a maple expression such as

   ```
   factor( (x^5-1));
   ```

   We can write that in sage as

```
sage: maple('factor(x^5-1)')                    # optional - maple
(x-1)*(x^4+x^3+x^2+x+1)
```

Notice, there is no need to use a semicolon.

2. Since Sage is written in Python, we can also import maple commands and write our scripts in a pythonic way. For example, `factor()` is a maple command, so we can also factor in Sage using

```
sage: maple('(x^5-1)').factor()                 # optional - maple
(x-1)*(x^4+x^3+x^2+x+1)
```

where `expression.command()` means the same thing as `command(expression)` in Maple. We will use this second type of syntax whenever possible, resorting to the first when needed.

```
sage: maple('(x^12-1)/(x-1)').simplify()      # optional - maple
x^11+x^10+x^9+x^8+x^7+x^6+x^5+x^4+x^3+x^2+x+1
```

The normal command will always reduce a rational function to the lowest terms. The factor command will factor a polynomial with rational coefficients into irreducible factors over the ring of integers. So for example,

```
sage: maple('(x^12-1)').factor( )             # optional - maple
(x-1)*(x+1)*(x^2+x+1)*(x^2-x+1)*(x^2+1)*(x^4-x^2+1)
```

```
sage: maple('(x^28-1)').factor( )             # optional - maple
(x-1)*(x^6+x^5+x^4+x^3+x^2+x+1)*(x+1)*(1-x+x^2-x^3+x^4-x^5+x^6)*(x^2+1)*(x^12-x^10+x^8-x^6+x^4-x^2+1)
```

Another important feature of maple is its online help. We can access this through sage as well. After reading the description of the command, you can press q to immediately get back to your original prompt.

Incidentally you can always get into a maple console by the command

```
sage:  maple.console()           # not tested
sage:  !maple                    # not tested
```

Note that the above two commands are slightly different, and the first is preferred.

For example, for help on the maple command fibonacci, we type

```
sage: maple.help('fibonacci')   # not tested, since it uses a pager
```

We see there are two choices. Type

```
sage: maple.help('combinat, fibonacci')    # not tested, since it uses a pager
```

We now see how the Maple command fibonacci works under the combinatorics package. Try typing in

```
sage: maple.fibonacci(10)                 # optional - maple
fibonacci(10)
```

You will get fibonacci(10) as output since Maple has not loaded the combinatorics package yet. To rectify this type

```
sage: maple('combinat[fibonacci]')(10)      # optional - maple
55
```

instead.

If you want to load the combinatorics package for future calculations, in Sage this can be done as

```
sage: maple.with_package('combinat')        # optional - maple
```

or

```
sage: maple.load('combinat')                # optional - maple
```

Now if we type `maple.fibonacci(10)`, we get the correct output:

```
sage: maple.fibonacci(10)                   # optional - maple
55
```

Some common maple packages include `combinat`, `linalg`, and `numtheory`. To produce the first 19 Fibonacci numbers, use the sequence command.

```
sage: maple('seq(fibonacci(i),i=1..19)')      # optional - maple
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584,
4181
```

Two other useful Maple commands are ifactor and isprime. For example

```
sage: maple.isprime(maple.fibonacci(27))      # optional - maple
false
sage: maple.ifactor(maple.fibonacci(27))      # optional - maple
"(2)*"(17)*"(53)*"(109)
```

Note that the isprime function that is included with Sage (which uses PARI) is better than the Maple one (it is faster and gives a provably correct answer, whereas Maple is sometimes wrong).

```
sage: alpha = maple('(1+sqrt(5))/2')          # optional - maple
sage: beta = maple('(1-sqrt(5))/2')           # optional - maple
sage: f19  = alpha^19 - beta^19/maple('sqrt(5)')      # optional - maple
sage: f19                                             # optional - maple
(1/2+1/2*5^(1/2))^19-1/5*(1/2-1/2*5^(1/2))^19*5^(1/2)
sage: f19.simplify()                 # somewhat randomly ordered output; optional - maple
6765+5778/5*5^(1/2)
```

Let's say we want to write a maple program now that squares a number if it is positive and cubes it if it is negative. In maple, that would look like

```
mysqcu := proc(x)
if x > 0 then x^2;
else x^3; fi;
end;
```

In Sage, we write

```
sage: mysqcu = maple('proc(x) if x > 0 then x^2 else x^3 fi end')     # optional - maple
sage: mysqcu(5)                                                       # optional - maple
25
sage: mysqcu(-5)                                                      # optional - maple
-125
```

More complicated programs should be put in a separate file and loaded.

class Maple (*maxread=100, script_subdirectory='', server=None, server_tmpdir=None, logfile=None*)
Interface to the Maple interpreter.

Type `maple.[tab]` for a list of all the functions available from your Maple install. Type `maple.[tab]?` for Maple's help about a given function. Type `maple(...)` to create a new Maple object, and `maple.eval(...)` to run a string using Maple (and get the result back as a string).

clear (*var*)
Clear the variable named var.

Unfortunately, Maple does not have a clear command. The next best thing is to set equal to the constant 0, so that memory will be freed.

EXAMPLES:

```
sage: maple.set('xx', '2')  # optional - maple
sage: maple.get('xx')       # optional - maple
'2'
sage: maple.clear('xx')     # optional - maple
sage: maple.get('xx')       # optional - maple
'0'
```

completions (*s*)
Return all commands that complete the command starting with the string s. This is like typing s[Ctrl-T] in the maple interpreter.

EXAMPLES:

```
sage: c = maple.completions('di')  # optional - maple
sage: 'divide' in c                # optional - maple
True
```

console ()
Spawn a new Maple command-line session.

EXAMPLES:

```
sage: maple.console() # not tested
    |^/|     Maple 11 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2007
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
>
```

cputime (*t=None*)
Returns the amount of CPU time that the Maple session has used. If `t` is not None, then it returns the difference between the current CPU time and `t`.

EXAMPLES:

```
sage: t = maple.cputime() # optional - maple
sage: t                   # random; optional - maple
0.02
sage: x = maple('x')      # optional - maple
sage: maple.diff(x^2, x)  # optional - maple
2*x
sage: maple.cputime(t)    # random; optional - maple
0.0
```

expect ()
Returns the pexpect object for this Maple session.

EXAMPLES:

```
sage: m = Maple()
sage: m.expect() is None
True
sage: m._start()           # optional - maple
sage: m.expect()           # optional - maple
<pexpect.spawn instance at 0x...>
sage: m.quit()             # optional - maple
```

get(*var*)

Get the value of the variable var.

EXAMPLES:

```
sage: maple.set('xx', '2') # optional - maple
sage: maple.get('xx')      # optional - maple
'2'
```

help(*str*)

Display Maple help about str. This is the same as typing "?str" in the Maple console.

INPUT:

- `str` - a string to search for in the maple help system

EXAMPLES:

```
sage: maple.help('digamma') #not tested
Psi - the Digamma and Polygamma functions
...
```

load(*package*)

Make a package of Maple procedures available in the interpreter.

INPUT:

- `package` - string

EXAMPLES: Some functions are unknown to Maple until you use with to include the appropriate package.

```
sage: maple.quit()    # optional -- to reset maple.
sage: maple('partition(10)')              # optional -- requires maple
partition(10)
sage: maple('bell(10)')                   # optional -- requires maple
bell(10)
sage: maple.with_package('combinat')      # optional -- requires maple
sage: maple('partition(10)')              # optional -- requires maple
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 2], [1, 1, 1, 1, 1, 1, 2, 2], [1,
sage: maple('bell(10)')                   # optional -- requires maple
115975
sage: maple('fibonacci(10)')              # optional -- requires maple
55
```

set(*var, value*)

Set the variable var to the given value.

EXAMPLES:

```
sage: maple.set('xx', '2') # optional - maple
sage: maple.get('xx')      # optional - maple
'2'
```

source(*s*)

Display the Maple source (if possible) about s. This is the same as returning the output produced by the following Maple commands:

interface(verboseproc=2): print(s)

INPUT:

- s - a string representing the function whose source code you want

EXAMPLES:

```
sage: maple.source('curry')  #not tested
p -> subs('_X' = args[2 .. nargs], () -> p(_X, args))
```

trait_names(*verbose=True, use_disk_cache=True*)

Returns a list of all the commands defined in Maple and optionally (per default) store them to disk.

EXAMPLES:

```
sage: c = maple.trait_names(use_disk_cache=False, verbose=False) # optional - maple
sage: len(c) > 100  # optional - maple
True
sage: 'dilog' in c  # optional - maple
True
```

with_package(*package*)

Make a package of Maple procedures available in the interpreter.

INPUT:

- package - string

EXAMPLES: Some functions are unknown to Maple until you use with to include the appropriate package.

```
sage: maple.quit()   # optional -- to reset maple.
sage: maple('partition(10)')              # optional -- requires maple
partition(10)
sage: maple('bell(10)')                   # optional -- requires maple
bell(10)
sage: maple.with_package('combinat')      # optional -- requires maple
sage: maple('partition(10)')              # optional -- requires maple
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 2], [1, 1, 1, 1, 1, 1, 2, 2], [1,
sage: maple('bell(10)')                   # optional -- requires maple
115975
sage: maple('fibonacci(10)')              # optional -- requires maple
55
```

class MapleElement(*parent, value, is_name=False, name=None*)

trait_names()

EXAMPLES:

```
sage: a = maple(2) # optional - maple
sage: 'sin' in a.trait_names() # optional - maple
True
```

class MapleFunction(*parent, name*)

class MapleFunctionElement(*obj, name*)

maple_console()

Spawn a new Maple command-line session.

EXAMPLES:

```
sage: maple_console() #not tested
    |^/|     Maple 11 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) Maplesoft, a division of Waterloo Maple Inc. 2007
 \  MAPLE  /  All rights reserved. Maple is a trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
>
```

reduce_load_Maple()
Returns the maple object created in sage.interfaces.maple.

EXAMPLES:

```
sage: from sage.interfaces.maple import reduce_load_Maple
sage: reduce_load_Maple()
Maple
```

13.9 Interface to MATLAB

According to their website, MATLAB is "a high-level language and interactive environment that enables you to perform computationally intensive tasks faster than with traditional programming languages such as C, C++, and Fortran."

The commands in this section only work if you have the "matlab" interpreter installed and available in your PATH. It's not necessary to install any special Sage packages.

EXAMPLES:

```
sage: matlab.eval('2+2')                    # optional
'\nans =\n\n     4\n'


sage: a = matlab(10)                        # optional
sage: a**10                                 # optional
   1.0000e+10
```

AUTHORS:

- William Stein (2006-10-11)

13.9.1 Tutorial

EXAMPLES:

```
sage: matlab('4+10')                        # optional
14
sage: matlab('date')                       # optional; random output
18-Oct-2006
sage: matlab('5*10 + 6')                    # optional
56
sage: matlab('(6+6)/3')                     # optional
4
sage: matlab('9')^2                         # optional
81
sage: a = matlab(10); b = matlab(20); c = matlab(30)     # optional
sage: avg = (a+b+c)/3                       # optional
sage: avg                                   # optional
20
sage: parent(avg)                           # optional
Matlab
```

```
sage: my_scalar = matlab('3.1415')        # optional
sage: my_scalar                           # optional
3.1415
sage: my_vector1 = matlab('[1,5,7]')      # optional
sage: my_vector1                          # optional
1     5     7
sage: my_vector2 = matlab('[1;5;7]')      # optional
sage: my_vector2                          # optional
1
5
7
sage: my_vector1 * my_vector2             # optional
75


sage: row_vector1 = matlab('[1 2 3]')              # optional
sage: row_vector2 = matlab('[3 2 1]')              # optional
sage: matrix_from_row_vec = matlab('[%s; %s]'%(row_vector1.name(), row_vector2.name()))     # optiona
sage: matrix_from_row_vec                          # optional
1     2     3
3     2     1


sage: column_vector1 = matlab('[1;3]')             # optional
sage: column_vector2 = matlab('[2;8]')             # optional
sage: matrix_from_col_vec = matlab('[%s %s]'%(column_vector1.name(), column_vector2.name()))
sage: matrix_from_col_vec                          # optional
1     2
3     8


sage: my_matrix = matlab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')     # optional
sage: my_matrix                                    # optional
     8    12    19
     7     3     2
    12     4    23
     8     1     1


sage: combined_matrix = matlab('[%s, %s]'%(my_matrix.name(), my_matrix.name()))
sage: combined_matrix                             # optional
 8    12    19     8    12    19
 7     3     2     7     3     2
12     4    23    12     4    23
 8     1     1     8     1     1


sage: tm = matlab('0.5:2:10')                     # optional
sage: tm                                          # optional
0.5000    2.5000    4.5000    6.5000    8.5000


sage: my_vector1 = matlab('[1,5,7]')              # optional
sage: my_vector1(1)                               # optional
1
sage: my_vector1(2)                               # optional
5
sage: my_vector1(3)                               # optional
7
```

Matrix indexing works as follows:

```
sage: my_matrix = matlab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')      # optional
sage: my_matrix(3,2)                                  # optional
4
```

Setting using paranthesis cannot work (because of how the Python language works). Use square brackets or the set function:

```
sage: my_matrix = matlab('[8, 12, 19; 7, 3, 2; 12, 4, 23; 8, 1, 1]')     # optional
sage: my_matrix.set(2,3, 1999)                            # optional
sage: my_matrix                                           # optional
           8          12          19
           7           3        1999
          12           4          23
           8           1           1
```

class Matlab(*maxread=100, script_subdirectory='', logfile=None, server=None, server_tmpdir=None*)

Interface to the Matlab interpreter.

EXAMPLES:

```
sage: a = matlab('[ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]')     # optional
sage: b = matlab('[ 1; 3; 13]')                           # optional
sage: c = a * b                                           # optional
sage: print c                                             # optional
    30
   122
   505
```

console()

get(*var*)

Get the value of the variable var.

sage2matlab_matrix_string(*A*)

Return an matlab matrix from a Sage matrix.

INPUT: A Sage matrix with entries in the rationals or reals.

OUTPUT: A string that evaluates to an Matlab matrix.

EXAMPLES:

```
sage: M33 = MatrixSpace(QQ,3,3)
sage: A = M33([1,2,3,4,5,6,7,8,0])
sage: matlab.sage2matlab_matrix_string(A)    # requires optional matlab
'[1, 2, 3; 4, 5, 6; 7, 8, 0]'
```

AUTHOR:

- David Joyner and William Stein

set(*var, value*)

Set the variable var to the given value.

version()

whos()

class MatlabElement(*parent, value, is_name=False, name=None*)

set(*i, j, x*)

matlab_console()
This requires that the optional matlab program be installed and in your PATH, but no optional Sage packages need be installed.

EXAMPLES:

```
sage: matlab_console()                                  # optional and not tested
                                  < M A T L A B >
                    Copyright 1984-2006 The MathWorks, Inc.
...
>> 2+3

ans =

5

quit
```

Typing quit exits the matlab console and returns you to Sage. matlab, like Sage, remembers its history from one session to another.

matlab_version()
Return the version of Matlab installed.

EXAMPLES:

```
sage: matlab_version()    # optional matlab package
'7.2.0.283 (R2006a)'
```

reduce_load_Matlab()

13.10 Interface to Maxima

Maxima is a free GPL'd general purpose computer algebra system whose development started in 1968 at MIT. It contains symbolic manipulation algorithms, as well as implementations of special functions, including elliptic functions and generalized hypergeometric functions. Moreover, Maxima has implementations of many functions relating to the invariant theory of the symmetric group S_n. (However, the commands for group invariants, and the corresponding Maxima documenation, are in French.) For many links to Maxima documentation see http://maxima.sourceforge.net/docs.shtml/.

AUTHORS:

- William Stein (2005-12): Initial version
- David Joyner: Improved documentation
- William Stein (2006-01-08): Fixed bug in parsing
- William Stein (2006-02-22): comparisons (following suggestion of David Joyner)
- William Stein (2006-02-24): *greatly* improved robustness by adding sequence numbers to IO bracketing in _eval_line

If the string "error" (case insensitive) occurs in the output of anything from maxima, a RuntimeError exception is raised.

EXAMPLES: We evaluate a very simple expression in maxima.

```
sage: maxima('3 * 5')
15
```

We factor $x^5 - y^5$ in Maxima in several different ways. The first way yields a Maxima object.

```
sage: F = maxima.factor('x^5 - y^5')
sage: F
-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)
sage: type(F)
<class 'sage.interfaces.maxima.MaximaElement'>
```

Note that Maxima objects can also be displayed using "ASCII art"; to see a normal linear representation of any Maxima object x. Just use the print command: use `str(x)`.

```
sage: print F
                              4       3    2  2    3       4
                    - (y - x) (y  + x y  + x  y  + x  y + x )
```

You can always use `repr(x)` to obtain the linear representation of an object. This can be useful for moving maxima data to other systems.

```
sage: repr(F)
'-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)'
sage: F.str()
'-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)'
```

The `maxima.eval` command evaluates an expression in maxima and returns the result as a *string* not a maxima object.

```
sage: print maxima.eval('factor(x^5 - y^5)')
-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)
```

We can create the polynomial f as a Maxima polynomial, then call the factor method on it. Notice that the notation `f.factor()` is consistent with how the rest of Sage works.

```
sage: f = maxima('x^5 - y^5')
sage: f^2
(x^5-y^5)^2
sage: f.factor()
-(y-x)*(y^4+x*y^3+x^2*y^2+x^3*y+x^4)
```

Control-C interruption works well with the maxima interface, because of the excellent implementation of maxima. For example, try the following sum but with a much bigger range, and hit control-C.

```
sage: maxima('sum(1/x^2, x, 1, 10)')
1968329/1270080
```

13.10.1 Tutorial

We follow the tutorial at http://maxima.sourceforge.net/docs/intromax/.

```
sage: maxima('1/100 + 1/101')
201/10100
```

```
sage: a = maxima('(1 + sqrt(2))^5'); a
(sqrt(2)+1)^5
sage: a.expand()
29*sqrt(2)+41
```

```
sage: a = maxima('(1 + sqrt(2))^5')
sage: float(a)
82.012193308819747
sage: a.numer()
82.01219330881975
```

```
sage: maxima.eval('fpprec : 100')
'100'
sage: a.bfloat()
8.2012193308819756415248973002081244278520484385931494122123712401731241875401104126661238495501605
```

```
sage: maxima('100!')
933262154439441526816992388562667004907159682643816214685929638952175999932299156089414639761565182
```

```
sage: f = maxima('(x + 3*y + x^2*y)^3')
sage: f.expand()
x^6*y^3+9*x^4*y^3+27*x^2*y^3+27*y^3+3*x^5*y^2+18*x^3*y^2+27*x*y^2+3*x^4*y+9*x^2*y+x^3
sage: f.subst('x=5/z')
(5/z+25*y/z^2+3*y)^3
sage: g = f.subst('x=5/z')
sage: h = g.ratsimp(); h
(27*y^3*z^6+135*y^2*z^5+(675*y^3+225*y)*z^4+(2250*y^2+125)*z^3+(5625*y^3+1875*y)*z^2+9375*y^2*z+1562
sage: h.factor()
(3*y*z^2+5*z+25*y)^3/z^6
```

```
sage: eqn = maxima(['a+b*c=1', 'b-a*c=0', 'a+b=5'])
sage: s = eqn.solve('[a,b,c]'); s
[[a=(25*sqrt(79)*%i+25)/(6*sqrt(79)*%i-34),b=(5*sqrt(79)*%i+5)/(sqrt(79)*%i+11),c=(sqrt(79)*%i+1)/10]
```

Here is an example of solving an algebraic equation:

```
sage: maxima('x^2+y^2=1').solve('y')
[y=-sqrt(1-x^2),y=sqrt(1-x^2)]
sage: maxima('x^2 + y^2 = (x^2 - y^2)/sqrt(x^2 + y^2)').solve('y')
[y=-sqrt((-y^2-x^2)*sqrt(y^2+x^2)+x^2),y=sqrt((-y^2-x^2)*sqrt(y^2+x^2)+x^2)]
```

You can even nicely typeset the solution in latex:

```
sage: latex(s)
\left[ \left[ a={{25\,\sqrt{79}\,i+25}\over{6\,\sqrt{79}\,i-34}} ,   b={{5\,\sqrt{79}\,i+5}\over{\sqr
```

To have the above appear onscreen via `xdvi`, type `view(s)`. (TODO: For OS X should create pdf output and use preview instead?)

```
sage: e = maxima('sin(u + v) * cos(u)^3'); e
cos(u)^3*sin(v+u)
sage: f = e.trigexpand(); f
cos(u)^3*(cos(u)*sin(v)+sin(u)*cos(v))
```

```
sage: f.trigreduce()
(sin(v+4*u)+sin(v-2*u))/8+(3*sin(v+2*u)+3*sin(v))/8
sage: w = maxima('3 + k*%i')
sage: f = w^2 + maxima('%e')^w
sage: f.realpart()
%e^3*cos(k)-k^2+9


sage: f = maxima('x^3 * %e^(k*x) * sin(w*x)'); f
x^3*%e^(k*x)*sin(w*x)
sage: f.diff('x')
k*x^3*%e^(k*x)*sin(w*x)+3*x^2*%e^(k*x)*sin(w*x)+w*x^3*%e^(k*x)*cos(w*x)
sage: f.integrate('x')
(((k*w^6+3*k^3*w^4+3*k^5*w^2+k^7)*x^3+(3*w^6+3*k^2*w^4-3*k^4*w^2-3*k^6)*x^2+(-18*k*w^4-12*k^3*w^2+6*k


sage: f = maxima('1/x^2')
sage: f.integrate('x', 1, 'inf')
1
sage: g = maxima('f/sinh(k*x)^4')
sage: g.taylor('x', 0, 3)
f/(k^4*x^4)-2*f/(3*k^2*x^2)+11*f/45-62*k^2*f*x^2/945


sage: maxima.taylor('asin(x)','x',0, 10)
x+x^3/6+3*x^5/40+5*x^7/112+35*x^9/1152
```

13.10.2 Examples involving matrices

We illustrate computing with the matrix whose i, j entry is i/j, for $i, j = 1, \ldots, 4$.

```
sage: f = maxima.eval('f[i,j] := i/j')
sage: A = maxima('genmatrix(f,4,4)'); A
matrix([1,1/2,1/3,1/4],[2,1,2/3,1/2],[3,3/2,1,3/4],[4,2,4/3,1])
sage: A.determinant()
0
sage: A.echelon()
matrix([1,1/2,1/3,1/4],[0,0,0,0],[0,0,0,0],[0,0,0,0])
sage: A.eigenvalues()
[[0,4],[3,1]]
sage: A.eigenvectors()
[[[0,4],[3,1]],[1,0,0,-4],[0,1,0,-2],[0,0,1,-4/3],[1,2,3,4]]
```

We can also compute the echelon form in Sage:

```
sage: B = matrix(QQ, A)
sage: B.echelon_form()
[  1 1/2 1/3 1/4]
[  0   0   0   0]
[  0   0   0   0]
[  0   0   0   0]
sage: B.charpoly('x').factor()
(x - 4) * x^3
```

13.10.3 Laplace Transforms

We illustrate Laplace transforms:

```
sage: _ = maxima.eval("f(t) := t*sin(t)")
sage: maxima("laplace(f(t),t,s)")
2*s/(s^2+1)^2


sage: maxima("laplace(delta(t-3),t,s)") #Dirac delta function
%e^-(3*s)


sage: _ = maxima.eval("f(t) := exp(t)*sin(t)")
sage: maxima("laplace(f(t),t,s)")
1/(s^2-2*s+2)


sage: _ = maxima.eval("f(t) := t^5*exp(t)*sin(t)")
sage: maxima("laplace(f(t),t,s)")
360*(2*s-2)/(s^2-2*s+2)^4-480*(2*s-2)^3/(s^2-2*s+2)^5+120*(2*s-2)^5/(s^2-2*s+2)^6
sage: print maxima("laplace(f(t),t,s)")
                                      3                   5
           360 (2 s - 2)     480 (2 s - 2)      120 (2 s - 2)
          --------------- - --------------- + ---------------
            2            4     2            5     2            6
          (s  - 2 s + 2)     (s  - 2 s + 2)     (s  - 2 s + 2)


sage: maxima("laplace(diff(x(t),t),t,s)")
s*?%laplace(x(t),t,s)-x(0)


sage: maxima("laplace(diff(x(t),t,2),t,s)")
-?%at('diff(x(t),t,1),t=0)+s^2*?%laplace(x(t),t,s)-x(0)*s
```

It is difficult to read some of these without the 2d representation:

```
sage: print maxima("laplace(diff(x(t),t,2),t,s)")
                     !
           d         !          2
         - -- (x(t))!       + s  laplace(x(t), t, s) - x(0) s
           dt        !
                     !t = 0
```

Even better, use `view(maxima("laplace(diff(x(t),t,2),t,s)"))` to see a typeset version.

13.10.4 Continued Fractions

A continued fraction $a + 1/(b + 1/(c + \cdots))$ is represented in maxima by the list $[a, b, c, \ldots]$.

```
sage: maxima("cf((1 + sqrt(5))/2)")
[1,1,1,1,2]
sage: maxima("cf ((1 + sqrt(341))/2)")
[9,1,2,1,2,1,17,1,2,1,2,1,17,1,2,1,2,1,17,2]
```

13.10.5 Special examples

In this section we illustrate calculations that would be awkward to do (as far as I know) in non-symbolic computer algebra systems like MAGMA or GAP.

We compute the gcd of $2x^{n+4} - x^{n+2}$ and $4x^{n+1} + 3x^n$ for arbitrary n.

```
sage: f = maxima('2*x^(n+4) - x^(n+2)')
sage: g = maxima('4*x^(n+1) + 3*x^n')
sage: f.gcd(g)
x^n
```

You can plot 3d graphs (via gnuplot):

```
sage: maxima('plot3d(x^2-y^2, [x,-2,2], [y,-2,2], [grid,12,12])')  # not tested
[displays a 3 dimensional graph]
```

You can formally evaluate sums (note the `nusum` command):

```
sage: S = maxima('nusum(exp(1+2*i/n),i,1,n)')
sage: print S
                        2/n + 3                     2/n + 1
                      %e                          %e
                ---------------------- - ----------------------
                   1/n          1/n            1/n          1/n
                (%e    - 1) (%e    + 1)    (%e    - 1) (%e    + 1)
```

We formally compute the limit as $n \to \infty$ of $2S/n$ as follows:

```
sage: T = S*maxima('2/n')
sage: T.tlimit('n','inf')
%e^3-%e
```

13.10.6 Miscellaneous

Obtaining digits of π:

```
sage: maxima.eval('fpprec : 100')
'100'
sage: maxima(pi).bfloat()
3.14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706
```

Defining functions in maxima:

```
sage: maxima.eval('fun[a] := a^2')
'fun[a]:=a^2'
sage: maxima('fun[10]')
100
```

13.10.7 Interactivity

Unfortunately maxima doesn't seem to have a non-interactive mode, which is needed for the Sage interface. If any Sage call leads to maxima interactively answering questions, then the questions can't be answered and the maxima

session may hang. See the discussion at http://www.ma.utexas.edu/pipermail/maxima/2005/011061.html for some ideas about how to fix this problem. An example that illustrates this problem is `maxima.eval('integrate (exp(a*x), x, 0, inf)')`.

13.10.8 Latex Output

To tex a maxima object do this:

```
sage: latex(maxima('sin(u) + sinh(v^2)'))
\sinh v^2+\sin u
```

Here's another example:

```
sage: g = maxima('exp(3*%i*x)/(6*%i) + exp(%i*x)/(2*%i) + c')
sage: latex(g)
-{{i\,e^{3\,i\,x}}\over{6}}-{{i\,e^{i\,x}}\over{2}}+c
```

13.10.9 Long Input

The MAXIMA interface reads in even very long input (using files) in a robust manner, as long as you are creating a new object.

Note: Using `maxima.eval` for long input is much less robust, and is not recommended.

```
sage: t = '"%s"'%10^10000   # ten thousand character string.
sage: a = maxima(t)
```

TESTS: This working tests that a subtle bug has been fixed:

```
sage: f = maxima.function('x','gamma(x)')
sage: g = f(1/7)
sage: g
gamma(1/7)
sage: del f
sage: maxima(sin(x))
sin(x)
```

This tests to make sure we handle the case where Maxima asks if an expression is positive or zero.

```
sage: var('Ax,Bx,By')
(Ax, Bx, By)
sage: t = -Ax*sin(sqrt(Ax^2)/2)/(sqrt(Ax^2)*sqrt(By^2 + Bx^2))
sage: t.limit(Ax=0,dir='above')
...
TypeError: Computation failed since Maxima requested additional constraints (try the command 'assume
Is By^2+Bx^2  positive or zero?
```

A long complicated input expression:

```
sage: maxima._eval_line('((((((((((0) + ((1) / ((n0) ^ (0)))) + ((1) / ((n1) ^ (1)))) + ((1) / ((n2)
'1/n9^9+1/n8^8+1/n7^7+1/n6^6+1/n5^5+1/n4^4+1/n3^3+1/n2^2+1/n1+1'
```

class **Maxima** (*script_subdirectory=None, logfile=None, server=None, init_code=None*)
Interface to the Maxima interpreter.

chdir(*dir*)

Change Maxima's current working directory.

EXAMPLES:

```
sage: maxima.chdir('/')
```

clear(*var*)

Clear the variable named var.

EXAMPLES:

```
sage: maxima.set('xxxxx', '2')
sage: maxima.get('xxxxx')
'2'
sage: maxima.clear('xxxxx')
sage: maxima.get('xxxxx')
'xxxxx'
```

completions(*s, verbose=True*)

Return all commands that complete the command starting with the string s. This is like typing s[tab] in the Maxima interpreter.

EXAMPLES:

```
sage: sorted(maxima.completions('gc', verbose=False))
['gc', 'gcd', 'gcdex', 'gcfactor', 'gcprint', 'gctime']
```

console()

Start the interactive Maxima console. This is a completely separate maxima session from this interface. To interact with this session, you should instead use `maxima.interact()`.

EXAMPLES:

```
sage: maxima.console()              # not tested (since we can't)
Maxima 5.13.0 http://maxima.sourceforge.net
Using Lisp CLISP 2.41 (2006-10-13)
Distributed under the GNU Public License. See the file COPYING.
Dedicated to the memory of William Schelter.
This is a development version of Maxima. The function bug_report()
provides bug reporting information.
(%i1)

sage: maxima.interact()     # this is not tested either
  --> Switching to Maxima <--
maxima: 2+2
4
maxima:
  --> Exiting back to Sage <--
```

cputime(*t=None*)

Returns the amount of CPU time that this Maxima session has used. If var{t} is not None, then it returns the difference between the current CPU time and var{t}.

EXAMPLES: sage: t = maxima.cputime() sage: _ = maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'], [1,1,1]) sage: maxima.cputime(t) # output random 0.568913

de_solve(*maxima, de, vars, ics=None*)

Solves a 1st or 2nd order ordinary differential equation (ODE) in two variables, possibly with initial conditions.

INPUT:

- `de` - a string representing the ODE
- `vars` - a list of strings representing the two variables.

- ics - a triple of numbers [a,b1,b2] representing y(a)=b1, y'(a)=b2

EXAMPLES:

```
sage: maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'], [1,1,1])
y=3*x-2*%e^(x-1)
sage: maxima.de_solve('diff(y,x,2) + 3*x = y', ['x','y'])
y=%k1*%e^x+%k2*%e^-x+3*x
sage: maxima.de_solve('diff(y,x) + 3*x = y', ['x','y'])
y=(%c-3*(-x-1)*%e^-x)*%e^x
sage: maxima.de_solve('diff(y,x) + 3*x = y', ['x','y'],[1,1])
y=-%e^-1*(5*%e^x-3*%e*x-3*%e)
```

de_solve_laplace(*de, vars, ics=None*)

Solves an ordinary differential equation (ODE) using Laplace transforms.

INPUT:

- de - a string representing the ODE (e.g., de = "diff(f(x),x,2)=diff(f(x),x)+sin(x)")
- vars - a list of strings representing the variables (e.g., vars = ["x","f"])
- ics - a list of numbers representing initial conditions, with symbols allowed which are represented by strings (eg, f(0)=1, f'(0)=2 is ics = [0,1,2])

EXAMPLES:

```
sage: maxima.clear('x'); maxima.clear('f')
sage: maxima.de_solve_laplace("diff(f(x),x,2) = 2*diff(f(x),x)-f(x)", ["x","f"], [0,1,2])
f(x)=x*%e^x+%e^x

sage: maxima.clear('x'); maxima.clear('f')
sage: f = maxima.de_solve_laplace("diff(f(x),x,2) = 2*diff(f(x),x)-f(x)", ["x","f"])
sage: f
f(x)=x*%e^x*('at('diff(f(x),x,1),x=0))-f(0)*x*%e^x+f(0)*%e^x
sage: print f
                                    !
                        x  d        !                    x          x
            f(x) = x %e  (-- (f(x))!     ) - f(0) x %e  + f(0) %e
                         dx         !
                                    !x = 0
```

Note: The second equation sets the values of $f(0)$ and $f'(0)$ in Maxima, so subsequent ODEs involving these variables will have these initial conditions automatically imposed.

demo(*s*)

EXAMPLES:

```
sage: maxima.demo('array') # not tested
batching /opt/sage/local/share/maxima/5.16.3/demo/array.dem
```

At the _ prompt, type ';' followed by enter to get next demo subscrmap : true _

describe(*s*)

EXAMPLES:

```
sage: maxima.help('gcd')
-- Function: gcd (<p_1>, <p_2>, <x_1>, ...)
...
```

example(*s*)

EXAMPLES:

```
sage: maxima.example('arrays')
a[n]:=n*a[n-1]
                                  a  := n a
```

```
                                        n        n - 1
a[0]:1
a[5]
                                      120
a[n]:=n
a[6]
                                       6
a[4]
                                      24
                                     done
```

function(*args, defn, rep=None, latex=None*)

Return the Maxima function with given arguments and definition.

INPUT:

- `args` - a string with variable names separated by commas
- `defn` - a string (or Maxima expression) that defines a function of the arguments in Maxima.
- `rep` - an optional string; if given, this is how the function will print.

EXAMPLES:

```
sage: f = maxima.function('x', 'sin(x)')
sage: f(3.2)
-.05837414342758009
sage: f = maxima.function('x,y', 'sin(x)+cos(y)')
sage: f(2,3.5)
sin(2)-.9364566872907963
sage: f
sin(x)+cos(y)

sage: g = f.integrate('z')
sage: g
(cos(y)+sin(x))*z
sage: g(1,2,3)
3*(cos(2)+sin(1))
```

The function definition can be a maxima object:

```
sage: an_expr = maxima('sin(x)*gamma(x)')
sage: t = maxima.function('x', an_expr)
sage: t
gamma(x)*sin(x)
sage: t(2)
 sin(2)
sage: float(t(2))
0.90929742682568171
sage: loads(t.dumps())
gamma(x)*sin(x)
```

get(*var*)

Get the string value of the variable var.

EXAMPLES:

```
sage: maxima.set('xxxxx', '2')
sage: maxima.get('xxxxx')
'2'
```

help(*s*)

EXAMPLES:

```
sage: maxima.help('gcd')
-- Function: gcd (<p_1>, <p_2>, <x_1>, ...)
...
```

lisp(*cmd*)

Send a lisp command to maxima.

Note: The output of this command is very raw - not pretty.

EXAMPLES:

```
sage: maxima.lisp("(+ 2 17)")    # random formated output
 :lisp (+ 2 17)
19
(
```

plot2d(**args*)

Plot a 2d graph using Maxima / gnuplot.

maxima.plot2d(f, '[var, min, max]', options)

INPUT:

- `f` - a string representing a function (such as f="sin(x)") [var, xmin, xmax]
- `options` - an optional string representing plot2d options in gnuplot format

EXAMPLES:

```
sage: maxima.plot2d('sin(x)','[x,-5,5]')    # not tested
sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-plot.eps"]'
sage: maxima.plot2d('sin(x)','[x,-5,5]',opts)    # not tested
```

The eps file is saved in the current directory.

plot2d_parametric(*r, var, trange, nticks=50, options=None*)

Plots r = [x(t), y(t)] for t = tmin...tmax using gnuplot with options

INPUT:

- `r` - a string representing a function (such as r="[x(t),y(t)]")
- `var` - a string representing the variable (such as var = "t")
- `trange` - [tmin, tmax] are numbers with tmintmax
- `nticks` - int (default: 50)
- `options` - an optional string representing plot2d options in gnuplot format

EXAMPLES:

```
sage: maxima.plot2d_parametric(["sin(t)","cos(t)"], "t",[-3.1,3.1])    # not tested

sage: opts = '[gnuplot_preamble, "set nokey"], [gnuplot_term, ps], [gnuplot_out_file, "circl
sage: maxima.plot2d_parametric(["sin(t)","cos(t)"], "t", [-3.1,3.1], options=opts)    # not t
```

The eps file is saved to the current working directory.

Here is another fun plot:

```
sage: maxima.plot2d_parametric(["sin(5*t)","cos(11*t)"], "t", [0,2*pi()], nticks=400)    # r
```

plot3d(**args*)

Plot a 3d graph using Maxima / gnuplot.

maxima.plot3d(f, '[x, xmin, xmax]', '[y, ymin, ymax]', '[grid, nx, ny]', options)

INPUT:

- `f` - a string representing a function (such as f="sin(x)") [var, min, max]

EXAMPLES:

```
sage: maxima.plot3d('1 + x^3 - y^2', '[x,-2,2]', '[y,-2,2]', '[grid,12,12]')     # not tested
sage: maxima.plot3d('sin(x)*cos(y)', '[x,-2,2]', '[y,-2,2]', '[grid,30,30]')    # not tested
sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-plot.eps"]'
sage: maxima.plot3d('sin(x+y)', '[x,-5,5]', '[y,-1,1]', opts)     # not tested
```

The eps file is saved in the current working directory.

plot3d_parametric(*r, vars, urange, vrange, options=None*)

Plot a 3d parametric graph with r=(x,y,z), x = x(u,v), y = y(u,v), z = z(u,v), for u = umin...umax, v = vmin...vmax using gnuplot with options.

INPUT:

- `x, y, z` - a string representing a function (such as `x="u2+v2"`, ...) vars is a list or two strings representing variables (such as vars = ["u","v"])
- `urange` - [umin, umax]
- `vrange` - [vmin, vmax] are lists of numbers with umin umax, vmin vmax
- `options` - optional string representing plot2d options in gnuplot format

OUTPUT: displays a plot on screen or saves to a file

EXAMPLES:

```
sage: maxima.plot3d_parametric(["v*sin(u)","v*cos(u)","v"], ["u","v"],[-3.2,3.2],[0,3])
sage: opts = '[gnuplot_term, ps], [gnuplot_out_file, "sin-cos-plot.eps"]'
sage: maxima.plot3d_parametric(["v*sin(u)","v*cos(u)","v"], ["u","v"],[-3.2,3.2],[0,3],opts)
```

The eps file is saved in the current working directory.

Here is a torus:

```
sage: _ = maxima.eval("expr_1: cos(y)*(10.0+6*cos(x)); expr_2: sin(y)*(10.0+6*cos(x)); expr_
sage: maxima.plot3d_parametric(["expr_1","expr_2","expr_3"], ["x","y"],[0,6],[0,6])    # not
```

Here is a Mobius strip:

```
sage: x = "cos(u)*(3 + v*cos(u/2))"
sage: y = "sin(u)*(3 + v*cos(u/2))"
sage: z = "v*sin(u/2)"
sage: maxima.plot3d_parametric([x,y,z],["u","v"],[-3.1,3.2],[-1/10,1/10])    # not tested
```

plot_list(*ptsx, ptsy, options=None*)

Plots a curve determined by a sequence of points.

INPUT:

- `ptsx` - [x1,...,xn], where the xi and yi are real,
- `ptsy` - [y1,...,yn]
- `options` - a string representing maxima plot2d options.

The points are (x1,y1), (x2,y2), etc.

This function requires maxima 5.9.2 or newer.

Note: More that 150 points can sometimes lead to the program hanging. Why?

EXAMPLES:

```
sage: zeta_ptsx = [ (pari(1/2 + i*I/10).zeta().real()).precision(1) for i in range (70,150)]
sage: zeta_ptsy = [ (pari(1/2 + i*I/10).zeta().imag()).precision(1) for i in range (70,150)]
sage: maxima.plot_list(zeta_ptsx, zeta_ptsy)          # not tested
sage: opts='[gnuplot_preamble, "set nokey"], [gnuplot_term, ps], [gnuplot_out_file, "zeta.ep
sage: maxima.plot_list(zeta_ptsx, zeta_ptsy, opts)       # not tested
```

plot_multilist(*pts_list, options=None*)

Plots a list of list of points pts_list=[pts1,pts2,...,ptsn], where each ptsi is of the form [[x1,y1],...,[xn,yn]] x's must be integers and y's reals options is a string representing maxima plot2d options.

Requires maxima 5.9.2 at least.

Note: More that 150 points can sometimes lead to the program hanging.

EXAMPLES:

```
sage: xx = [ i/10.0 for i in range (-10,10)]
sage: yy = [ i/10.0 for i in range (-10,10)]
sage: x0 = [ 0 for i in range (-10,10)]
sage: y0 = [ 0 for i in range (-10,10)]
sage: zeta_ptsx1 = [ (pari(1/2+i*I/10).zeta().real()).precision(1) for i in range (10)]
sage: zeta_ptsy1 = [ (pari(1/2+i*I/10).zeta().imag()).precision(1) for i in range (10)]
sage: maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1],[xx,y0],[x0,yy]])       # not tested
sage: zeta_ptsx1 = [ (pari(1/2+i*I/10).zeta().real()).precision(1) for i in range (10,150)]
sage: zeta_ptsy1 = [ (pari(1/2+i*I/10).zeta().imag()).precision(1) for i in range (10,150)]
sage: maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1],[xx,y0],[x0,yy]])      # not tested
sage: opts='[gnuplot_preamble, "set nokey"]'
sage: maxima.plot_multilist([[zeta_ptsx1,zeta_ptsy1],[xx,y0],[x0,yy]],opts)    # not tested
```

set (*var, value*)

Set the variable var to the given value.

INPUT:

- `var` - string
- `value` - string

EXAMPLES:

```
sage: maxima.set('xxxxx', '2')
sage: maxima.get('xxxxx')
'2'
```

solve_linear (*eqns, vars*)

Wraps maxima's linsolve.

INPUT: eqns is a list of m strings, each rperesenting a linear question in m = n variables vars is a list of n strings, each representing a variable

EXAMPLES:

```
sage: eqns = ["x + z = y","2*a*x - y = 2*a^2","y - 2*z = 2"]
sage: vars = ["x","y","z"]
sage: maxima.solve_linear(eqns, vars)
[x=a+1,y=2*a,z=a-1]
```

trait_names (*verbose=True, use_disk_cache=True*)

Return all Maxima commands, which is useful for tab completion.

EXAMPLES:

```
sage: t = maxima.trait_names(verbose=False)
sage: 'gcd' in t
True
sage: len(t)    # random output
1743
```

unit_quadratic_integer (*n*)

Finds a unit of the ring of integers of the quadratic number field $\mathbf{Q}(\sqrt{n})$, $n > 1$, using the qunit maxima command.

EXAMPLES:

```
sage: u = maxima.unit_quadratic_integer(101); u
a + 10
sage: u.parent()
Number Field in a with defining polynomial x^2 - 101
```

```
sage: u = maxima.unit_quadratic_integer(13)
sage: u
5*a + 18
sage: u.parent()
Number Field in a with defining polynomial x^2 - 13
```

version()

Return the version of Maxima that Sage includes.

EXAMPLES:

```
sage: maxima.version()
'5.16.3'
```

class **MaximaElement**(*parent, value, is_name=False, name=None*)

bool()

EXAMPLES:

```
sage: maxima(0).bool()
False
sage: maxima(1).bool()
True
```

comma(*args*)

Form the expression that would be written 'self, args' in Maxima.

EXAMPLES:

```
sage: maxima('sqrt(2) + I').comma('numer')
I+1.414213562373095
sage: maxima('sqrt(2) + I*a').comma('a=5')
5*I+sqrt(2)
```

derivative(*var='x', n=1*)

Return the n-th derivative of self.

INPUT:

- `var` - variable (default: 'x')
- `n` - integer (default: 1)

OUTPUT: n-th derivative of self with respect to the variable var

EXAMPLES:

```
sage: f = maxima('x^2')
sage: f.diff()
2*x
sage: f.diff('x')
2*x
sage: f.diff('x', 2)
2
sage: maxima('sin(x^2)').diff('x',4)
16*x^4*sin(x^2)-12*sin(x^2)-48*x^2*cos(x^2)

sage: f = maxima('x^2 + 17*y^2')
sage: f.diff('x')
34*y*'diff(y,x,1)+2*x
sage: f.diff('y')
34*y
```

diff(*var='x', n=1*)

Return the n-th derivative of self.

INPUT:

- var - variable (default: 'x')
- n - integer (default: 1)

OUTPUT: n-th derivative of self with respect to the variable var

EXAMPLES:

```
sage: f = maxima('x^2')
sage: f.diff()
2*x
sage: f.diff('x')
2*x
sage: f.diff('x', 2)
2
sage: maxima('sin(x^2)').diff('x',4)
16*x^4*sin(x^2)-12*sin(x^2)-48*x^2*cos(x^2)

sage: f = maxima('x^2 + 17*y^2')
sage: f.diff('x')
34*y*'diff(y,x,1)+2*x
sage: f.diff('y')
34*y
```

display2d(*onscreen=True*)

EXAMPLES:

```
sage: F = maxima('x^5 - y^5').factor()
sage: F.display2d ()
                          4      3    2  2    3      4
            - (y - x) (y  + x y  + x  y  + x  y + x )
```

dot(*other*)

Implements the notation self . other.

EXAMPLES:

```
sage: A = maxima('matrix ([a1],[a2])')
sage: B = maxima('matrix ([b1, b2])')
sage: A.dot(B)
matrix([a1*b1,a1*b2],[a2*b1,a2*b2])
```

imag()

Return the imaginary part of this maxima element.

EXAMPLES:

```
sage: maxima('2 + (2/3)*%i').imag()
2/3
```

integral(*var='x', min=None, max=None*)

Return the integral of self with respect to the variable x.

INPUT:

- var - variable
- min - default: None
- max - default: None

Returns the definite integral if xmin is not None, otherwise returns an indefinite integral.

EXAMPLES:

```
sage: maxima('x^2+1').integral()
x^3/3+x
sage: maxima('x^2+ 1 + y^2').integral('y')
y^3/3+x^2*y+y
sage: maxima('x / (x^2+1)').integral()
log(x^2+1)/2
sage: maxima('1/(x^2+1)').integral()
atan(x)
sage: maxima('1/(x^2+1)').integral('x', 0, infinity)
%pi/2
sage: maxima('x/(x^2+1)').integral('x', -1, 1)
0

sage: f = maxima('exp(x^2)').integral('x',0,1); f
-sqrt(%pi)*%i*erf(%i)/2
sage: f.numer()          # I wonder how to get a real number (~1.463)??
-.8862269254527579*%i*erf(%i)
```

integrate(*var='x', min=None, max=None*)

Return the integral of self with respect to the variable x.

INPUT:

- `var` - variable
- `min` - default: None
- `max` - default: None

Returns the definite integral if xmin is not None, otherwise returns an indefinite integral.

EXAMPLES:

```
sage: maxima('x^2+1').integral()
x^3/3+x
sage: maxima('x^2+ 1 + y^2').integral('y')
y^3/3+x^2*y+y
sage: maxima('x / (x^2+1)').integral()
log(x^2+1)/2
sage: maxima('1/(x^2+1)').integral()
atan(x)
sage: maxima('1/(x^2+1)').integral('x', 0, infinity)
%pi/2
sage: maxima('x/(x^2+1)').integral('x', -1, 1)
0

sage: f = maxima('exp(x^2)').integral('x',0,1); f
-sqrt(%pi)*%i*erf(%i)/2
sage: f.numer()          # I wonder how to get a real number (~1.463)??
-.8862269254527579*%i*erf(%i)
```

nintegral(*var='x', a=0, b=1, desired_relative_error='1e-8', maximum_num_subintervals=200*)

Return a numerical approximation to the integral of self from a to b.

INPUT:

- `var` - variable to integrate with respect to
- `a` - lower endpoint of integration
- `b` - upper endpoint of integration
- `desired_relative_error` - (default: '1e-8') the desired relative error
- `maximum_num_subintervals` - (default: 200) maxima number of subintervals

OUTPUT:

- approximation to the integral

- estimated absolute error of the approximation
- the number of integrand evaluations
- an error code:
  - 0 - no problems were encountered
  - 1 - too many subintervals were done
  - 2 - excessive roundoff error
  - 3 - extremely bad integrand behavior
  - 4 - failed to converge
  - 5 - integral is probably divergent or slowly convergent
  - 6 - the input is invalid

EXAMPLES:

```
sage: maxima('exp(-sqrt(x))').nintegral('x',0,1)
(.5284822353142306, 4.163314137883845e-11, 231, 0)
```

Note that GP also does numerical integration, and can do so to very high precision very quickly:

```
sage: gp('intnum(x=0,1,exp(-sqrt(x)))')
0.52848223531423071361790491935415653021   # 64-bit
sage: _ = gp.set_precision(80)
sage: gp('intnum(x=0,1,exp(-sqrt(x)))')
0.52848223531423071361790491935415653021675547587292866196865279321015401702040079
```

numer()

Return numerical approximation to self as a Maxima object.

EXAMPLES:

```
sage: a = maxima('sqrt(2)').numer(); a
1.414213562373095
sage: type(a)
<class 'sage.interfaces.maxima.MaximaElement'>
```

partial_fraction_decomposition(*var='x'*)

Return the partial fraction decomposition of self with respect to the variable var.

EXAMPLES:

```
sage: f = maxima('1/((1+x)*(x-1))')
sage: f.partial_fraction_decomposition('x')
1/(2*(x-1))-1/(2*(x+1))
sage: print f.partial_fraction_decomposition('x')
                      1           1
                  --------- - ---------
                  2 (x - 1)   2 (x + 1)
```

real()

Return the real part of this maxima element.

EXAMPLES:

```
sage: maxima('2 + (2/3)*%i').real()
2
```

str()

Return string representation of this maxima object.

EXAMPLES:

```
sage: maxima('sqrt(2) + 1/3').str()
'sqrt(2)+1/3'
```

subst(*val*)

Substitute a value or several values into this Maxima object.

EXAMPLES:

```
sage: maxima('a^2 + 3*a + b').subst('b=2')
a^2+3*a+2
sage: maxima('a^2 + 3*a + b').subst('a=17')
b+340
sage: maxima('a^2 + 3*a + b').subst('a=17, b=2')
342
```

trait_names(*verbose=False*)

Return all Maxima commands, which is useful for tab completion.

EXAMPLES:

```
sage: m = maxima(2)
sage: 'gcd' in m.trait_names()
True
```

class **MaximaExpectFunction**(*parent, name*)

class **MaximaFunction**(*parent, name, defn, args, latex*)

arguments(*split=True*)

Returns the arguments of this Maxima function.

EXAMPLES:

```
sage: f = maxima.function('x,y','sin(x+y)')
sage: f.arguments()
['x', 'y']
sage: f.arguments(split=False)
'x,y'
sage: f = maxima.function('', 'sin(x)')
sage: f.arguments()
[]
```

definition()

Returns the definition of this Maxima function as a string.

EXAMPLES:

```
sage: f = maxima.function('x,y','sin(x+y)')
sage: f.definition()
'sin(x+y)'
```

integral(*var*)

Returns the integral of self with respect to the variable var.

Note that integrate is an alias of integral.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = maxima.function('x','sin(x)')
sage: f.integral(x)
-cos(x)
sage: f.integral(y)
sin(x)*y
```

integrate(*var*)

Returns the integral of self with respect to the variable var.

Note that integrate is an alias of integral.

EXAMPLES:

```
sage: x,y = var('x,y')
sage: f = maxima.function('x','sin(x)')
sage: f.integral(x)
-cos(x)
sage: f.integral(y)
sin(x)*y
```

class MaximaFunctionElement(*obj, name*)

is_MaximaElement(*x*)

Returns True if x is of type MaximaElement.

EXAMPLES:

```
sage: from sage.interfaces.maxima import is_MaximaElement
sage: m = maxima(1)
sage: is_MaximaElement(m)
True
sage: is_MaximaElement(1)
False
```

maxima_console()

Spawn a new Maxima command-line session.

EXAMPLES:

```
sage: from sage.interfaces.maxima import maxima_console
sage: maxima_console()                    # not tested
Maxima 5.16.3 http://maxima.sourceforge.net
...
```

maxima_version()

EXAMPLES:

```
sage: from sage.interfaces.maxima import maxima_version
sage: maxima_version()
'5.16.3'
```

reduce_load_Maxima()

EXAMPLES:

```
sage: from sage.interfaces.maxima import reduce_load_Maxima
sage: reduce_load_Maxima()
Maxima
```

reduce_load_Maxima_function(*parent, defn, args, latex*)

13.11 Interface to Mathematica

The Mathematica interface will only work if Mathematica is installed on your computer with a command line interface that runs when you give the `math` command. The interface offers three pieces of functionality:

1. `mathematica_console()` - A function that dumps you into an interactive command-line Mathematica session. This is an enhanced version of the usual Mathematica command-line, in that it provides readline editing and history (the usual one doesn't!)

2. `mathematica(expr)` - Creation of a Sage object that wraps a Mathematica object. This provides a Pythonic interface to Mathematica. For example, if `f=mathematica('x2-1')`, then `f.Factor()` returns the factorization of $x^2 - 1$ computed using Mathematica.

3. `mathematica.eval(expr)` - Evaluation of arbitrary Mathematica expressions, with the result returned as a string.

13.11.1 Tutorial

We follow some of the tutorial from http://library.wolfram.com/conferences/devconf99/withoff/Basic1.html/.

For any of this to work you must buy and install the Mathematica program, and it must be available as the command `math` in your PATH.

Syntax

Now make 1 and add it to itself. The result is a Mathematica object.

```
sage: m = mathematica
sage: a = m(1) + m(1); a                # optional - mathematica
2
sage: a.parent()                        # optional - mathematica
Mathematica
sage: m('1+1')                          # optional - mathematica
2
sage: m(3)**m(50)                       # optional - mathematica
717897987691852588770249
```

The following is equivalent to `Plus[2, 3]` in Mathematica:

```
sage: m = mathematica
sage: m(2).Plus(m(3))                   # optional - mathematica
5
```

We can also compute $7(2 + 3)$.

```
sage: m(7).Times(m(2).Plus(m(3)))       # optional - mathematica
35
sage: m('7(2+3)')                       # optional - mathematica
35
```

Some typical input

We solve an equation and a system of two equations:

```
sage: eqn = mathematica('3x + 5 == 14') # optional - mathematica
sage: eqn                               # optional - mathematica
5 + 3*x == 14
sage: eqn.Solve('x')                    # optional - mathematica
{{x -> 3}}
```

```
sage: sys = mathematica('{x^2 - 3y == 3, 2x - y == 1}')  # optional - mathematica
sage: print sys                          # optional - mathematica
          2
        {x  - 3 y == 3, 2 x - y == 1}
sage: sys.Solve('{x, y}')                # optional - mathematica
{{y -> -1, x -> 0}, {y -> 11, x -> 6}}
```

Assignments and definitions

If you assign the mathematica 5 to a variable c in Sage, this does not affect the c in Mathematica.

```
sage: c = m(5)                           # optional - mathematica
sage: print m('b + c x')                 # optional - mathematica
            b + c x
sage: print m('b') + c*m('x')            # optional - mathematica
        b + 5 x
```

The Sage interfaces changes Sage lists into Mathematica lists:

```
sage: m = mathematica
sage: eq1 = m('x^2 - 3y == 3')           # optional - mathematica
sage: eq2 = m('2x - y == 1')             # optional - mathematica
sage: v = m([eq1, eq2]); v               # optional - mathematica
{x^2 - 3*y == 3, 2*x - y == 1}
sage: v.Solve(['x', 'y'])                # optional - mathematica
{{y -> -1, x -> 0}, {y -> 11, x -> 6}}
```

Function definitions

Define mathematica functions by simply sending the definition to the interpreter.

```
sage: m = mathematica
sage: _ = mathematica('f[p_] = p^2');    # optional - mathematica
sage: m('f[9]')                          # optional - mathematica
81
```

Numerical Calculations

We find the x such that $e^x - 3x = 0$.

```
sage: e = mathematica('Exp[x] - 3x == 0') # optional - mathematica
sage: e.FindRoot(['x', 2])               # optional - mathematica
{x -> 1.512134551657842}
```

Note that this agrees with what the PARI interpreter gp produces:

```
sage: gp('solve(x=1,2,exp(x)-3*x)')
1.5121345516578424738967396780720387046    # 64-bit
1.5121345516578424738967396780720387046    # 64-bit
```

Next we find the minimimum of a polynomial using the two different ways of accessing Mathematica:

```
sage: mathematica('FindMinimum[x^3 - 6x^2 + 11x - 5, {x,3}]')  # optional - mathematica
{0.6150998205402516, {x -> 2.5773502699629733}}
sage: f = mathematica('x^3 - 6x^2 + 11x - 5')  # optional - mathematica
sage: f.FindMinimum(['x', 3])                  # optional - mathematica
{0.6150998205402516, {x -> 2.5773502699629733}}
```

Polynomial and Integer Factorization

We factor a polynomial of degree 200 over the integers.

```
sage: R.<x> = PolynomialRing(ZZ)
sage: f = (x**100+17*x+5)*(x**100-5*x+20)
sage: f
x^200 + 12*x^101 + 25*x^100 - 85*x^2 + 315*x + 100
sage: g = mathematica(str(f))            # optional - mathematica
sage: print g                            # optional - mathematica
                          2       100       101    200
        100 + 315 x - 85 x  + 25 x    + 12 x    + x
sage: g                                  # optional - mathematica
100 + 315*x - 85*x^2 + 25*x^100 + 12*x^101 + x^200
sage: print g.Factor()                   # optional - mathematica
                     100               100
        (20 - 5 x + x    ) (5 + 17 x + x    )
```

We can also factor a multivariate polynomial:

```
sage: f = mathematica('x^6 + (-y - 2)*x^5 + (y^3 + 2*y)*x^4 - y^4*x^3')  # optional - mathematica
sage: print f.Factor()                   # optional - mathematica
         3                   2    3
        x  (x - y) (-2 x + x  + y )
```

We factor an integer:

```
sage: n = mathematica(2434500)           # optional - mathematica
sage: n.FactorInteger()                  # optional - mathematica
{{2, 2}, {3, 2}, {5, 3}, {541, 1}}       # optional - mathematica
sage: n = mathematica(2434500)           # optional - mathematica
sage: F = n.FactorInteger(); F           # optional - mathematica
{{2, 2}, {3, 2}, {5, 3}, {541, 1}}
sage: F[1]                               # optional - mathematica
{2, 2}
sage: F[4]                               # optional - mathematica
{541, 1}
```

We can also load the ECM package and factoring using it:

```
sage: _ = mathematica.eval("<<NumberTheory`FactorIntegerECM`");  # optional - mathematica
sage: mathematica.FactorIntegerECM('932901*939321')              # optional - mathematica
8396109
```

13.11.2 Long Input

The Mathematica interface reads in even very long input (using files) in a robust manner.

```
sage: t = '"%s"'%10^10000   # ten thousand character string.
sage: a = mathematica(t)        # optional - mathematica
sage: a = mathematica.eval(t)   # optional - mathematica
```

13.11.3 Loading and saving

Mathematica has an excellent `InputForm` function, which makes saving and loading Mathematica objects possible. The first examples test saving and loading to strings.

```
sage: x = mathematica(pi/2)     # optional - mathematica
sage: print x                   # optional - mathematica
        Pi
        --
        2
sage: loads(dumps(x)) == x      # optional - mathematica
True
sage: n = x.N(50)               # optional - mathematica
sage: print n                   # optional - mathematica
             1.5707963267948966192313216916397514420985846996876
sage: loads(dumps(n)) == n      # optional - mathematica
True
```

OTHER Examples:

```
sage: def math_bessel_K(nu,x):
...       return mathematica(nu).BesselK(x).N(20).sage()
...
sage: math_bessel_K(2,I)                      # optional - mathematica
0.180489972066962*I - 2.592886175491197
```

AUTHORS:

- William Stein (2005): first version
- Doug Cutrell (2006-03-01): Instructions for use under Cygwin/Windows.

class **Mathematica**(*maxread=100, script_subdirectory='', logfile=None, server=None, server_tmpdir=None*)
Interface to the Mathematica interpreter.

chdir(*dir*)
Change Mathematica's current working directory.
EXAMPLES:

```
sage: mathematica.chdir('/')           # optional
sage: mathematica('Directory[]')       # optional
"/"
```

console(*readline=True*)

eval(*code, strip=True, **kwds*)

get(*var, ascii_art=False*)
Get the value of the variable var.
AUTHORS:
•William Stein
•Kiran Kedlaya (2006-02-04): suggested using InputForm

help(*cmd*)

set(*var, value*)
Set the variable var to the given value.

trait_names()

class **MathematicaElement**(*parent, value, is_name=False, name=None*)

show(*filename=None, ImageSize=600*)
Show a mathematica expression or plot in the Sage notebook.
EXAMPLES:

```
sage: P = mathematica('Plot[Sin[x],{x,-2Pi,4Pi}]')    # optional - mathematica
sage: show(P)                                         # optional - mathematica
sage: P.show(ImageSize=800)                           # optional - mathematica
sage: Q = mathematica('Sin[x Cos[y]]/Sqrt[1-x^2]')    # optional - mathematica
sage: show(Q)                                         # optional - mathematica
<html><div class="math">\frac{\sin (x \cos (y))}{\sqrt{1-x^2}}</div></html>
```

str()

class **MathematicaFunction**(*parent, name*)

class **MathematicaFunctionElement**(*obj, name*)

clean_output(*s*)

mathematica_console(*readline=True*)

reduce_load(*X*)

13.12 Interface to mwrank

Mwrank(*options='', server=None, server_tmpdir=None*)
Create and return an mwrank interpreter, with given options.
INPUT:

- `options` - string; passed when starting mwrank. The format is q pprecision vverbosity bhlim_q xnaux chlim_c l t o s d]

class **Mwrank_class**(*options='', server=None, server_tmpdir=None*)
Interface to the Mwrank interpreter.

console()

eval(**args, **kwds*)
Send a line of input to mwrank, then when it finishes return everything that mwrank output.
NOTE: If a RuntimeError exception is raised, then the mwrank interface is restarted and the command is retried once.

EXAMPLES: sage: mwrank.eval('12 3 4 5 6') 'Curve [12,3,4,5,6] :...'

quit(*verbose=False*)
Quit the mwrank process using kill -9 (so exit doesn't dump core, etc.).

INPUT: verbose – ignored

EXAMPLES: sage: m = Mwrank() sage: e = m('1 2 3 4 5') sage: m.quit()

mwrank_console()

13.13 Interface to Octave

Octave is an open source MATLAB-like program with numerical routines for integrating, solving systems of equations, special functions, and solving (numerically) differential equations. Please see http://octave.sourceforge.net for more details.

The commands in this section only work if you have the optional "octave" interpreter installed and available in your PATH. It's not necessary to install any special Sage packages.

EXAMPLES:

```
sage: octave.eval('2+2')     # optional -- requires Octave
'ans = 4'

sage: a = octave(10)         # optional -- requires Octave
sage: a**10                  # optional -- requires Octave
1e+10
```

LOG: - creation (William Stein) - ? (David Joyner, 2005-12-18) - Examples (David Joyner, 2005-01-03)

13.13.1 Computation of Special Functions

Octave implements computation of the following special functions (see the maxima and gp interfaces for even more special functions):

```
airy
    Airy functions of the first and second kind, and their derivatives.
    airy(0,x) = Ai(x), airy(1,x) = Ai'(x), airy(2,x) = Bi(x), airy(3,x) = Bi'(x)
besselj
    Bessel functions of the first kind.
bessely
    Bessel functions of the second kind.
besseli
    Modified Bessel functions of the first kind.
besselk
    Modified Bessel functions of the second kind.
besselh
    Compute Hankel functions of the first (k = 1) or second (k = 2) kind.
beta
    The Beta function,
          beta (a, b) = gamma (a) * gamma (b) / gamma (a + b).
betainc
    The incomplete Beta function,
erf
    The error function,
erfinv
    The inverse of the error function.
gamma
    The Gamma function,
gammainc
    The incomplete gamma function,
```

For example,

```
sage: octave("airy(3,2)")          # optional -- requires Octave
4.10068
```

```
sage: octave("beta(2,2)")          # optional -- requires Octave
0.166667
sage: octave("betainc(0.2,2,2)")   # optional -- requires Octave
0.104
sage: octave("besselh(0,2)")       # optional -- requires Octave
(0.223891,0.510376)
sage: octave("besselh(0,1)")       # optional -- requires Octave
(0.765198,0.088257)
sage: octave("besseli(1,2)")       # optional -- requires Octave
1.59064
sage: octave("besselj(1,2)")       # optional -- requires Octave
0.576725
sage: octave("besselk(1,2)")       # optional -- requires Octave
0.139866
sage: octave("erf(0)")             # optional -- requires Octave
0
sage: octave("erf(1)")             # optional -- requires Octave
0.842701
sage: octave("erfinv(0.842)")      # optional -- requires Octave
0.998315
sage: octave("gamma(1.5)")         # optional -- requires Octave
0.886227
sage: octave("gammainc(1.5,1)")    # optional -- requires Octave
0.77687
```

The Octave interface reads in even very long input (using files) in a robust manner:

```
sage: t = '"%s"'%10^10000   # ten thousand character string.
sage: a = octave.eval(t + ';')     # optional -- requires Octave, < 1/100th of a second
sage: a = octave(t)                # optional -- requires Octave
```

Note that actually reading a back out takes forever. This *must* be fixed ASAP - see http://trac.sagemath.org/sage_trac/ticket/940/.

13.13.2 Tutorial

EXAMPLES:

```
sage: octave('4+10')               # optional -- requires Octave
14
sage: octave('date')               # optional -- requires Octave; random output
18-Oct-2007
sage: octave('5*10 + 6')           # optional -- requires Octave
56
sage: octave('(6+6)/3')            # optional -- requires Octave
4
sage: octave('9')^2                # optional -- requires Octave
81
sage: a = octave(10); b = octave(20); c = octave(30)    # optional -- requires Octave
sage: avg = (a+b+c)/3              # optional -- requires Octave
sage: avg                          # optional -- requires Octave
20
sage: parent(avg)                  # optional -- requires Octave
Octave
```

```
sage: my_scalar = octave('3.1415')        # optional -- requires Octave
sage: my_scalar                           # optional -- requires Octave
3.1415
sage: my_vector1 = octave('[1,5,7]')      # optional -- requires Octave
sage: my_vector1                          # optional -- requires Octave
1     5     7
sage: my_vector2 = octave('[1;5;7]')      # optional -- requires Octave
sage: my_vector2                          # optional -- requires Octave
1
5
7
sage: my_vector1 * my_vector2             # optional -- requires Octave
75
```

class **Octave**(*maxread=100, script_subdirectory='', logfile=None, server=None, server_tmpdir=None*)

Interface to the Octave interpreter.

EXAMPLES:

```
sage: octave.eval("a = [ 1, 1, 2; 3, 5, 8; 13, 21, 33 ]")     # optional -- requires Octave
'a =\n\n 1 1 2\n 3 5 8\n 13 21 33\n\n'
sage: octave.eval("b = [ 1; 3; 13]")                          # optional -- requires Octave
'b =\n\n 1\n 3\n 13\n\n'
sage: octave.eval("c=a \\ b") # solves linear equation: a*c = b  # optional -- requires Octave;
'c =\n\n 1\n 7.21645e-16\n -7.21645e-16\n\n'
sage: octave.eval("c")                                  # optional -- requires Octave; random out
'c =\n\n 1\n 7.21645e-16\n -7.21645e-16\n\n'
```

clear(*var*)

Clear the variable named var.

EXAMPLES:

```
sage: octave.set('x', '2') #optional -- requires Octave
sage: octave.clear('x') #optional -- requires Octave
sage: octave.get('x') #optional -- requires Octave
"error: 'x' undefined near line ... column 1"
```

console()

Spawn a new Octave command-line session.

This requires that the optional octave program be installed and in your PATH, but no optional Sage packages need be installed.

EXAMPLES:

```
sage: octave_console()          # not tested
GNU Octave, version 2.1.73 (i386-apple-darwin8.5.3).
Copyright (C) 2006 John W. Eaton.
...
octave:1> 2+3
ans = 5
octave:2> [ctl-d]
```

Pressing ctrl-d exits the octave console and returns you to Sage. octave, like Sage, remembers its history from one session to another.

de_system_plot(*f, ics, trange*)

Plots (using octave's interface to gnuplot) the solution to a 2×2 system of differential equations.

INPUT:

- `f` - a pair of strings representing the differential equations; The independent variable must be called x and the dependent variable must be called y.

- `ics` - a pair [x0,y0] such that x(t0) = x0, y(t0) = y0
- `trange` - a pair [t0,t1]

OUTPUT: a gnuplot window appears

EXAMPLES:

```
sage: octave.de_system_plot(['x+y','x-y'], [1,-1], [0,2])  # not tested -- does this actuall
```

This should yield the two plots $(t, x(t)), (t, y(t))$ on the same graph (the t-axis is the horizonal axis) of the system of ODEs

$$x' = x + y, x(0) = 1; \qquad y' = x - y, y(0) = -1, \quad \text{for} \quad 0 < t < 2.$$

get (*var*)

Get the value of the variable var.

EXAMPLES:

```
sage: octave.set('x', '2') #optional -- requires Octave
sage: octave.get('x') #optional -- requires Octave
' 2'
```

quit (*verbose=False*)

EXAMPLES:

```
sage: o = Octave()
sage: o._start()    #optional -- requires Octave
sage: o.quit(True)  #optional -- requires Octave
Exiting spawned Octave process.
```

sage2octave_matrix_string (*A*)

Return an octave matrix from a Sage matrix.

INPUT: A Sage matrix with entries in the rationals or reals.

OUTPUT: A string that evaluates to an Octave matrix.

EXAMPLES:

```
sage: M33 = MatrixSpace(QQ,3,3)
sage: A = M33([1,2,3,4,5,6,7,8,0])
sage: octave.sage2octave_matrix_string(A)    # optional -- requires Octave
'[1, 2, 3; 4, 5, 6; 7, 8, 0]'
```

AUTHORS:

- David Joyner and William Stein

set (*var, value*)

Set the variable var to the given value.

EXAMPLES:

```
sage: octave.set('x', '2') #optional -- requires Octave
sage: octave.get('x') #optional -- requires Octave
' 2'
```

solve_linear_system (*A, b*)

Use octave to compute a solution x to A*x = b, as a list.

INPUT:

- `A` - mxn matrix A with entries in QQ or RR
- `b` - m-vector b entries in QQ or RR (resp)

OUTPUT: An list x (if it exists) which solves M*x = b

EXAMPLES:

```
sage: M33 = MatrixSpace(QQ,3,3)
sage: A   = M33([1,2,3,4,5,6,7,8,0])
sage: V3  = VectorSpace(QQ,3)
sage: b   = V3([1,2,3])
sage: octave.solve_linear_system(A,b)     # optional -- requires Octave (and output is slight
[-0.33333299999999999, 0.66666700000000001, -3.5236600000000002e-18]
```

AUTHORS:

- David Joyner and William Stein

version()

Return the version of Octave.

OUTPUT: string

EXAMPLES:

```
sage: octave.version()    # optional -- requires Octave; random output depending on version
'2.1.73'
```

class OctaveElement(*parent, value, is_name=False, name=None*)

octave_console()

Spawn a new Octave command-line session.

This requires that the optional octave program be installed and in your PATH, but no optional Sage packages need be installed.

EXAMPLES:

```
sage: octave_console()          # not tested
GNU Octave, version 2.1.73 (i386-apple-darwin8.5.3).
Copyright (C) 2006 John W. Eaton.
...
octave:1> 2+3
ans = 5
octave:2> [ctl-d]
```

Pressing ctrl-d exits the octave console and returns you to Sage. octave, like Sage, remembers its history from one session to another.

octave_version()

Return the version of Octave installed.

EXAMPLES:

```
sage: octave_version()     # optional -- requires Octave; and output is random
'2.9.12'
```

reduce_load_Octave()

EXAMPLES:

```
sage: from sage.interfaces.octave import reduce_load_Octave
sage: reduce_load_Octave()
Octave
```

13.14 Interface to Sage

This is an expect interface to *another* copy of the Sage interpreter.

class Sage (*logfile=None, preparse=True, python=False, init_code=None, server=None, server_tmpdir=None, remote_cleaner=True, **kwds*)

Expect interface to the Sage interpreter itself.

INPUT:

- `server` - (optional); if specified runs Sage on a remote machine with address. You must have ssh keys setup so you can login to the remote machine by typing "ssh remote_machine" and no password, call _install_hints_ssh() for hints on how to do that.

The version of Sage should be the same as on the local machine, since pickling is used to move data between the two Sage process.

EXAMPLES: We create an interface to a copy of Sage. This copy of Sage runs as an external process with its own memory space, etc.

```
sage: s = Sage()
```

Create the element 2 in our new copy of Sage, and cubeit.

```
sage: a = s(2)
sage: a^3
8
```

Create a vector space of dimension 4, and compute its generators:

```
sage: V = s('QQ^4')
sage: V.gens()
((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))
```

Note that V is a not a vector space, it's a wrapper around an object (which happens to be a vector space), in another running instance of Sage.

```
sage: type(V)
<class 'sage.interfaces.sage0.SageElement'>
sage: V.parent()
Sage
sage: g = V.0;  g
(1, 0, 0, 0)
sage: g.parent()
Sage
```

We can still get the actual parent by using the name attribute of g, which is the variable name of the object in the child process.

```
sage: s('%s.parent()'%g.name())
Vector space of dimension 4 over Rational Field
```

Note that the memory space is completely different.

```
sage: x = 10
sage: s('x = 5')
5
sage: x
10
sage: s('x')
5
```

We can have the child interpreter itself make another child Sage process, so now three copies of Sage are running:

```
sage: s3 = s('Sage()')
sage: a = s3(10)
sage: a
10
```

This $a = 10$ is in a subprocess of a subprocesses of your original Sage.

```
sage: _ = s.eval('%s.eval("x=8")'%s3.name())
sage: s3('"x"')
8
sage: s('x')
5
sage: x
10
```

The double quotes are needed because the call to s3 first evaluates its arguments using the s interpeter, so the call to s3 is passed `s('"x"')`, which is the string `"x"` in the s interpreter.

clear(*var*)

Clear the variable named var.

EXAMPLES:

```
sage: sage0.set('x', '2')
sage: sage0.get('x')
'2'
sage: sage0.clear('x')
sage: sage0.get('x')
"...NameError: name 'x' is not defined"
```

console()

Spawn a new Sage command-line session.

EXAMPLES:

```
sage: sage0.console() #not tested
--------------------------------------------------------------------------
| Sage Version ..., Release Date: ...                                    |
| Type notebook() for the GUI, and license() for information.            |
--------------------------------------------------------------------------
...
```

cputime(*t=None*)

Return cputime since this Sage subprocess was started.

EXAMPLES:

```
sage: sage0.cputime()      # random output
1.3530439999999999
sage: sage0('factor(2^157-1)')
852133201 * 60726444167 * 1654058017289 * 2134387368610417
sage: sage0.cputime()      # random output
1.6462939999999999
```

eval(*line, strip=True, **kwds*)

Send the code x to a second instance of the Sage interpreter and return the output as a string.

This allows you to run two completely independent copies of Sage at the same time in a unified way.

INPUT:

- `line` - input line of code
- `strip` - ignored

EXAMPLES:

```
sage: sage0.eval('2+2')
'4'
```

get(*var*)
Get the value of the variable var.
EXAMPLES:

```
sage: sage0.set('x', '2')
sage: sage0.get('x')
'2'
```

new(*x*)
EXAMPLES:

```
sage: sage0.new(2)
2
sage: _.parent()
Sage
```

preparse(*x*)
Returns the preparsed version of the string s.
EXAMPLES:

```
sage: sage0.preparse('2+2')
'Integer(2)+Integer(2)'
```

quit(*verbose=False*)
EXAMPLES:

```
sage: s = Sage()
sage: s.eval('2+2')
'4'
sage: s.quit()
```

set(*var, value*)
Set the variable var to the given value.
EXAMPLES:

```
sage: sage0.set('x', '2')
sage: sage0.get('x')
'2'
```

trait_names()
EXAMPLES:

```
sage: t = sage0.trait_names()
sage: len(t) > 100
True
sage: 'gcd' in t
True
```

version()
EXAMPLES:

```
sage: sage0.version()
'Sage Version ..., Release Date: ...'
sage: sage0.version() == version()
True
```

class SageElement(*parent, value, is_name=False, name=None*)

class SageFunction(*obj*, *name*)

reduce_load_Sage()
EXAMPLES:

```
sage: from sage.interfaces.sage0 import reduce_load_Sage
sage: reduce_load_Sage()
Sage
```

reduce_load_element(*s*)
EXAMPLES:

```
sage: from sage.interfaces.sage0 import reduce_load_element
sage: s = dumps(1/2)
sage: half = reduce_load_element(s); half
1/2
sage: half.parent()
Sage
```

sage0_console()
Spawn a new Sage command-line session.

EXAMPLES:

```
sage: sage0_console() #not tested
--------------------------------------------------------------------------
| Sage Version ..., Release Date: ...                                    |
| Type notebook() for the GUI, and license() for information.            |
--------------------------------------------------------------------------
...
```

sage0_version()
EXAMPLES:

```
sage: from sage.interfaces.sage0 import sage0_version
sage: sage0_version() == version()
True
```

13.15 Interface to Singular

AUTHORS:

- David Joyner and William Stein (2005): first version
- Martin Albrecht (2006-03-05): code so singular.[tab] and x = singular(...), x.[tab] includes all singular commands.
- Martin Albrecht (2006-03-06): This patch adds the equality symbol to singular. Also fix problem in which" " as prompt means comparison will break all further communication with Singular.
- Martin Albrecht (2006-03-13): added current_ring() and current_ring_name()
- William Stein (2006-04-10): Fixed problems with ideal constructor
- Martin Albrecht (2006-05-18): added sage_poly.

13.15.1 Introduction

This interface is extremely flexible, since it's exactly like typing into the Singular interpreter, and anything that works there should work here.

The Singular interface will only work if Singular is installed on your computer; this should be the case, since Singular is included with Sage. The interface offers three pieces of functionality:

1. `singular_console()` - A function that dumps you into an interactive command-line Singular session.
2. `singular(expr, type='def')` - Creation of a Singular object. This provides a Pythonic interface to Singular. For example, if `f=singular(10)`, then `f.factorize()` returns the factorization of 10 computed using Singular.
3. `singular.eval(expr)` - Evaluation of arbitrary Singular expressions, with the result returned as a string.

13.15.2 Tutorial

EXAMPLES: First we illustrate multivariate polynomial factorization:

```
sage: R1 = singular.ring(0, '(x,y)', 'dp')
sage: R1
//   characteristic : 0
//   number of vars : 2
//        block   1 : ordering dp
//                  : names    x y
//        block   2 : ordering C
sage: f = singular('9x16 - 18x13y2 - 9x12y3 + 9x10y4 - 18x11y2 + 36x8y4 + 18x7y5 - 18x5y6 + 9x6y4 - 1
sage: f
9*x^16-18*x^13*y^2-9*x^12*y^3+9*x^10*y^4-18*x^11*y^2+36*x^8*y^4+18*x^7*y^5-18*x^5*y^6+9*x^6*y^4-18*x
sage: f.parent()
Singular


sage: F = f.factorize(); F
[1]:
   _[1]=9
   _[2]=x^6-2*x^3*y^2-x^2*y^3+y^4
   _[3]=-x^5+y^2
[2]:
   1,1,2


sage: F[1]
9,
x^6-2*x^3*y^2-x^2*y^3+y^4,
-x^5+y^2
sage: F[1][2]
x^6-2*x^3*y^2-x^2*y^3+y^4
```

We can convert f and each exponent back to Sage objects as well.

```
sage: R.<x, y> = PolynomialRing(QQ,2)
sage: g = eval(f.sage_polystring()); g
9*x^16 - 18*x^13*y^2 - 9*x^12*y^3 + 9*x^10*y^4 - 18*x^11*y^2 + 36*x^8*y^4 + 18*x^7*y^5 - 18*x^5*y^6 -
sage: eval(F[1][2].sage_polystring())
x^6 - 2*x^3*y^2 - x^2*y^3 + y^4
```

This example illustrates polynomial GCD's:

```
sage: R2 = singular.ring(0, '(x,y,z)', 'lp')
sage: a = singular.new('3x2*(x+y)')
sage: b = singular.new('9x*(y2-x2)')
sage: g = a.gcd(b)
sage: g
x^2+x*y
```

This example illustrates computation of a Groebner basis:

```
sage: R3 = singular.ring(0, '(a,b,c,d)', 'lp')
sage: I = singular.ideal(['a + b + c + d', 'a*b + a*d + b*c + c*d', 'a*b*c + a*b*d + a*c*d + b*c*d',
sage: I2 = I.groebner()
sage: I2
c^2*d^6-c^2*d^2-d^4+1,
c^3*d^2+c^2*d^3-c-d,
b*d^4-b+d^5-d,
b*c-b*d^5+c^2*d^4+c*d-d^6-d^2,
b^2+2*b*d+d^2,
a+b+c+d
```

The following example is the same as the one in the Singular - Gap interface documentation:

```
sage: R  = singular.ring(0, '(x0,x1,x2)', 'lp')
sage: I1 = singular.ideal(['x0*x1*x2 -x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2', 'x0*x1-x0*x2-x1*x
sage: I2 = I1.groebner()
sage: I2
x1^2*x2^2,
x0*x2^3-x1^2*x2^2+x1*x2^3,
x0*x1-x0*x2-x1*x2,
x0^2*x2-x0*x1*x2
```

This example illustrates moving a polynomial from one ring to another. It also illustrates calling a method of an object with an argument.

```
sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: f = singular('x3+y3+(x-y)*x2y2+z2')
sage: f
x^3*y^2-x^2*y^3+x^3+y^3+z^2
sage: R1 = singular.ring(0, '(x,y,z)', 'ds')
sage: f = R.fetch(f)
sage: f
z^2+x^3+y^3+x^3*y^2-x^2*y^3
```

We can calculate the Milnor number of f:

```
sage: _=singular.LIB('sing.lib')      # assign to _ to suppress printing
sage: f.milnor()
4
```

The Jacobian applied twice yields the Hessian matrix of f, with which we can compute.

```
sage: H = f.jacob().jacob()
sage: H
6*x+6*x*y^2-2*y^3,6*x^2*y-6*x*y^2,  0,
6*x^2*y-6*x*y^2,  6*y+2*x^3-6*x^2*y,0,
```

```
0,                0,                2
sage: H.det()
72*x*y+24*x^4-72*x^3*y+72*x*y^3-24*y^4-48*x^4*y^2+64*x^3*y^3-48*x^2*y^4
```

The 1x1 and 2x2 minors:

```
sage: H.minor(1)
2,
6*y+2*x^3-6*x^2*y,
6*x^2*y-6*x*y^2,
6*x^2*y-6*x*y^2,
6*x+6*x*y^2-2*y^3
sage: H.minor(2)
12*y+4*x^3-12*x^2*y,
12*x^2*y-12*x*y^2,
12*x^2*y-12*x*y^2,
12*x+12*x*y^2-4*y^3,
-36*x*y-12*x^4+36*x^3*y-36*x*y^3+12*y^4+24*x^4*y^2-32*x^3*y^3+24*x^2*y^4
```

```
sage: _=singular.eval('option(redSB)')
sage: H.minor(1).groebner()
1
```

13.15.3 Computing the Genus

We compute the projective genus of ideals that define curves over $\mathbf{Q}$. It is *very important* to load the `normal.lib` library before calling the `genus` command, or you'll get an error message.

EXAMPLE:

```
sage: singular.lib('normal.lib')
sage: R = singular.ring(0,'(x,y)','dp')
sage: i2 = singular.ideal('y9 - x2*(x-1)^9 + x')
sage: i2.genus()
40
```

Note that the genus can be much smaller than the degree:

```
sage: i = singular.ideal('y9 - x2*(x-1)^9')
sage: i.genus()
0
```

13.15.4 An Important Concept

AUTHORS:

- Neal Harris

The following illustrates an important concept: how Sage interacts with the data being used and returned by Singular. Let's compute a Groebner basis for some ideal, using Singular through Sage.

```
sage: singular.lib('poly.lib')
sage: singular.ring(32003, '(a,b,c,d,e,f)', 'lp')
        //   characteristic : 32003
        //   number of vars : 6
        //        block   1 : ordering lp
        //                  : names    a b c d e f
        //        block   2 : ordering C
sage: I = singular.ideal('cyclic(6)')
sage: g = singular('groebner(I)')
...
TypeError: Singular error:
...
```

We restart everything and try again, but correctly.

```
sage: singular.quit()
sage: singular.lib('poly.lib'); R = singular.ring(32003, '(a,b,c,d,e,f)', 'lp')
sage: I = singular.ideal('cyclic(6)')
sage: I.groebner()
f^48-2554*f^42-15674*f^36+12326*f^30-12326*f^18+15674*f^12+2554*f^6-1,
...
```

It's important to understand why the first attempt at computing a basis failed. The line where we gave singular the input 'groebner(I)' was useless because Singular has no idea what 'I' is! Although 'I' is an object that we computed with calls to Singular functions, it actually lives in Sage. As a consequence, the name 'I' means nothing to Singular. When we called `I.groebner()`, Sage was able to call the groebner function on'I' in Singular, since 'I' actually means something to Sage.

13.15.5 Long Input

The Singular interface reads in even very long input (using files) in a robust manner, as long as you are creating a new object.

```
sage: t = '"%s"'%10^15000   # 15 thousand character string (note that normal Singular input must be a
sage: a = singular.eval(t)
sage: a = singular(t)
```

TESTS: We test an automatic coercion:

```
sage: a = 3*singular('2'); a
6
sage: type(a)
<class 'sage.interfaces.singular.SingularElement'>
sage: a = singular('2')*3; a
6
sage: type(a)
<class 'sage.interfaces.singular.SingularElement'>
```

class **Singular**(*maxread=1000, script_subdirectory=None, logfile=None, server=None, server_tmpdir=None*)

Interface to the Singular interpreter.

EXAMPLES: A Groebner basis example.

```
sage: R = singular.ring(0, '(x0,x1,x2)', 'lp')
sage: I = singular.ideal([ 'x0*x1*x2 -x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2', 'x0*x1-x0*x2
```

```
sage: I.groebner()
x1^2*x2^2,
x0*x2^3-x1^2*x2^2+x1*x2^3,
x0*x1-x0*x2-x1*x2,
x0^2*x2-x0*x1*x2
```

AUTHORS:

- David Joyner and William Stein

LIB(*lib, reload=False*)

Load the Singular library named lib.

Note that if the library was already loaded during this session it is not reloaded unless the optional reload argument is True (the default is False).

EXAMPLES:

```
sage: singular.lib('sing.lib')
sage: singular.lib('sing.lib', reload=True)
```

clear(*var*)

Clear the variable named var.

EXAMPLES:

```
sage: singular.set('int', 'x', '2')
sage: singular.get('x')
'2'
sage: singular.clear('x')
sage: singular.get('x')
''x''
```

console()

EXAMPLES:

```
sage: singular_console() #not tested
                     SINGULAR                                /  Development
 A Computer Algebra System for Polynomial Computations     /   version 3-0-4
                                                         0<
     by: G.-M. Greuel, G. Pfister, H. Schoenemann          \   Nov 2007
FB Mathematik der Universitaet, D-67653 Kaiserslautern
```

cputime(*t=None*)

Returns the amount of CPU time that the Singular session has used. If `t` is not None, then it returns the difference between the current CPU time and `t`.

EXAMPLES:

```
sage: t = singular.cputime()
sage: R = singular.ring(0, '(x0,x1,x2)', 'lp')
sage: I = singular.ideal([ 'x0*x1*x2 -x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2', 'x0*x1-x
sage: gb = I.groebner()
sage: singular.cputime(t) #random
0.02
```

current_ring()

Returns the current ring of the runnging Singular session.

EXAMPLES:

```
sage: r = PolynomialRing(GF(127),3,'xyz', order='invlex')
sage: r._singular_()
//   characteristic : 127
```

```
//   number of vars : 3
//        block   1 : ordering rp
//                  : names    x y z
//        block   2 : ordering C
sage: singular.current_ring()
//   characteristic : 127
//   number of vars : 3
//        block   1 : ordering rp
//                  : names    x y z
//        block   2 : ordering C
```

current_ring_name()

Returns the Singular name of the currently active ring in Singular.

OUTPUT: currently active ring's name

EXAMPLES:

```
sage: r = PolynomialRing(GF(127),3,'xyz')
sage: r._singular_().name() == singular.current_ring_name()
True
```

eval(*x, allow_semicolon=True, strip=True, **kwds*)

Send the code x to the Singular interpreter and return the output as a string.

INPUT:

- x - string (of code)
- allow_semicolon - default: False; if False then raise a TypeError if the input line contains a semicolon.
- strip - ignored

EXAMPLES:

```
sage: singular.eval('2 > 1')
'1'
sage: singular.eval('2 + 2')
'4'
```

if the verbosity level is > 1 comments are also printed and not only returned.

```
sage: r = singular.ring(0,'(x,y,z)','dp')
sage: i = singular.ideal(['x^2','y^2','z^2'])
sage: s = i.std()
sage: singular.eval('hilb(%s)'%(s.name()))
'// 1 t^0\n// -3 t^2\n// 3 t^4\n// -1 t^6\n\n// 1 t^0\n//
3 t^1\n// 3 t^2\n// 1 t^3\n// dimension (affine) = 0\n//
degree (affine) = 8'

sage: set_verbose(1)
sage: o = singular.eval('hilb(%s)'%(s.name()))
//         1 t^0
//        -3 t^2
//         3 t^4
//        -1 t^6
//         1 t^0
//         3 t^1
//         3 t^2
//         1 t^3
// dimension (affine) = 0
// degree (affine)  = 8
```

This is mainly useful if this method is called implicitly. Because then intermediate results, debugging outputs and printed statements are printed

```
sage: o = s.hilb()
//         1 t^0
//        -3 t^2
//         3 t^4
//        -1 t^6
//         1 t^0
//         3 t^1
//         3 t^2
//         1 t^3
// dimension (affine) = 0
// degree (affine)  = 8
// ** right side is not a datum, assignment ignored
```

rather than ignored

```
sage: set_verbose(0)
sage: o = s.hilb()
```

get(*var*)

Get string representation of variable named var.

EXAMPLES:

```
sage: singular.set('int', 'x', '2')
sage: singular.get('x')
'2'
```

ideal(**gens*)

Return the ideal generated by gens.

INPUT:

- gens - list or tuple of Singular objects (or objects that can be made into Singular objects via evaluation)

OUTPUT: the Singular ideal generated by the given list of gens

EXAMPLES: A Groebner basis example done in a different way.

```
sage: _ = singular.eval("ring R=0,(x0,x1,x2),lp")
sage: i1 = singular.ideal([ 'x0*x1*x2 -x0^2*x2', 'x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2', 'x0*x1-
sage: i1
-x0^2*x2+x0*x1*x2,
x0^2*x1*x2-x0*x1^2*x2-x0*x1*x2^2,
x0*x1-x0*x2-x1*x2

sage: i2 = singular.ideal('groebner(%s);'%i1.name())
sage: i2
x1^2*x2^2,
x0*x2^3-x1^2*x2^2+x1*x2^3,
x0*x1-x0*x2-x1*x2,
x0^2*x2-x0*x1*x2
```

lib(*lib, reload=False*)

Load the Singular library named lib.

Note that if the library was already loaded during this session it is not reloaded unless the optional reload argument is True (the default is False).

EXAMPLES:

```
sage: singular.lib('sing.lib')
sage: singular.lib('sing.lib', reload=True)
```

list(*x*)

Creates a list in Singular from a Sage list x.

EXAMPLES:

```
sage: singular.list([1,2])
[1]:
   1
[2]:
   2
```

load(*lib, reload=False*)

Load the Singular library named lib.

Note that if the library was already loaded during this session it is not reloaded unless the optional reload argument is True (the default is False).

EXAMPLES:

```
sage: singular.lib('sing.lib')
sage: singular.lib('sing.lib', reload=True)
```

matrix(*nrows, ncols, entries=None*)

EXAMPLES:

```
sage: singular.lib("matrix")
sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: A = singular.matrix(3,2,'1,2,3,4,5,6')
sage: A
1,2,
3,4,
5,6
sage: A.gauss_col()
2,-1,
1,0,
0,1
```

AUTHORS:

- Martin Albrecht (2006-01-14)

option(*cmd=None, val=None*)

Access to Singular's options as follows:

Syntax: option() Returns a string of all defined options.

Syntax: option('option_name') Sets an option. Note to disable an option, use the prefix no.

Syntax: option('get') Returns an intvec of the state of all options.

Syntax: option('set', intvec_expression) Restores the state of all options from an intvec (produced by option('get')).

EXAMPLES:

```
sage: singular.option()
//options: redefine loadLib usage prompt
sage: singular.option('get')
0,
10321
sage: old_options = _
sage: singular.option('noredefine')
sage: singular.option()
//options: loadLib usage prompt
sage: singular.option('set', old_options)
sage: singular.option('get')
0,
10321
```

ring(*char=0, vars='(x)', order='lp', check=True*)

Create a Singular ring and makes it the current ring.

INPUT:

- `char` - characteristic of the base ring (see examples below), which must be either 0, prime (!), or one of several special codes (see examples below).
- `vars` - a tuple or string that defines the variable names
- `order` - string - the monomial order (default: 'lp')
- `check` - if True, check primality of the characteristic if it is an integer.

OUTPUT: a Singular ring

Note: This function is *not* identical to calling the Singular `ring` function. In particular, it also attempts to "kill" the variable names, so they can actually be used without getting errors, and it sets printing of elements for this range to short (i.e., with *'s and carets).

EXAMPLES: We first declare $\mathbf{Q}[x, y, z]$ with degree reverse lexicographic ordering.

```
sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: R
//   characteristic : 0
//   number of vars : 3
//        block   1 : ordering dp
//                  : names    x y z
//        block   2 : ordering C

sage: R1 = singular.ring(32003, '(x,y,z)', 'dp')
sage: R2 = singular.ring(32003, '(a,b,c,d)', 'lp')
```

This is a ring in variables named x(1) through x(10) over the finite field of order 7:

```
sage: R3 = singular.ring(7, '(x(1..10))', 'ds')
```

This is a polynomial ring over the transcendental extension $\mathbf{Q}(a)$ of $\mathbf{Q}$:

```
sage: R4 = singular.ring('(0,a)', '(mu,nu)', 'lp')
```

This is a ring over the field of single-precision floats:

```
sage: R5 = singular.ring('real', '(a,b)', 'lp')
```

This is over 50-digit floats:

```
sage: R6 = singular.ring('(real,50)', '(a,b)', 'lp')
sage: R7 = singular.ring('(complex,50,i)', '(a,b)', 'lp')
```

To use a ring that you've defined, use the set_ring() method on the ring. This sets the ring to be the "current ring". For example,

```
sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.new('10*a')
1.000e+01*a
sage: R.set_ring()
sage: singular.new('10*a')
3*a
```

set (*type, name, value*)

Set the variable with given name to the given value.

EXAMPLES:

```
sage: singular.set('int', 'x', '2')
sage: singular.get('x')
'2'
```

set_ring (*R*)

Sets the current Singular ring to R.

EXAMPLES:

```
sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.current_ring()
//   characteristic : 0 (real)
//   number of vars : 2
//        block   1 : ordering lp
//                  : names    a b
//        block   2 : ordering C
sage: singular.set_ring(R)
sage: singular.current_ring()
//   characteristic : 7
//   number of vars : 2
//        block   1 : ordering ds
//                  : names    a b
//        block   2 : ordering C
```

setring(*R*)

Sets the current Singular ring to R.

EXAMPLES:

```
sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.current_ring()
//   characteristic : 0 (real)
//   number of vars : 2
//        block   1 : ordering lp
//                  : names    a b
//        block   2 : ordering C
sage: singular.set_ring(R)
sage: singular.current_ring()
//   characteristic : 7
//   number of vars : 2
//        block   1 : ordering ds
//                  : names    a b
//        block   2 : ordering C
```

string(*x*)

Creates a Singular string from a Sage string. Note that the Sage string has to be "double-quoted".

EXAMPLES:

```
sage: singular.string('"Sage"')
Sage
```

trait_names()

Return a list of all Singular commands.

EXAMPLES:

```
sage: singular.trait_names()
['headStand',
 ...
 'stdfglm']
```

version()

EXAMPLES:

class SingularElement(*parent, type, value, is_name=False*)

attrib(*name, value=None*)

Get and set attributs for self.

INPUT:

- `name` - string to choose the attribute
- `value` - boolean value or None for reading, (default:None)

VALUES: isSB - the standard basis property is set by all commands computing a standard basis like groebner, std, stdhilb etc.; used by lift, dim, degree, mult, hilb, vdim, kbase isHomog - the weight vector for homogeneous or quasihomogeneous ideals/modules isCI - complete intersection property isCM - Cohen-Macaulay property rank - set the rank of a module (see nrows) withSB - value of type ideal, resp. module, is std withHilb - value of type intvec is hilb(_,1) (see hilb) withRes - value of type list is a free resolution withDim - value of type int is the dimension (see dim) withMult - value of type int is the multiplicity (see mult)

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([z^2, y*z, y^2, x*z, x*y, x^2])
sage: Ibar = I._singular_()
sage: Ibar.attrib('isSB')
0
sage: singular.eval('vdim(%s)'%Ibar.name()) # sage7 name is random
// ** sage7 is no standard basis
4
sage: Ibar.attrib('isSB',1)
sage: singular.eval('vdim(%s)'%Ibar.name())
'4'
```

sage_flattened_str_list()

EXAMPLES:

```
sage: R=singular.ring(0,'(x,y)','dp')
sage: RL = R.ringlist()
sage: RL.sage_flattened_str_list()
['0', 'x', 'y', 'dp', '1,1', 'C', '0', '_[1]=0']
```

sage_matrix(*R, sparse=True*)

Returns Sage matrix for self

EXAMPLES:

```
sage: R = singular.ring(0, '(x,y,z)', 'dp')
sage: A = singular.matrix(2,2)
sage: A.sage_matrix(ZZ)
[0 0]
[0 0]
sage: A.sage_matrix(RDF)
[0.0 0.0]
[0.0 0.0]
```

sage_poly(*R, kcache=None*)

Returns a Sage polynomial in the ring r matching the provided poly which is a singular polynomial.

INPUT:

- `R` - PolynomialRing: you *must* take care it matches the current singular ring as, e.g., returned by singular.current_ring()
- `kcache` - (default: None); an optional dictionary for faster finite field lookups, this is mainly useful for finite extension fields

OUTPUT: MPolynomial

EXAMPLES:

```
sage: R = PolynomialRing(GF(2^8,'a'),2,'xy')
sage: f=R('a^20*x^2*y+a^10+x')
sage: f._singular_().sage_poly(R)==f
True
sage: R = PolynomialRing(GF(2^8,'a'),1,'x')
sage: f=R('a^20*x^3+x^2+a^10')
sage: f._singular_().sage_poly(R)==f
True

sage: P.<x,y> = PolynomialRing(QQ, 2)
sage: f = x*y**3 - 1/9 * x + 1; f
x*y^3 - 1/9*x + 1
sage: singular(f)
x*y^3-1/9*x+1
sage: P(singular(f))
x*y^3 - 1/9*x + 1
```

AUTHOR:

•Martin Albrecht (2006-05-18)

Note: For very simple polynomials `eval(SingularElement.sage_polystring())` is faster than SingularElement.sage_poly(R), maybe we should detect the crossover point (in dependence of the string length) and choose an appropriate conversion strategy

sage_polystring()

If this Singular element is a polynomial, return a string representation of this polynomial that is suitable for evaluation in Python. Thus * is used for multiplication and ** for exponentiation. This function is primarily used internally.

The short=0 option *must* be set for the parent ring or this function will not work as expected. This option is set by default for rings created using `singular.ring` or set using `ring_name.set_ring()`.

EXAMPLES:

```
sage: R = singular.ring(0,'(x,y)')
sage: f = singular('x^3 + 3*y^11 + 5')
sage: f
x^3+3*y^11+5
sage: f.sage_polystring()
'x**3+3*y**11+5'
```

sage_structured_str_list()

If self is a Singular list of lists of Singular elements, returns corresponding Sage list of lists of strings.

EXAMPLES:

```
sage: R=singular.ring(0,'(x,y)','dp')
sage: RL=R.ringlist()
sage: RL
[1]:
   0
[2]:
   [1]:
      x
   [2]:
      y
[3]:
   [1]:
      [1]:
         dp
      [2]:
         1,1
```

```
      [2]:
         [1]:
            C
         [2]:
            0
   [4]:
      _[1]=0
   sage: RL.sage_structured_str_list()
   ['0', ['x', 'y'], [['dp', '1,\n1 '], ['C', '0 ']], '0']
```

set_ring()

Sets the current ring in Singular to be self.

EXAMPLES:

```
sage: R = singular.ring(7, '(a,b)', 'ds')
sage: S = singular.ring('real', '(a,b)', 'lp')
sage: singular.current_ring()
//   characteristic : 0 (real)
//   number of vars : 2
//        block   1 : ordering lp
//                  : names    a b
//        block   2 : ordering C
sage: R.set_ring()
sage: singular.current_ring()
//   characteristic : 7
//   number of vars : 2
//        block   1 : ordering ds
//                  : names    a b
//        block   2 : ordering C
```

trait_names()

Returns the possible tab-completions for self. In this case, we just return all the tab completions for the Singular object.

EXAMPLES:

```
sage: R = singular.ring(0,'(x,y)','dp')
sage: R.trait_names()
['headStand',
 ...
 'stdfglm']
```

type()

Returns the internal type of this element.

EXAMPLES:

```
sage: R = PolynomialRing(GF(2^8,'a'),2,'x')
sage: R._singular_().type()
'ring'
sage: fs = singular('x0^2','poly')
sage: fs.type()
'poly'
```

class SingularFunction(*parent, name*)

class SingularFunctionElement(*obj, name*)

is_SingularElement(*x*)

Returns True is x is of type `SingularElement`.

EXAMPLES:

```
sage: from sage.interfaces.singular import is_SingularElement
sage: is_SingularElement(singular(2))
True
sage: is_SingularElement(2)
False
```

reduce_load()

Note that this returns an invalid Singular object!

EXAMPLES:

```
sage: from sage.interfaces.singular import reduce_load
sage: reduce_load()
(invalid object -- defined in terms of closed session)
```

reduce_load_Singular()

EXAMPLES:

```
sage: from sage.interfaces.singular import reduce_load_Singular
sage: reduce_load_Singular()
Singular
```

singular_console()

Spawn a new Singular command-line session.

EXAMPLES:

```
sage: singular_console() #not tested
                     SINGULAR                                /  Development
 A Computer Algebra System for Polynomial Computations     /   version 3-0-4
                                                         0<
     by: G.-M. Greuel, G. Pfister, H. Schoenemann          \   Nov 2007
FB Mathematik der Universitaet, D-67653 Kaiserslautern
```

singular_version()

Returns the version of Singular being used.

EXAMPLES:

13.16 The Tachyon Ray Tracer

AUTHOR:

- John E. Stone

class TachyonRT()

The Tachyon Ray Tracer

tachyon_rt(model, outfile='sage.png', verbose=1, block=True, extra_opts='')

INPUT:

- `model` - a string that describes a 3d model in the Tachyon modeling format. Type tachyon_rt.help() for a description of this format.
- `outfile` - (default: 'sage.png') output filename; the extension of the filename determines the type. Supported types include:

- –tga - 24-bit (uncompressed)
- –bmp - 24-bit Windows BMP (uncompressed)
- –ppm - 24-bit PPM (uncompressed)
- –rgb - 24-bit SGI RGB (uncompressed)
- –png - 24-bit PNG (compressed, lossless)

•verbose - integer; (default: 1)

- –0 - silent
- –1 - some output
- –2 - very verbose output

•block - bool (default: True); if False, run the rendering command in the background.

•extra_opts - passed directly to tachyon command line. Use tachyon_rt.usage() to see some of the possibilities.

OUTPUT:

•Some text may be displayed onscreen.

•The file outfile is created.

EXAMPLES:

AUTHORS:

•John E. Stone

help()

usage()

CHAPTER

FOURTEEN

C/C++ LIBRARY INTERFACES

An underlying philosophy in the development of Sage is that it should provide unified library-level access to the some of the best GPL'd C/C++ libraries. Currently Sage provides some access to MWRANK, NTL, PARI, and Hanke, each of which are included with Sage.

The interfaces are implemented via shared libraries and data is moved between systems purely in memory. In particular, there is no interprocess interpreter parsing (e.g., `expect`), since everything is linked together and run as a single process. This is much more robust and efficient than using `expect`.

Each of these interfaces is used by other parts of Sage. For example, mwrank is used by the elliptic curves module to compute ranks of elliptic curves, and PARI is used for computation of class groups. It is thus probably not necessary for a casual user of Sage to be aware of the modules described in this chapter.

14.1 PARI C-library interface

AUTHORS:

- William Stein (2006-03-01): updated to work with PARI 2.2.12-beta (this involved changing almost every doc string, among other things; the precision behavior of PARI seems to change from any version to the next...).
- William Stein (2006-03-06): added newtonpoly
- Justin Walker: contributed some of the function definitions
- Gonzalo Tornaria: improvements to conversions; much better error handling.

EXAMPLES:

```
sage: pari('5! + 10/x')
(120*x + 10)/x
sage: pari('intnum(x=0,13,sin(x)+sin(x^2) + x)')
85.1885681951527
sage: f = pari('x^3-1')
sage: v = f.factor(); v
[x - 1, 1; x^2 + x + 1, 1]
sage: v[0]   # indexing is 0-based unlike in GP.
[x - 1, x^2 + x + 1]~
sage: v[1]
[1, 1]~
```

Arithmetic obeys the usual coercion rules.

```
sage: type(pari(1) + 1)
<type 'sage.libs.pari.gen.gen'>
sage: type(1 + pari(1))
<type 'sage.libs.pari.gen.gen'>
```

GUIDE TO REAL PRECISION AND THE PARI LIBRARY

The default real precision in communicating with the Pari library is the same as the default Sage real precision, which is 53 bits. Inexact Pari objects are therefore printed by default to 15 decimal digits (even if they are actually more precise).

Default precision example (53 bits, 15 significant decimals):

```
sage: a = pari(1.23); a
1.23000000000000
sage: a.sin()
0.942488801931698
```

Example with custom precision of 200 bits (60 significant decimals):

```
sage: R = RealField(200)
sage: a = pari(R(1.23)); a   # only 15 significant digits printed
1.23000000000000
sage: R(a)          # but the number is known to precision of 200 bits
1.2300000000000000000000000000000000000000000000000000000000000000000000000000
sage: a.sin()       # only 15 significant digits printed
0.942488801931698
sage: R(a.sin())    # but the number is known to precision of 200 bits
0.94248880193169751002382356538924454146128740562765030213504
```

It is possible to change the number of printed decimals:

```
sage: R = RealField(200)     # 200 bits of precision in computations
sage: old_prec = pari.set_real_precision(60)  # 60 decimals printed
sage: a = pari(R(1.23)); a
1.23000000000000000000000000000000000000000000000000000000000000000000000000000
sage: a.sin()
0.94248880193169751002382356538924454146128740562765030213503 8
sage: pari.set_real_precision(old_prec)  # restore the default printing behavior
60
```

Unless otherwise indicated in the docstring, most Pari functions that return inexact objects use the precision of their arguments to decide the precision of the computation. However, if some of these arguments happen to be exact numbers (integers, rationals, etc.), an optional parameter indicates the precision (in bits) to which these arguments should be converted before the computation. If this precision parameter is missing, the default precision of 53 bits is used. The following first converts 2 into a real with 53-bit precision:

```
sage: R = RealField()
sage: R(pari(2).sin())
0.909297426825682
```

We can ask for a better precision using the optional parameter:

```
sage: R = RealField(150)
sage: R(pari(2).sin(precision=150))
0.90929742682568169539601986591174484270225497
```

Warning regarding conversions Sage - Pari - Sage: Some care must be taken when juggling inexact types back and forth between Sage and Pari. In theory, calling p=pari(s) creates a Pari object p with the same precision as s; in practice, the Pari library's precision is word-based, so it will go up to the next word. For example, a default 53-bit Sage real s will be bumped up to 64 bits by adding bogus 11 bits. The function p.python() returns a Sage object with exactly the same precision as the Pari object p. So pari(s).python() is definitely not equal to s, since it has 64 bits of precision, including the bogus 11 bits. The correct way of avoiding this is to coerce pari(s).python() back into a domain with the right precision. This has to be done by the user (or by Sage functions that use Pari library functions in gen.pyx). For instance, if we want to use the Pari library to compute sqrt(pi) with a precision of 100 bits:

```
sage: R = RealField(100)
sage: s = R(pi); s
3.1415926535897932384626433833
sage: p = pari(s).sqrt()
sage: x = p.python(); x  # wow, more digits than I expected!
1.7724538509055160272981674833410973484
sage: x.prec()           # has precision 'improved' from 100 to 128?
128
sage: x == RealField(128)(pi).sqrt()  # sadly, no!
False
sage: R(x)               # x should be brought back to precision 100
1.7724538509055160272981674833
sage: R(x) == s.sqrt()
True
```

Elliptic curves and precision: If you are working with elliptic curves and want to compute with a precision other than the default 53 bits, you should use the precision parameter of ellinit():

```
sage: R = RealField(150)
sage: e = pari([0,0,0,-82,0]).ellinit(precision=150)
sage: eta1 = e.elleta()[0]
sage: R(eta1)
3.6054636014326520863839536934492002728802618
```

Number fields and precision: TODO

exception PariError

errmessage

class PariInstance()

allocatemem()
Double the *PARI* stack.

complex()
Create a new complex number, initialized from re and im.

default()

double_to_gen()

euler()
Return Euler's constant to the requested real precision (in bits).
EXAMPLES:

```
sage: pari.euler()
0.577215664901533
sage: pari.euler(precision=100).python()
0.57721566490153286060651209008 2...
```

factorial()
Return the factorial of the integer n as a PARI gen.

EXAMPLES:

```
sage: pari.factorial(0)
1
sage: pari.factorial(1)
1
sage: pari.factorial(5)
120
sage: pari.factorial(25)
15511210043330985984000000
```

get_debug_level()
Set the debug PARI C library variable.

get_real_precision()
Returns the current PARI default real precision.

This is used both for creation of new objects from strings and for printing. It is the number of digits *IN DECIMAL* in which real numbers are printed. It also determines the precision of objects created by parsing strings (e.g. pari('1.2')), which is *not* the normal way of creating new pari objects in Sage. It has *no* effect on the precision of computations within the pari library.

get_series_precision()

getrand()
Returns Pari's current random number seed.

EXAMPLES:

```
sage: pari.setrand(50)
sage: pari.getrand()
50
sage: pari.pari_rand31()
621715893
sage: pari.getrand()
621715893
```

init_primes()
Recompute the primes table including at least all primes up to M (but possibly more).

EXAMPLES:

```
sage: pari.init_primes(200000)
```

listcreate()
listcreate(n): return an empty pari list of maximal length n.

EXAMPLES:

```
sage: pari.listcreate(20)
List([])
```

matrix()
matrix(long m, long n, entries=None): Create and return the m x n PARI matrix with given list of entries.

new_with_bits_prec()
pari.new_with_bits_prec(self, s, precision) creates s as a PARI gen with (at most) precision *bits* of precision.

nth_prime()

pari_rand31()
Returns a random number from Pari's random number generator.

Warning: You probably don't want to use this; it's a very poor random number generator. Sage exposes it only as a way to test `getrand()` and `setrand()`.

pari_version()

pi()

Return the value of the constant pi to the requested real precision (in bits).

EXAMPLES:

```
sage: pari.pi()
3.14159265358979
sage: pari.pi(precision=100).python()
3.1415926535897932384626433832...
```

polcyclo()

polcyclo(n, v=x): cyclotomic polynomial of degree n, in variable v.

EXAMPLES:

```
sage: pari.polcyclo(8)
x^4 + 1
sage: pari.polcyclo(7, 'z')
z^6 + z^5 + z^4 + z^3 + z^2 + z + 1
sage: pari.polcyclo(1)
x - 1
```

pollegendre()

pollegendre(n, v=x): Legendre polynomial of degree n (n C-integer), in variable v.

EXAMPLES:

```
sage: pari.pollegendre(7)
429/16*x^7 - 693/16*x^5 + 315/16*x^3 - 35/16*x
sage: pari.pollegendre(7, 'z')
429/16*z^7 - 693/16*z^5 + 315/16*z^3 - 35/16*z
sage: pari.pollegendre(0)
1
```

polsubcyclo()

polsubcyclo(n, d, v=x): return the pari list of polynomial(s) defining the sub-abelian extensions of degree d of the cyclotomic field $\mathbf{Q}(\zeta_n)$, where d divides $\phi(n)$.

EXAMPLES:

```
sage: pari.polsubcyclo(8, 4)
[x^4 + 1]
sage: pari.polsubcyclo(8, 2, 'z')
[z^2 - 2, z^2 + 1, z^2 + 2]
sage: pari.polsubcyclo(8, 1)
[x - 1]
sage: pari.polsubcyclo(8, 3)
[]
```

poltchebi()

poltchebi(n, v=x): Chebyshev polynomial of the first kind of degree n, in variable v.

EXAMPLES:

```
sage: pari.poltchebi(7)
64*x^7 - 112*x^5 + 56*x^3 - 7*x
sage: pari.poltchebi(7, 'z')
64*z^7 - 112*z^5 + 56*z^3 - 7*z
sage: pari.poltchebi(0)
1
```

polzagier()

prime_list()

prime_list(n): returns list of the first n primes

To extend the table of primes use pari.init_primes(M).

INPUT:

- n - C long

OUTPUT:

- gen - PARI list of first n primes

EXAMPLES:

```
sage: pari.prime_list(0)
[]
sage: pari.prime_list(-1)
[]
sage: pari.prime_list(3)
[2, 3, 5]
sage: pari.prime_list(10)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]
sage: pari.prime_list(20)
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71]
sage: len(pari.prime_list(1000))
1000
```

primes_up_to_n()

Return the primes = n as a pari list.

EXAMPLES:

```
sage: pari.primes_up_to_n(1)
[]
sage: pari.primes_up_to_n(20)
[2, 3, 5, 7, 11, 13, 17, 19]
```

read()

Read a script from the named filename into the interpreter, where s is a string. The functions defined in the script are then available for use from Sage/PARI.

EXAMPLE:

If foo.gp is a script that contains

```
{foo(n) =
    n^2
}
```

and you type `read("foo.gp")`, then the command `pari("foo(12)")` will create the Python/PARI gen which is the integer 144.

CONSTRAINTS: The PARI script must *not* contain the following function calls:

print, default, ??? (please report any others that cause trouble)

set_debug_level()

Set the debug PARI C library variable.

set_real_precision()

Sets the PARI default real precision.

This is used both for creation of new objects from strings and for printing. It is the number of digits *IN DECIMAL* in which real numbers are printed. It also determines the precision of objects created by parsing strings (e.g. pari('1.2')), which is *not* the normal way of creating new pari objects in Sage. It has *no* effect on the precision of computations within the pari library.

Returns the previous PARI real precision.

set_series_precision()

setrand()

Sets Pari's current random number seed.

This should not be called directly; instead, use Sage's global random number seed handling in `sage.misc.randstate` and call `current_randstate().set_seed_pari()`.

EXAMPLES:

```
sage: pari.setrand(12345)
sage: pari.getrand()
12345
```

vector()

vector(long n, entries=None): Create and return the length n PARI vector with given list of entries.

EXAMPLES:

```
sage: pari.vector(5, [1, 2, 5, 4, 3])
[1, 2, 5, 4, 3]
sage: pari.vector(2, [x, 1])
[x, 1]
sage: pari.vector(2, [x, 1, 5])
...
IndexError: length of entries (=3) must equal n (=2)
```

class gen()

Python extension class that models the PARI GEN type.

Col()

Col(x): Transforms the object x into a column vector.

The vector will have only one component, except in the following cases:

- When x is a vector or a quadratic form, the resulting vector is the initial object considered as a column vector.
- When x is a matrix, the resulting vector is the column of row vectors comprising the matrix.
- When x is a character string, the result is a column of individual characters.
- When x is a polynomial, the coefficients of the vector start with the leading coefficient of the polynomial.
- When x is a power series, only the significant coefficients are taken into account, but this time by increasing order of degree.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari(1.5).Col()
[1.50000000000000]~
sage: pari([1,2,3,4]).Col()
[1, 2, 3, 4]~
sage: pari('[1,2; 3,4]').Col()
[[1, 2], [3, 4]]~
sage: pari('"Sage"').Col()
["S", "a", "g", "e"]~
sage: pari('3*x^3 + x').Col()
[3, 0, 1, 0]~
sage: pari('x + 3*x^3 + O(x^5)').Col()
[1, 0, 3, 0]~
```

List()

List(x): transforms the PARI vector or list x into a list.

EXAMPLES:

```
sage: v = pari([1,2,3])
sage: v
[1, 2, 3]
sage: v.type()
't_VEC'
sage: w = v.List()
sage: w
List([1, 2, 3])
sage: w.type()
't_LIST'
```

Mat()

Mat(x): Returns the matrix defined by x.

- If x is already a matrix, a copy of x is created and returned.
- If x is not a vector or a matrix, this function returns a 1x1 matrix.
- If x is a row (resp. column) vector, this functions returns a 1-row (resp. 1-column) matrix, *unless* all elements are column (resp. row) vectors of the same length, in which case the vectors are concatenated sideways and the associated big matrix is returned.

INPUT:

- x - gen

OUTPUT:

- gen - a PARI matrix

EXAMPLES:

```
sage: x = pari(5)
sage: x.type()
't_INT'
sage: y = x.Mat()
sage: y
Mat(5)
sage: y.type()
't_MAT'
sage: x = pari('[1,2;3,4]')
sage: x.type()
't_MAT'
sage: x = pari('[1,2,3,4]')
sage: x.type()
't_VEC'
sage: y = x.Mat()
sage: y
Mat([1, 2, 3, 4])
sage: y.type()
't_MAT'

sage: v = pari('[1,2;3,4]').Vec(); v
[[1, 3]~, [2, 4]~]
sage: v.Mat()
[1, 2; 3, 4]
sage: v = pari('[1,2;3,4]').Col(); v
[[1, 2], [3, 4]]~
sage: v.Mat()
[1, 2; 3, 4]
```

Mod()

Mod(x, y): Returns the object x modulo y, denoted Mod(x, y).

The input y must be a an integer or a polynomial:

- If y is an INTEGER, x must also be an integer, a rational number, or a p-adic number compatible with the modulus y.
- If y is a POLYNOMIAL, x must be a scalar (which is not a polmod), a polynomial, a rational function, or a power series.

Warning: This function is not the same as `x % y` which is an integer or a polynomial.

INPUT:

- `x` - gen
- `y` - integer or polynomial

OUTPUT:

- `gen` - intmod or polmod

EXAMPLES:

```
sage: z = pari(3)
sage: x = z.Mod(pari(7))
sage: x
Mod(3, 7)
sage: x^2
Mod(2, 7)
sage: x^100
Mod(4, 7)
sage: x.type()
't_INTMOD'

sage: f = pari("x^2 + x + 1")
sage: g = pari("x")
sage: a = g.Mod(f)
sage: a
Mod(x, x^2 + x + 1)
sage: a*a
Mod(-x - 1, x^2 + x + 1)
sage: a.type()
't_POLMOD'
```

Pol()

Pol(x, v): convert x into a polynomial with main variable v and return the result.

- If x is a scalar, returns a constant polynomial.
- If x is a power series, the effect is identical to `truncate`, i.e. it chops off the $O(X^k)$.
- If x is a vector, this function creates the polynomial whose coefficients are given in x, with x[0] being the leading coefficient (which can be zero).

Warning: This is *not* a substitution function. It will not transform an object containing variables of higher priority than v:

```
sage: pari('x+y').Pol('y')
...
PariError:  (8)
```

INPUT:

- x - gen
- v - (optional) which variable, defaults to 'x'

OUTPUT:

- gen - a polynomial

EXAMPLES:

```
sage: v = pari("[1,2,3,4]")
sage: f = v.Pol()
sage: f
x^3 + 2*x^2 + 3*x + 4
sage: f*f
x^6 + 4*x^5 + 10*x^4 + 20*x^3 + 25*x^2 + 24*x + 16

sage: v = pari("[1,2;3,4]")
sage: v.Pol()
[1, 3]~*x + [2, 4]~
```

Polrev()

Polrev(x, v): Convert x into a polynomial with main variable v and return the result. This is the reverse of Pol if x is a vector, otherwise it is identical to Pol. By "reverse" we mean that the coefficients are reversed.

INPUT:

- x - gen

OUTPUT:

- gen - a polynomial

EXAMPLES:

```
sage: v = pari("[1,2,3,4]")
sage: f = v.Polrev()
sage: f
4*x^3 + 3*x^2 + 2*x + 1
sage: v.Pol()
x^3 + 2*x^2 + 3*x + 4
sage: v.Polrev('y')
4*y^3 + 3*y^2 + 2*y + 1
```

Note that Polrev does *not* reverse the coefficients of a polynomial!

```
sage: f
4*x^3 + 3*x^2 + 2*x + 1
sage: f.Polrev()
4*x^3 + 3*x^2 + 2*x + 1
sage: v = pari("[1,2;3,4]")
sage: v.Polrev()
[2, 4]~*x + [1, 3]~
```

Qfb()

Qfb(a,b,c,D=0.): Returns the binary quadratic form

$$ax^2 + bxy + cy^2.$$

The optional D is 0 by default and initializes Shanks's distance if $b^2 - 4ac > 0$.

Note: Negative definite forms are not implemented, so use their positive definite counterparts instead. (I.e., if f is a negative definite quadratic form, then -f is positive definite.)

INPUT:

- a - gen
- b - gen

- c - gen
- D - gen (optional, defaults to 0)

OUTPUT:

- gen - binary quadratic form

EXAMPLES:

```
sage: pari(3).Qfb(7, 2)
Qfb(3, 7, 2, 0.E-19)
```

Ser()

Ser(x,v=x): Create a power series from x with main variable v and return the result.

- If x is a scalar, this gives a constant power series with precision given by the default series precision, as returned by get_series_precision().
- If x is a polynomial, the precision is the greatest of get_series_precision() and the degree of the polynomial.
- If x is a vector, the precision is similarly given, and the coefficients of the vector are understood to be the coefficients of the power series starting from the constant term (i.e. the reverse of the function Pol).

Warning: This is *not* a substitution function. It will not transform an object containing variables of higher priority than v.

INPUT:

- x - gen
- v - PARI variable (default: x)

OUTPUT:

- gen - PARI object of PARI type t_SER

EXAMPLES:

```
sage: pari(2).Ser()
2 + O(x^16)
sage: x = pari([1,2,3,4,5])
sage: x.Ser()
1 + 2*x + 3*x^2 + 4*x^3 + 5*x^4 + O(x^5)
sage: f = x.Ser('v'); print f
1 + 2*v + 3*v^2 + 4*v^3 + 5*v^4 + O(v^5)
sage: pari(1)/f
1 - 2*v + v^2 + O(v^5)
sage: pari(1).Ser()
1 + O(x^16)
```

Set()

Set(x): convert x into a set, i.e. a row vector of strings in increasing lexicographic order.

INPUT:

- x - gen

OUTPUT:

- gen - a vector of strings in increasing lexicographic order.

EXAMPLES:

```
sage: pari([1,5,2]).Set()
["1", "2", "5"]
sage: pari([]).Set()      # the empty set
```

```
[]
sage: pari([1,1,-1,-1,3,3]).Set()
["-1", "1", "3"]
sage: pari(1).Set()
["1"]
sage: pari('1/(x*y)').Set()
["1/(y*x)"]
sage: pari('["bc","ab","bc"]').Set()
["ab", "bc"]
```

Str()

Str(self): Return the print representation of self as a PARI object.

INPUT:

- `self` - gen

OUTPUT:

- `gen` - a PARI gen of type t_STR, i.e., a PARI string

EXAMPLES:

```
sage: pari([1,2,['abc',1]]).Str()
[1, 2, [abc, 1]]
sage: pari([1,1, 1.54]).Str()
[1, 1, 1.540000000000000]
sage: pari(1).Str()        # 1 is automatically converted to string rep
1
sage: x = pari('x')        # PARI variable "x"
sage: x.Str()              # is converted to string rep.
x
sage: x.Str().type()
't_STR'
```

Strchr()

Strchr(x): converts x to a string, translating each integer into a character (in ASCII).

Note: `Vecsmall()` is (essentially) the inverse to `Strchr()`.

INPUT:

- `x` - PARI vector of integers

OUTPUT:

- `gen` - a PARI string

EXAMPLES:

```
sage: pari([65,66,123]).Strchr()
AB{
sage: pari('"Sage"').Vecsmall()    # pari('"Sage"') --> PARI t_STR
Vecsmall([83, 97, 103, 101])
sage: _.Strchr()
Sage
sage: pari([83, 97, 103, 101]).Strchr()
Sage
```

Strexpand()

Strexpand(x): Concatenate the entries of the vector x into a single string, performing tilde expansion.

Note: I have no clue what the point of this function is. - William

Strtex()

Strtex(x): Translates the vector x of PARI gens to TeX format and returns the resulting concatenated strings as a PARI t_STR.

INPUT:

- x - gen

OUTPUT:

- gen - PARI t_STR (string)

EXAMPLES:

```
sage: v=pari('x^2')
sage: v.Strtex()
x^2
sage: v=pari(['1/x^2','x'])
sage: v.Strtex()
\frac{1}{x^2}x
sage: v=pari(['1 + 1/x + 1/(y+1)','x-1'])
sage: v.Strtex()
\frac{ \left(y
 + 2\right)  x
 + \left(y
 + 1\right) }{ \left(y
 + 1\right)  x}x
 - 1
```

Vec()

Vec(x): Transforms the object x into a vector.

INPUT:

- x - gen

OUTPUT:

- gen - of PARI type t_VEC

EXAMPLES:

```
sage: pari(1).Vec()
[1]
sage: pari('x^3').Vec()
[1, 0, 0, 0]
sage: pari('x^3 + 3*x - 2').Vec()
[1, 0, 3, -2]
sage: pari([1,2,3]).Vec()
[1, 2, 3]
sage: pari('ab').Vec()
[1, 0]
```

Vecrev()

Vecrev(x): Transforms the object x into a vector. Identical to Vec(x) except when x is - a polynomial, this is the reverse of Vec. - a power series, this includes low-order zero coefficients. - a Laurent series, raises an exception

INPUT:

- x - gen

OUTPUT:

- gen - of PARI type t_VEC

EXAMPLES:

```
sage: pari(1).Vecrev()
[1]
sage: pari('x^3').Vecrev()
[0, 0, 0, 1]
sage: pari('x^3 + 3*x - 2').Vecrev()
```

```
[-2, 3, 0, 1]
sage: pari([1, 2, 3]).Vecrev()
[1, 2, 3]
sage: pari('Col([1, 2, 3])').Vecrev()
[1, 2, 3]
sage: pari('[1, 2; 3, 4]').Vecrev()
[[1, 3]~, [2, 4]~]
sage: pari('ab').Vecrev()
[0, 1]
sage: pari('x^2 + 3*x^3 + O(x^5)').Vecrev()
[0, 0, 1, 3, 0]
sage: pari('x^-2 + 3*x^3 + O(x^5)').Vecrev()
...
ValueError: Vecrev() is not defined for Laurent series
```

Vecsmall()

Vecsmall(x): transforms the object x into a t_VECSMALL.

INPUT:

- x - gen

OUTPUT:

- gen - PARI t_VECSMALL

EXAMPLES:

```
sage: pari([1,2,3]).Vecsmall()
Vecsmall([1, 2, 3])
sage: pari('"Sage"').Vecsmall()
Vecsmall([83, 97, 103, 101])
sage: pari(1234).Vecsmall()
Vecsmall([1234])
```

abs()

Returns the absolute value of x (its modulus, if x is complex). Rational functions are not allowed. Contrary to most transcendental functions, an exact argument is not converted to a real number before applying abs and an exact result is returned if possible.

EXAMPLES:

```
sage: x = pari("-27.1")
sage: x.abs()
27.1000000000000
```

If x is a polynomial, returns -x if the leading coefficient is real and negative else returns x. For a power series, the constant coefficient is considered instead.

EXAMPLES:

```
sage: pari('x-1.2*x^2').abs()
1.20000000000000*x^2 - x
```

acos()

The principal branch of $\cos^{-1}(x)$, so that $\mathbf{R}e(\mathrm{acos}(x))$ belongs to $[0, Pi]$. If x is real and $|x| > 1$, then $\mathrm{acos}(x)$ is complex.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0.5).acos()
1.04719755119660
sage: pari(1/2).acos()
1.04719755119660
sage: pari(1.1).acos()
-0.443568254385115*I
sage: C.<i> = ComplexField()
sage: pari(1.1+i).acos()
0.849343054245252 - 1.09770986682533*I
```

acosh()

The principal branch of $\cosh^{-1}(x)$, so that $\Im(\text{acosh}(x))$ belongs to $[0, Pi]$. If x is real and $x < 1$, then $\text{acosh}(x)$ is complex.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).acosh()
1.31695789692482
sage: pari(0).acosh()
1.57079632679490*I
sage: C.<i> = ComplexField()
sage: pari(i).acosh()
0.881373587019543 + 1.57079632679490*I
```

agm()

The arithmetic-geometric mean of x and y. In the case of complex or negative numbers, the principal square root is always chosen. p-adic or power series arguments are also allowed. Note that a p-adic AGM exists only if x/y is congruent to 1 modulo p (modulo 16 for p=2). x and y cannot both be vectors or matrices.

If any of x or y is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their two precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).agm(2)
2.00000000000000
sage: pari(0).agm(1)
0
sage: pari(1).agm(2)
1.45679103104691
sage: C.<i> = ComplexField()
sage: pari(1+i).agm(-3)
-0.964731722290876 + 1.15700282952632*I
```

algdep()

EXAMPLES:

```
sage: n = pari.set_real_precision (200)
sage: w1 = pari('z1=2-sqrt(26); (z1+I)/(z1-I)')
sage: f = w1.algdep(12); f
545*x^11 - 297*x^10 - 281*x^9 + 48*x^8 - 168*x^7 + 690*x^6 - 168*x^5 + 48*x^4 - 281*x^3 - 29
sage: f(w1)
7.75513996 E-200 + 5.70672991 E-200*I     # 32-bit
3.780069700150794274 E-209 - 9.362977321012524836 E-211*I   # 64-bit
sage: f.factor()
[x, 1; x + 1, 2; x^2 + 1, 1; x^2 + x + 1, 1; 545*x^4 - 1932*x^3 + 2790*x^2 - 1932*x + 545, 1
```

```
sage: pari.set_real_precision(n)
200
```

arg()

arg(x): argument of x,such that $-\pi < \arg(x) \leq \pi$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(2+i).arg()
0.463647609000806
```

asin()

The principal branch of $\sin^{-1}(x)$, so that $\mathbf{R}e(\mathrm{asin}(x))$ belongs to $[-\pi/2, \pi/2]$. If x is real and $|x| > 1$ then $\mathrm{asin}(x)$ is complex.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(pari(0.5).sin()).asin()
0.500000000000000
sage: pari(2).asin()
1.57079632679490 + 1.31695789692482*I
```

asinh()

The principal branch of $\sinh^{-1}(x)$, so that $\Im(\mathrm{asinh}(x))$ belongs to $[-\pi/2, \pi/2]$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).asinh()
1.44363547517881
sage: C.<i> = ComplexField()
sage: pari(2+i).asinh()
1.52857091948100 + 0.427078586392476*I
```

atan()

The principal branch of $\tan^{-1}(x)$, so that $\mathbf{R}e(\mathrm{atan}(x))$ belongs to $]-\pi/2, \pi/2[$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).atan()
0.785398163397448
sage: C.<i> = ComplexField()
sage: pari(1.5+i).atan()
1.10714871779409 + 0.255412811882995*I
```

atanh()

The principal branch of $\tanh^{-1}(x)$, so that $\Im(\mathrm{atanh}(x))$ belongs to $]-\pi/2, \pi/2]$. If x is real and $|x| > 1$ then $\mathrm{atanh}(x)$ is complex.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0).atanh()
0.E-19
sage: pari(2).atanh()
0.549306144334055 + 1.57079632679490*I
```

bernfrac()

The Bernoulli number B_x, where $B_0 = 1$, $B_1 = -1/2$, $B_2 = 1/6, \ldots$, expressed as a rational number. The argument x should be of type integer.

EXAMPLES:

```
sage: pari(18).bernfrac()
43867/798
sage: [pari(n).bernfrac() for n in range(10)]
[1, -1/2, 1/6, 0, -1/30, 0, 1/42, 0, -1/30, 0]
```

bernreal()

The Bernoulli number B_x, as for the function bernfrac, but B_x is returned as a real number (with the current precision).

EXAMPLES:

```
sage: pari(18).bernreal()
54.9711779448622
```

bernvec()

Creates a vector containing, as rational numbers, the Bernoulli numbers $B_0, B_2, \ldots, B_{2x}$. This routine is obsolete. Use bernfrac instead each time you need a Bernoulli number in exact form.

Note: this routine is implemented using repeated independent calls to bernfrac, which is faster than the standard recursion in exact arithmetic.

EXAMPLES:

```
sage: pari(8).bernvec()
[1, 1/6, -1/30, 1/42, -1/30, 5/66, -691/2730, 7/6, -3617/510]
sage: [pari(2*n).bernfrac() for n in range(9)]
[1, 1/6, -1/30, 1/42, -1/30, 5/66, -691/2730, 7/6, -3617/510]
```

besselh1()

The H^1-Bessel function of index ν and argument x.

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).besselh1(3)
0.486091260585891 - 0.160400393484924*I
```

besselh2()

The H^2-Bessel function of index ν and argument x.

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).besselh2(3)
0.486091260585891 + 0.160400393484924*I
```

besseli()

Bessel I function (Bessel function of the second kind), with index ν and argument x. If x converts to a power series, the initial factor $(x/2)^\nu/\Gamma(\nu+1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).besseli(3)
2.24521244092995
sage: C.<i> = ComplexField()
sage: pari(2).besseli(3+i)
1.12539407613913 + 2.08313822670661*I
```

besselj()

Bessel J function (Bessel function of the first kind), with index ν and argument x. If x converts to a power series, the initial factor $(x/2)^\nu/\Gamma(\nu+1)$ is omitted (since it cannot be represented in PARI when ν is not integral).

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).besselj(3)
0.486091260585891
```

besseljh()

J-Bessel function of half integral index (Speherical Bessel function of the first kind). More precisely, besseljh(n,x) computes $J_{n+1/2}(x)$ where n must an integer, and x is any complex value. In the current implementation (PARI, version 2.2.11), this function is not very accurate when x is small.

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).besseljh(3)
0.4127100324          # 32-bit
0.412710032209716     # 64-bit
```

besselk()

nu.besselk(x, flag=0): K-Bessel function (modified Bessel function of the second kind) of index nu, which can be complex, and argument x.

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

INPUT:

- `nu` - a complex number
- `x` - real number (positive or negative)
- `flag` - default: 0 or 1: use hyperu (hyperu is much slower for small x, and doesn't work for negative x).

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(2+i).besselk(3)
0.0455907718407551 + 0.0289192946582081*I

sage: pari(2+i).besselk(-3)
-4.34870874986752 - 5.38744882697109*I

sage: pari(2+i).besselk(300, flag=1)
3.74224603319728 E-132 + 2.49071062641525 E-134*I
```

besseln()

nu.besseln(x): Bessel N function (Spherical Bessel function of the second kind) of index nu and argument x.

If nu or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(2+i).besseln(3)
-0.280775566958244 - 0.486708533223726*I
```

bezout()

binary()

binary(x): gives the vector formed by the binary digits of abs(x), where x is of type t_INT.

INPUT:

- x - gen of type t_INT

OUTPUT:

- gen - of type t_VEC

EXAMPLES:

```
sage: pari(0).binary()
[0]
sage: pari(-5).binary()
[1, 0, 1]
sage: pari(5).binary()
[1, 0, 1]
sage: pari(2005).binary()
[1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1]

sage: pari('"2"').binary()
...
TypeError: x (=2) must be of type t_INT, but is of type t_STR.
```

binomial()

binomial(x, k): return the binomial coefficient "x choose k".

INPUT:

- x - any PARI object (gen)
- k - integer

EXAMPLES:

```
sage: pari(6).binomial(2)
15
sage: pari('x+1').binomial(3)
1/6*x^3 - 1/6*x
```

```
sage: pari('2+x+O(x^2)').binomial(3)
1/3*x + O(x^2)
```

bitand()

bitand(x,y): Bitwise and of two integers x and y. Negative numbers behave as if modulo some large power of 2.

INPUT:

- x - gen (of type t_INT)
- y - coercible to gen (of type t_INT)

OUTPUT:

- gen - of type type t_INT

EXAMPLES:

```
sage: pari(8).bitand(4)
0
sage: pari(8).bitand(8)
8
sage: pari(6).binary()
[1, 1, 0]
sage: pari(7).binary()
[1, 1, 1]
sage: pari(6).bitand(7)
6
sage: pari(19).bitand(-1)
19
sage: pari(-1).bitand(-1)
-1
```

bitneg()

bitneg(x,n=-1): Bitwise negation of the integer x truncated to n bits. n=-1 (the default) represents an infinite sequence of the bit 1. Negative numbers behave as if modulo some large power of 2.

With n=-1, this function returns -n-1. With n = 0, it returns a number a such that $a \cong -n-1 \pmod{2^n}$.

INPUT:

- x - gen (t_INT)
- n - long, default = -1

OUTPUT:

- gen - t_INT

EXAMPLES:

```
sage: pari(10).bitneg()
-11
sage: pari(1).bitneg()
-2
sage: pari(-2).bitneg()
1
sage: pari(-1).bitneg()
0
sage: pari(569).bitneg()
-570
sage: pari(569).bitneg(10)
454
sage: 454 % 2^10
454
sage: -570 % 2^10
454
```

bitnegimply()

bitnegimply(x,y): Bitwise negated imply of two integers x and y, in other words, x BITAND BITNEG(y). Negative numbers behave as if modulo big power of 2.

INPUT:

- x - gen (of type t_INT)
- y - coercible to gen (of type t_INT)

OUTPUT:

- gen - of type type t_INT

EXAMPLES:

```
sage: pari(14).bitnegimply(0)
14
sage: pari(8).bitnegimply(8)
0
sage: pari(8+4).bitnegimply(8)
4
```

bitor()

bitor(x,y): Bitwise or of two integers x and y. Negative numbers behave as if modulo big power of 2.

INPUT:

- x - gen (of type t_INT)
- y - coercible to gen (of type t_INT)

OUTPUT:

- gen - of type type t_INT

EXAMPLES:

```
sage: pari(14).bitor(0)
14
sage: pari(8).bitor(4)
12
sage: pari(12).bitor(1)
13
sage: pari(13).bitor(1)
13
```

bittest()

bittest(x, long n): Returns bit number n (coefficient of 2^n in binary) of the integer x. Negative numbers behave as if modulo a big power of 2.

INPUT:

- x - gen (pari integer)

OUTPUT:

- bool - a Python bool

EXAMPLES:

```
sage: x = pari(6)
sage: x.bittest(0)
False
sage: x.bittest(1)
True
sage: x.bittest(2)
True
sage: x.bittest(3)
False
```

```
sage: pari(-3).bittest(0)
True
sage: pari(-3).bittest(1)
False
sage: [pari(-3).bittest(n) for n in range(10)]
[True, False, True, True, True, True, True, True, True, True]
```

bitxor()

bitxor(x,y): Bitwise exclusive or of two integers x and y. Negative numbers behave as if modulo big power of 2.

INPUT:

- x - gen (of type t_INT)
- y - coercible to gen (of type t_INT)

OUTPUT:

- gen - of type type t_INT

EXAMPLES:

```
sage: pari(6).bitxor(4)
2
sage: pari(0).bitxor(4)
4
sage: pari(6).bitxor(0)
6
```

bnfcertify()

bnf being as output by bnfinit, checks whether the result is correct, i.e. whether the calculation of the contents of self are correct without assuming the Generalized Riemann Hypothesis. If it is correct, the answer is 1. If not, the program may output some error message, but more probably will loop indefinitely. In *no* occasion can the program give a wrong answer (barring bugs of course): if the program answers 1, the answer is certified.

> **Warning:** By default, most of the bnf routines depend on the correctness of a heuristic assumption which is stronger than GRH. In order to obtain a provably-correct result you *must* specify $c = c_2 = 12$ for the technical optional parameters to the function. There are known counterexamples for smaller c (which is the default).

bnfinit()

bnfisintnorm()

bnfisprincipal()

bnfisunit()

bnfnarrow()

bnfunit()

ceil()

For real x: return the smallest integer = x. For rational functions: the quotient of numerator by denominator. For lists: apply componentwise.

INPUT:

- x - gen

OUTPUT:

- gen - depends on type of x

EXAMPLES:

```
sage: pari(1.4).ceil()
2
sage: pari(-1.4).ceil()
-1
sage: pari(3/4).ceil()
1
sage: pari(x).ceil()
x
sage: pari((x^2+x+1)/x).ceil()
x + 1
```

This may be unexpected: but it is correct, treating the argument as a rational function in RR(x).

```
sage: pari(x^2+5*x+2.5).ceil()
x^2 + 5*x + 2.50000000000000
```

centerlift()

centerlift(x,v): Centered lift of x. This function returns exactly the same thing as lift, except if x is an integer mod.

INPUT:

- x - gen
- v - var (default: x)

OUTPUT: gen

EXAMPLES:

```
sage: x = pari(-2).Mod(5)
sage: x.centerlift()
-2
sage: x.lift()
3
sage: f = pari('x-1').Mod('x^2 + 1')
sage: f.centerlift()
x - 1
sage: f.lift()
x - 1
sage: f = pari('x-y').Mod('x^2+1')
sage: f
Mod(x - y, x^2 + 1)
sage: f.centerlift('x')
x - y
sage: f.centerlift('y')
Mod(x - y, x^2 + 1)
```

changevar()

changevar(gen x, y): change variables of x according to the vector y.

Warning: This doesn't seem to work right at all in Sage (!). Use with caution. *STRANGE*

INPUT:

- x - gen
- y - gen (or coercible to gen)

OUTPUT: gen

EXAMPLES:

```
sage: pari('x^3+1').changevar(pari(['y']))
y^3 + 1
```

charpoly()
charpoly(A,v=x,flag=0): det(v*Id-A) = characteristic polynomial of A using the comatrix. flag is optional and may be set to 1 (use Lagrange interpolation) or 2 (use Hessenberg form), 0 being the default.

chinese()

component()
component(x, long n): Return n'th component of the internal representation of x. This function is 1-based instead of 0-based.

Note: For vectors or matrices, it is simpler to use x[n-1]. For list objects such as is output by nfinit, it is easier to use member functions.

INPUT:

- x - gen
- n - C long (coercible to)

OUTPUT: gen

EXAMPLES:

```
sage: pari([0,1,2,3,4]).component(1)
0
sage: pari([0,1,2,3,4]).component(2)
1
sage: pari([0,1,2,3,4]).component(4)
3
sage: pari('x^3 + 2').component(1)
2
sage: pari('x^3 + 2').component(2)
0
sage: pari('x^3 + 2').component(4)
1

sage: pari('x').component(0)
...
PariError:  (8)
```

concat()

conj()
conj(x): Return the algebraic conjugate of x.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari('x+1').conj()
x + 1
sage: pari('x+I').conj()
x - I
sage: pari('1/(2*x+3*I)').conj()
1/(2*x - 3*I)
sage: pari([1,2,'2-I','Mod(x,x^2+1)', 'Mod(x,x^2-2)']).conj()
[1, 2, 2 + I, Mod(-x, x^2 + 1), Mod(-x, x^2 - 2)]
sage: pari('Mod(x,x^2-2)').conj()
Mod(-x, x^2 - 2)
sage: pari('Mod(x,x^3-3)').conj()
...
PariError: incorrect type (20)
```

conjvec()

conjvec(x): Returns the vector of all conjugates of the algebraic number x. An algebraic number is a polynomial over Q modulo an irreducible polynomial.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari('Mod(1+x,x^2-2)').conjvec()
[-0.414213562373095, 2.414213562373710]~
sage: pari('Mod(x,x^3-3)').conjvec()
[1.44224957030741, -0.721124785153704 + 1.24902476648341*I, -0.721124785153704 - 1.249024766
```

contfrac()

contfrac(x,b,lmax): continued fraction expansion of x (x rational, real or rational function). b and lmax are both optional, where b is the vector of numerators of the continued fraction, and lmax is a bound for the number of terms in the continued fraction expansion.

contfracpnqn()

contfracpnqn(x): [p_n,p_n-1; q_n,q_n-1] corresponding to the continued fraction x.

copy()

cos()

The cosine function.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1.5).cos()
0.0707372016677029
sage: C.<i> = ComplexField()
sage: pari(1+i).cos()
0.833730025131149 - 0.988897705762865*I
sage: pari('x+O(x^8)').cos()
1 - 1/2*x^2 + 1/24*x^4 - 1/720*x^6 + 1/40320*x^8 + O(x^9)
```

cosh()

The hyperbolic cosine function.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1.5).cosh()
2.35240961524325
sage: C.<i> = ComplexField()
sage: pari(1+i).cosh()
0.833730025131149 + 0.988897705762865*I
sage: pari('x+O(x^8)').cosh()
1 + 1/2*x^2 + 1/24*x^4 + 1/720*x^6 + O(x^8)
```

cotan()

The cotangent of x.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(5).cotan()
-0.295812915532746
```

Computing the cotangent of π doesn't raise an error, but instead just returns a very large (positive or negative) number.

```
sage: x = RR(pi)
sage: pari(x).cotan()          # random
-8.17674825 E15
```

denominator()

denominator(x): Return the denominator of x. When x is a vector, this is the least common multiple of the denominators of the components of x.

what about poly? INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari('5/9').denominator()
9
sage: pari('(x+1)/(x-2)').denominator()
x - 2
sage: pari('2/3 + 5/8*x + 7/3*x^2 + 1/5*y').denominator()
1
sage: pari('2/3*x').denominator()
1
sage: pari('[2/3, 5/8, 7/3, 1/5]').denominator()
120
```

deriv()

dilog()

The principal branch of the dilogarithm of x, i.e. the analytic continuation of the power series $\log_2(x) = \sum_{n>=1} x^n/n^2$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).dilog()
1.64493406684823
sage: C.<i> = ComplexField()
sage: pari(1+i).dilog()
0.616850275068085 + 1.46036211675312*I
```

dirzetak()

disc()

e.disc(): return the discriminant of the elliptic curve e.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.disc()
-161051
sage: _.factor()
[-1, 1; 11, 5]
```

divrem()

divrem(x, y, v): Euclidean division of x by y giving as a 2-dimensional column vector the quotient and the remainder, with respect to v (to main variable if v is omitted).

eint1()

x.eint1(n): exponential integral E1(x):

$$\int_x^\infty \frac{e^{-t}}{t} dt$$

If n is present, output the vector [eint1(x), eint1(2*x), ..., eint1(n*x)]. This is faster than repeatedly calling eint1(i*x).

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

REFERENCE:

•See page 262, Prop 5.6.12, of Cohen's book "A Course in Computational Algebraic Number Theory".

EXAMPLES:

elementval()

elladd()

e.elladd(z0, z1): return the sum of the points z0 and z1 on this elliptic curve.

INPUT:

•e - elliptic curve E

•z0 - point on E

•z1 - point on E

OUTPUT: point on E

EXAMPLES: First we create an elliptic curve:

```
sage: e = pari([0, 1, 1, -2, 0]).ellinit()
sage: str(e)[:65]   # first part of output
'[0, 1, 1, -2, 0, 4, -4, 1, -3, 112, -856, 389, 1404928/389, [0.90'
```

Next we add two points on the elliptic curve. Notice that the Python lists are automatically converted to PARI objects so you don't have to do that explicitly in your code.

```
sage: e.elladd([1,0], [-1,1])
[-3/4, -15/8]
```

ellak()

e.ellak(n): Returns the coefficient a_n of the L-function of the elliptic curve e, i.e. the n-th Fourier coefficient of the weight 2 newform associated to e (according to Shimura-Taniyama).

> The curve e *must* be a medium or long vector of the type given by ellinit. For this function to work for every n and not just those prime to the conductor, e must be a minimal Weierstrass equation. If this is not the case, use the function ellminimalmodel first before using ellak (or you will get INCORRECT RESULTS!)

INPUT:

•e - a PARI elliptic curve.

•n - integer.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.ellak(6)
2
sage: e.ellak(2005)
2
sage: e.ellak(-1)
0
sage: e.ellak(0)
0
```

ellan()

Return the first n Fourier coefficients of the modular form attached to this elliptic curve. See ellak for more details.

INPUT:

- n - a long integer
- python_ints - bool (default is False); if True, return a list of Python ints instead of a PARI gen wrapper.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.ellan(3)
[1, -2, -1]
sage: e.ellan(20)
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1, -4, -2, 4, 0, 2]
sage: e.ellan(-1)
[]
sage: v = e.ellan(10, python_ints=True); v
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
sage: type(v)
<type 'list'>
sage: type(v[0])
<type 'int'>
```

ellap()

e.ellap(p): Returns the prime-indexed coefficient a_p of the L-function of the elliptic curve e, i.e. the p-th Fourier coefficient of the newform attached to e.

The computation uses the baby-step giant-step method and a trick due to Mestre, and requires $O(p^{1/4})$ time and $O(p^{1/4})$ storage.

If p is not prime, this function will return an incorrect answer.
The curve e must be a medium or long vector of the type given by ellinit. For this function to work for every n and not just those prime to the conductor, e must be a minimal Weierstrass equation. If this is not the case, use the function ellminimalmodel first before using ellap (or you will get INCORRECT RESULTS!)

INPUT:

- e - a PARI elliptic curve.
- p - prime integer

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.ellap(2)
-2
sage: e.ellap(2003)
4
sage: e.ellak(-1)
0
```

ellaplist()

e.ellaplist(n): Returns a PARI list of all the prime-indexed coefficients a_p (up to n) of the L-function of the elliptic curve e, i.e. the Fourier coefficients of the newform attached to e.

INPUT:

- n - a long integer
- python_ints - bool (default is False); if True, return a list of Python ints instead of a PARI gen wrapper.

The curve e must be a medium or long vector of the type given by ellinit. For this function to work for every n and not just those prime to the conductor, e must be a minimal Weierstrass equation. If this is not the case, use the function ellminimalmodel first before using ellaplist (or you will get INCORRECT RESULTS!)

INPUT:

- e - a PARI elliptic curve.
- n - an integer

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: v = e.ellaplist(10); v
[-2, -1, 1, -2]
sage: type(v)
<type 'sage.libs.pari.gen.gen'>
sage: v.type()
't_VEC'
sage: e.ellan(10)
[1, -2, -1, 2, 1, 2, -2, 0, -2, -2]
sage: v = e.ellaplist(10, python_ints=True); v
[-2, -1, 1, -2]
sage: type(v)
<type 'list'>
sage: type(v[0])
<type 'int'>
```

ellbil()

e.ellbil(z0, z1): return the value of the canonical bilinear form on z0 and z1.

INPUT:

- e - elliptic curve (assumed integral given by a minimal model, as returned by ellminimalmodel)
- z0, z1 - rational points on e

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit().ellminimalmodel()[0]
sage: e.ellbil([1, 0], [-1, 1])
0.418188984498861
```

ellchangecurve()

e.ellchangecurve(ch): return the new model (equation) for the elliptic curve e given by the change of coordinates ch.

The change of coordinates is specified by a vector ch=[u,r,s,t]; if x' and y' are the new coordinates, then $x = u^2x' + r$ and $y = u^3y' + su^2x' + t$.

INPUT:

- e - elliptic curve
- ch - change of coordinates vector with 4 entries

EXAMPLES:

```
sage: e = pari([1,2,3,4,5]).ellinit()
sage: e.ellglobalred()
[10351, [1, -1, 0, -1], 1]
sage: f = e.ellchangecurve([1,-1,0,-1])
sage: f[:5]
[1, -1, 0, 4, 3]
```

ellchangepoint()

self.ellchangepoint(y): change data on point or vector of points self on an elliptic curve according to y=[u,r,s,t]

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit()
sage: x = pari([1,0])
sage: e.ellisoncurve([1,4])
False
sage: e.ellisoncurve(x)
True
sage: f = e.ellchangecurve([1,2,3,-1])
sage: f[:5]   # show only first five entries
[6, -2, -1, 17, 8]
sage: x.ellchangepoint([1,2,3,-1])
[-1, 4]
sage: f.ellisoncurve([-1,4])
True
```

elleisnum()

om.elleisnum(k, flag=0): om=[om1,om2] being a 2-component vector giving a basis of a lattice L and k an even positive integer, computes the numerical value of the Eisenstein series of weight k. When flag is non-zero and k=4 or 6, this gives g2 or g3 with the correct normalization.

INPUT:

- `om` - gen, 2-component vector giving a basis of a lattice L
- `k` - int (even positive)
- `flag` - int (default 0)

OUTPUT:

- `gen` - numerical value of E_k

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit()
sage: om = e.omega()
sage: om
[2.49021256085506, 1.97173770155165*I]
sage: om.elleisnum(2)
-5.28864933965426
sage: om.elleisnum(4)
112.000000000000
sage: om.elleisnum(100)
2.15314248576078 E50
```

elleta()

e.elleta(): return the vector [eta1,eta2] of quasi-periods associated with the period lattice e.omega() of the elliptic curve e.

EXAMPLES:

```
sage: e = pari([0,0,0,-82,0]).ellinit()
sage: e.elleta()
[3.60546360143265, 10.8163908042980*I]
```

ellglobalred()

e.ellglobalred(): return information related to the global minimal model of the elliptic curve e.

INPUT:

- `e` - elliptic curve (returned by ellinit)

OUTPUT:

- `gen` - the (arithmetic) conductor of e
- `gen` - a vector giving the coordinate change over Q from e to its minimal integral model (see also ellminimalmodel)

- gen - the product of the local Tamagawa numbers of e

EXAMPLES:

```
sage: e = pari([0, 5, 2, -1, 1]).ellinit()
sage: e.ellglobalred()
[20144, [1, -2, 0, -1], 1]
sage: e = pari(EllipticCurve('17a').a_invariants()).ellinit()
sage: e.ellglobalred()
[17, [1, 0, 0, 0], 4]
```

ellheight()

e.ellheight(a, flag=2): return the global Neron-Tate height of the point a on the elliptic curve e.

INPUT:

- e - elliptic curve over $\mathbf{Q}$, assumed to be in a standard minimal integral model (as given by ellminimalmodel)
- a - rational point on e
- flag (optional) - specifies which algorithm to be used for computing the archimedean local height:
  - **0 - uses sigma- and theta-functions and a trick** due to J. Silverman
  - 1 - uses Tate's 4^n algorithm
  - 2 - uses Mestre's AGM algorithm (this is the default, being faster than the other two)
- precision (optional) - the precision of the result, in bits.

Note that in order to achieve the desired precision, the elliptic curve must have been created using ellinit with the desired precision.

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit().ellminimalmodel()[0]
sage: e.ellheight([1,0])
0.476711659343740
sage: e.ellheight([1,0], flag=0)
0.476711659343740
sage: e.ellheight([1,0], flag=1)
0.476711659343740
```

ellheightmatrix()

e.ellheightmatrix(x): return the height matrix for the vector x of points on the elliptic curve e.

In other words, it returns the Gram matrix of x with respect to the height bilinear form on e (see ellbil).

INPUT:

- e - elliptic curve over $\mathbf{Q}$, assumed to be in a standard minimal integral model (as given by ellminimalmodel)
- x - vector of rational points on e

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit().ellminimalmodel()[0]
sage: e.ellheightmatrix([[1,0], [-1,1]])
[0.476711659343740, 0.418188984498861; 0.418188984498861, 0.686667083305587]
```

ellinit()

Return the Pari elliptic curve object with Weierstrass coefficients given by self, a list with 5 elements.

INPUT:

- self - a list of 5 coefficients
- flag (optional, default: 0) - if 0, ask for a Pari ell structure with 19 components; if 1, ask for a Pari sell structure with only the first 13 components

- precision (optional, default: 0) - the real precision to be used in the computation of the components of the Pari (s)ell structure; if 0, use the default 53 bits.
  Note: the parameter precision in ellinit() controls not only the real precision of the resulting (s)ell structure, but also the precision of most subsequent computations with this elliptic curve. You should therefore set it from the start to the value you require.

OUTPUT:

- gen - either a Pari ell structure with 19 components (if flag=0), or a Pari sell structure with 13 components (if flag=1)

EXAMPLES: An elliptic curve with integer coefficients:

```
sage: e = pari([0,1,0,1,0]).ellinit(); e
[0, 1, 0, 1, 0, 4, 2, 0, -1, -32, 224, -48, 2048/3, [0.E-28, -0.500000000000000 - 0.8660254(
[0, 1, 0, 1, 0, 4, 2, 0, -1, -32, 224, -48, 2048/3, [0.E-38, -0.500000000000000 - 0.8660254(
```

Its inexact components have the default precision of 53 bits:

```
sage: RR(e[14])
3.37150070962519
```

We can compute this to higher precision:

```
sage: R = RealField(150)
sage: e = pari([0,1,0,1,0]).ellinit(precision=150)
sage: R(e[14])
3.3715007096251920857424073155981539790016018
```

Using flag=1 returns a short elliptic curve Pari object:

```
sage: pari([0,1,0,1,0]).ellinit(flag=1)
[0, 1, 0, 1, 0, 4, 2, 0, -1, -32, 224, -48, 2048/3]
```

The coefficients can be any ring elements that convert to Pari:

```
sage: pari([0,1/2,0,-3/4,0]).ellinit(flag=1)
[0, 1/2, 0, -3/4, 0, 2, -3/2, 0, -9/16, 40, -116, 117/4, 256000/117]
sage: pari([0,0.5,0,-0.75,0]).ellinit(flag=1)
[0, 0.500000000000000, 0, -0.750000000000000, 0, 2.00000000000000, -1.50000000000000, 0, -0.
sage: pari([0,I,0,1,0]).ellinit(flag=1)
[0, I, 0, 1, 0, 4*I, 2, 0, -1, -64, 352*I, -80, 16384/5]
sage: pari([0,x,0,2*x,1]).ellinit(flag=1)
[0, x, 0, 2*x, 1, 4*x, 4*x, 4, -4*x^2 + 4*x, 16*x^2 - 96*x, -64*x^3 + 576*x^2 - 864, 64*x^4
```

ellisoncurve()

e.ellisoncurve(x): return True if the point x is on the elliptic curve e, False otherwise.

If the point or the curve have inexact coefficients, an attempt is made to take this into account.

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit()
sage: e.ellisoncurve([1,0])
True
sage: e.ellisoncurve([1,1])
False
sage: e.ellisoncurve([1,0.00000000000000001])
False
sage: e.ellisoncurve([1,0.000000000000000001])
True
sage: e.ellisoncurve([0])
True
```

ellj()

elllocalred()

e.elllocalred(p): computes the data of local reduction at the prime p on the elliptic curve e

For more details on local reduction and Kodaira types, see IV.8 and IV.9 in J. Silverman's book "Advanced topics in the arithmetic of elliptic curves".

INPUT:

- `e` - elliptic curve with coefficients in **Z**
- `p` - prime number

OUTPUT:

- `gen` - the exponent of p in the arithmetic conductor of e
- `gen` - the Kodaira type of e at p, encoded as an integer:
- `1` - type I_0: good reduction, nonsingular curve of genus 1
- `2` - type II: rational curve with a cusp
- `3` - type III: two nonsingular rational cuves intersecting tangentially at one point
- `4` - type IV: three nonsingular rational curves intersecting at one point
- `5` - type I_1: rational curve with a node
- `6 or larger` - think of it as $4 + v$, then it is type I_v: v nonsingular rational curves arranged as a v-gon
- `-1` - type I_0^*: nonsingular rational curve of multiplicity two with four nonsingular rational curves of multiplicity one attached
- `-2` - type II^*: nine nonsingular rational curves in a special configuration
- `-3` - type III^*: eight nonsingular rational curves in a special configuration
- `-4` - type IV^*: seven nonsingular rational curves in a special configuration
- `-5 or smaller` - think of it as $-4 - v$, then it is type I_v^*: chain of $v + 1$ nonsingular rational curves of multiplicity two, with two nonsingular rational curves of multiplicity one attached at either end
- `gen` - a vector with 4 components, giving the coordinate changes done during the local reduction; if the first component is 1, then the equation for e was already minimal at p
- `gen` - the local Tamagawa number c_p

EXAMPLES:

Type I_0:

```
sage: e = pari([0,0,0,0,1]).ellinit()
sage: e.elllocalred(7)
[0, 1, [1, 0, 0, 0], 1]
```

Type II:

```
sage: e = pari(EllipticCurve('27a3').a_invariants()).ellinit()
sage: e.elllocalred(3)
[3, 2, [1, -1, 0, 1], 1]
```

Type III:

```
sage: e = pari(EllipticCurve('24a4').a_invariants()).ellinit()
sage: e.elllocalred(2)
[3, 3, [1, 1, 0, 1], 2]
```

Type IV:

```
sage: e = pari(EllipticCurve('20a2').a_invariants()).ellinit()
sage: e.elllocalred(2)
[2, 4, [1, 1, 0, 1], 3]
```

Type I_1:

```
sage: e = pari(EllipticCurve('11a2').a_invariants()).ellinit()
sage: e.elllocalred(11)
[1, 5, [1, 0, 0, 0], 1]
```

Type I_2:

```
sage: e = pari(EllipticCurve('14a4').a_invariants()).ellinit()
sage: e.elllocalred(2)
[1, 6, [1, 0, 0, 0], 2]
```

Type I_6:

```
sage: e = pari(EllipticCurve('14a1').a_invariants()).ellinit()
sage: e.elllocalred(2)
[1, 10, [1, 0, 0, 0], 2]
```

Type I_0^*:

```
sage: e = pari(EllipticCurve('32a3').a_invariants()).ellinit()
sage: e.elllocalred(2)
[5, -1, [1, 1, 1, 0], 1]
```

Type II^*:

```
sage: e = pari(EllipticCurve('24a5').a_invariants()).ellinit()
sage: e.elllocalred(2)
[3, -2, [1, 2, 1, 4], 1]
```

Type III^*:

```
sage: e = pari(EllipticCurve('24a2').a_invariants()).ellinit()
sage: e.elllocalred(2)
[3, -3, [1, 2, 1, 4], 2]
```

Type IV^*:

```
sage: e = pari(EllipticCurve('20a1').a_invariants()).ellinit()
sage: e.elllocalred(2)
[2, -4, [1, 0, 1, 2], 3]
```

Type I_1^*:

```
sage: e = pari(EllipticCurve('24a1').a_invariants()).ellinit()
sage: e.elllocalred(2)
[3, -5, [1, 0, 1, 2], 4]
```

Type I_6^*:

```
sage: e = pari(EllipticCurve('90c2').a_invariants()).ellinit()
sage: e.elllocalred(3)
[2, -10, [1, 96, 1, 316], 4]
```

elllseries()

e.elllseries(s, A=1): return the value of the L-series of the elliptic curve e at the complex number s.

This uses an $O(N^{1/2})$ algorithm in the conductor N of e, so it is impractical for large conductors (say greater than 10^{12}).

INPUT:

- e - elliptic curve defined over $\mathbf{Q}$
- s - complex number
- A (optional) - cutoff point for the integral, which must be chosen close to 1 for best speed.

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit()
sage: e.elllseries(2.1)
0.402838047956645
sage: e.elllseries(1)    # random, close to 0
1.822829333527862 E-19
sage: e.elllseries(-2)
0
```

The following example differs for the last digit on 32 vs. 64 bit systems

```
sage: e.elllseries(2.1, A=1.1)
0.402838047956645
```

ellminimalmodel()

ellminimalmodel(e): return the standard minimal integral model of the rational elliptic curve e and the corresponding change of variables. INPUT:

- e - gen (that defines an elliptic curve)

OUTPUT:

- gen - minimal model
- gen - change of coordinates

EXAMPLES:

```
sage: e = pari([1,2,3,4,5]).ellinit()
sage: F, ch = e.ellminimalmodel()
sage: F[:5]
[1, -1, 0, 4, 3]
sage: ch
[1, -1, 0, -1]
sage: e.ellchangecurve(ch)[:5]
[1, -1, 0, 4, 3]
```

ellorder()

e.ellorder(x): return the order of the point x on the elliptic curve e (return 0 if x is not a torsion point)

INPUT:

- e - elliptic curve defined over $\mathbf{Q}$
- x - point on e

EXAMPLES:

```
sage: e = pari(EllipticCurve('65a1').a_invariants()).ellinit()
```

A point of order two:

```
sage: e.ellorder([0,0])
2
```

And a point of infinite order:

```
sage: e.ellorder([1,0])
0
```

ellordinate()

e.ellordinate(x): return the y-coordinates of the points on the elliptic curve e having x as x-coordinate.

INPUT:

- e - elliptic curve
- x - x-coordinate (can be a complex or p-adic number, or a more complicated object like a power series)

EXAMPLES:

```
sage: e = pari([0,1,1,-2,0]).ellinit()
sage: e.ellordinate(0)
[0, -1]
sage: C.<i> = ComplexField()
sage: e.ellordinate(i)
[0.582203589721741 - 1.38606082464177*I, -1.58220358972174 + 1.38606082464177*I]
sage: e.ellordinate(1+3*5^1+O(5^3))
[4*5 + 5^2 + O(5^3), 4 + 3*5^2 + O(5^3)]
sage: e.ellordinate('z+2*z^2+O(z^4)')
[-2*z - 7*z^2 - 23*z^3 + O(z^4), -1 + 2*z + 7*z^2 + 23*z^3 + O(z^4)]
```

ellpointtoz()

e.ellpointtoz(P): return the complex number (in the fundamental parallelogram) corresponding to the point P on the elliptic curve e, under the complex uniformization of e given by the Weierstrass p-function.

The complex number z returned by this function lies in the parallelogram formed by the real and complex periods of e, as given by e.omega().

EXAMPLES:

```
sage: e = pari([0,0,0,1,0]).ellinit()
sage: e.ellpointtoz([0,0])
1.85407467730137
```

The point at infinity is sent to the complex number 0:

```
sage: e.ellpointtoz([0])
0
```

ellpow()

e.ellpow(z, n): return n times the point z on the elliptic curve e.

INPUT:

- e - elliptic curve
- z - point on e
- n - integer, or a complex quadratic integer of complex multiplication for e (CM case is currently broken in pari)

EXAMPLES: We consider a CM curve:

```
sage: e = pari([0,0,0,1,0]).ellinit()
```

Multiplication by two:

```
sage: e.ellpow([0,0], 2)
[0]
```

Complex multiplication (this is broken at the moment):

```
sage: e.ellpow([0,0], I+1) # optional
```

ellrootno()

e.ellrootno(p): return the (local or global) root number of the L-series of the elliptic curve e

If p is a prime number, the local root number at p is returned. If p is 1, the global root number is returned. Note that the global root number is the sign of the functional equation of the L-series, and therefore conjecturally equal to the parity of the rank of e.

INPUT:

- e - elliptic curve over $\mathbf{Q}$
- p (default = 1) - 1 or a prime number

OUTPUT: 1 or -1

EXAMPLES: Here is a curve of rank 3:

```
sage: e = pari([0,0,0,-82,0]).ellinit()
sage: e.ellrootno()
-1
sage: e.ellrootno(2)
1
sage: e.ellrootno(1009)
1
```

ellsigma()

e.ellsigma(z, flag=0): return the value at the complex point z of the Weierstrass σ function associated to the elliptic curve e.

EXAMPLES:

```
sage: e = pari([0,0,0,1,0]).ellinit()
sage: C.<i> = ComplexField()
sage: e.ellsigma(2+i)
1.43490215804166 + 1.80307856719256*I
```

ellsub()

e.ellsub(z0, z1): return z0-z1 on this elliptic curve.

INPUT:

- e - elliptic curve E
- z0 - point on E
- z1 - point on E

OUTPUT: point on E

EXAMPLES:

```
sage: e = pari([0, 1, 1, -2, 0]).ellinit()
sage: e.ellsub([1,0], [-1,1])
[0, 0]
```

elltaniyama()

elltors()

e.elltors(flag = 0): return information about the torsion subgroup of the elliptic curve e

INPUT:

- e - elliptic curve over $\mathbf{Q}$
- flag (optional) - specify which algorithm to use:
- 0 (default) - use Doud's algorithm: bound torsion by computing the cardinality of e(GF(p)) for small primes of good reduction, then look for torsion points using Weierstrass parametrization and Mazur's classification
- 1 - use algorithm given by the Nagell-Lutz theorem (this is much slower)

OUTPUT:

- gen - the order of the torsion subgroup, a.k.a. the number of points of finite order
- gen - vector giving the structure of the torsion subgroup as a product of cyclic groups, sorted in non-increasing order
- gen - vector giving points on e generating these cyclic groups

EXAMPLES:

```
sage: e = pari([1,0,1,-19,26]).ellinit()
sage: e.elltors()
[12, [6, 2], [[-2, 8], [3, -2]]]
```

ellwp()

ellwp(E, z,flag=0): Return the complex value of the Weierstrass P-function at z on the lattice defined by e.

INPUT:

- E - list OR elliptic curve
- list - [om1, om2], which are Z-generators for a lattice
- elliptic curve - created using ellinit
- z - (optional) complex number OR string (default = "z")
- complex number - any number in the complex plane
- string (or PARI variable) - name of a variable.
- n - int (optional: default 20) if z is a variable, compute up to at least $o(z^n)$.
- flag - int: 0 (default): compute only P(z) 1 compute [P(z),P'(z)] 2 consider om or E as an elliptic curve and use P-function to compute the point on E (with the Weierstrass equation for E) P(z) for that curve (identical to ellztopoint in this case).

OUTPUT:

- gen - complex number or list of two complex numbers

EXAMPLES:

We first define the elliptic curve X_0(11):

```
sage: E = pari([0,-1,1,-10,-20]).ellinit()
```

Compute P(1).

```
sage: E.ellwp(1)
13.9658695257485 + 1.140149682... E-18*I
```

Compute P(1+i), where i = sqrt(-1).

```
sage: C.<i> = ComplexField()
sage: E.ellwp(pari(1+i))
-1.11510682565555 + 2.33419052307470*I
sage: E.ellwp(1+i)
-1.11510682565555 + 2.33419052307470*I
```

The series expansion, to the default 20 precision:

```
sage: E.ellwp()
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + 77531/41580*z^8 + 1202285717/928746000*z^10
```

Compute the series for wp to lower precision:

```
sage: E.ellwp(n=4)
z^-2 + 31/15*z^2 + O(z^4)
```

Next we use the version where the input is generators for a lattice:

```
sage: pari([1.2692, 0.63 + 1.45*i]).ellwp(1)
13.9656146936689 + 0.000644829272810537*I
```

With flag 1 compute the pair P(z) and P'(z):

```
sage: E.ellwp(1, flag=1)
[13.9658695257485 + 1.140149682 E-18*I, 50.5619300880073 + 1.040834085 E-17*I] # 32-bit
[13.9658695257485 + 1.14014968292839 E-18*I, 50.5619300880073 + 6.93889390390723 E-18*I] # 6
```

With flag=2, the computed pair is (x,y) on the curve instead of [P(z),P'(z)]:

```
sage: E.ellwp(1, flag=2)
[14.2992028590818 + 1.140149682 E-18*I, 50.0619300880073 + 1.040834085 E-17*I] # 32-bit
[14.2992028590818 + 1.14014968292839 E-18*I, 50.0619300880073 + 6.93889390390723 E-18*I] # 6
```

ellzeta()

e.ellzeta(z): return the value at the complex point z of the Weierstrass ζ function associated with the elliptic curve e.

Note: This function has infinitely many poles (one of which is at z=0); attempting to evaluate it too close to one of the poles will result in a PariError.

INPUT:

- e - elliptic curve
- z - complex number

EXAMPLES:

```
sage: e = pari([0,0,0,1,0]).ellinit()
sage: e.ellzeta(1)
1.06479841295883 + 0.E-19*I                 # 32-bit
1.06479841295883 - 5.42101086242752 E-20*I # 64-bit
sage: C.<i> = ComplexField()
sage: e.ellzeta(i-1)
-0.350122658523049 - 0.350122658523049*I
```

ellztopoint()

e.ellztopoint(z): return the point on the elliptic curve e corresponding to the complex number z, under the usual complex uniformization of e by the Weierstrass p-function.

INPUT:

- e - elliptic curve
- z - complex number

OUTPUT point on e

EXAMPLES:

```
sage: e = pari([0,0,0,1,0]).ellinit()
sage: C.<i> = ComplexField()
sage: e.ellztopoint(1+i)
[0.E-19 - 1.02152286795670*I, -0.149072813701096 - 0.149072813701096*I] # 32-bit
[8.67655312026478 E-20 - 1.02152286795670*I, -0.149072813701096 - 0.149072813701096*I] # 64-
```

Complex numbers belonging to the period lattice of e are of course sent to the point at infinity on e:

```
sage: e.ellztopoint(0)
[0]
```

erfc()

Return the complementary error function:

$$(2/\sqrt{\pi})\int_x^\infty e^{-t^2}dt.$$

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).erfc()
0.157299207050285
```

eta()

x.eta(flag=0): if flag=0, η function without the $q^{1/24}$; otherwise η of the complex number x in the upper half plane intelligently computed using $\mathrm{SL}(2,\mathbf{Z})$ transformations.

DETAILS: This functions computes the following. If the input x is a complex number with positive imaginary part, the result is $\prod_{n=1}^{\infty}(q-1^n)$, where $q=e^{2i\pi x}$. If x is a power series (or can be converted to a power series) with positive valuation, the result is $\prod_{n=1}^{\infty}(1-x^n)$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(i).eta()
0.998129069925959 + 0.E-21*I
```

eval()

exp()

x.exp(): exponential of x.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0).exp()
1.00000000000000
sage: pari(1).exp()
2.71828182845905
sage: pari('x+O(x^8)').exp()
1 + x + 1/2*x^2 + 1/6*x^3 + 1/24*x^4 + 1/120*x^5 + 1/720*x^6 + 1/5040*x^7 + O(x^8)
```

factor()

Return the factorization of x.

INPUT:

- `limit` - (default: -1) is optional and can be set whenever x is of (possibly recursive) rational type. If limit is set return partial factorization, using primes up to limit (up to primelimit if limit=0).

proof - (default: True) optional. If False (not the default), returned factors $< 10^{15}$ may only be pseudo-primes.

Note: In the standard PARI/GP interpreter and C-library the factor command *always* has proof=False, so beware!

EXAMPLES:

```
sage: pari('x^10-1').factor()
[x - 1, 1; x + 1, 1; x^4 - x^3 + x^2 - x + 1, 1; x^4 + x^3 + x^2 + x + 1, 1]
sage: pari(2^100-1).factor()
[3, 1; 5, 3; 11, 1; 31, 1; 41, 1; 101, 1; 251, 1; 601, 1; 1801, 1; 4051, 1; 8101, 1; 268501,
sage: pari(2^100-1).factor(proof=False)
[3, 1; 5, 3; 11, 1; 31, 1; 41, 1; 101, 1; 251, 1; 601, 1; 1801, 1; 4051, 1; 8101, 1; 268501,
```

We illustrate setting a limit:

```
sage: pari(next_prime(10^50)*next_prime(10^60)*next_prime(10^4)).factor(10^5)
[10007, 1; 10000000000000000000000000000000000000000000000015100000000070000000000000000000
```

PARI doesn't have an algorithm for factoring multivariate polynomials:

```
sage: pari('x^3 - y^3').factor()
...
PariError: sorry, (15)
```

factormod()

x.factormod(p,flag=0): factorization mod p of the polynomial x using Berlekamp. flag is optional, and can be 0: default or 1: simple factormod, same except that only the degrees of the irreducible factors are given.

factorpadic()

self.factorpadic(p,r=20,flag=0): p-adic factorization of the polynomial x to precision r. flag is optional and may be set to 0 (use round 4) or 1 (use Buchmann-Lenstra)

fibonacci()

Return the Fibonacci number of index x.

EXAMPLES:

```
sage: pari(18).fibonacci()
2584
sage: [pari(n).fibonacci() for n in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

floor()

For real x: return the largest integer = x. For rational functions: the quotient of numerator by denominator. For lists: apply componentwise.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari(5/9).floor()
0
sage: pari(11/9).floor()
1
sage: pari(1.17).floor()
1
sage: pari([1.5,2.3,4.99]).floor()
[1, 2, 4]
sage: pari([[1.1,2.2],[3.3,4.4]]).floor()
[[1, 2], [3, 4]]
sage: pari(x).floor()
x
sage: pari((x^2+x+1)/x).floor()
x + 1
sage: pari(x^2+5*x+2.5).floor()
x^2 + 5*x + 2.50000000000000

sage: pari('"hello world"').floor()
...
PariError: incorrect type (20)
```

frac()

frac(x): Return the fractional part of x, which is x - floor(x).

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari(1.75).frac()
0.750000000000000
sage: pari(sqrt(2)).frac()
0.414213562373095
sage: pari('sqrt(-2)').frac()
...
PariError: incorrect type (20)
```

galoisapply()

galoisconj()

galoisfixedfield()

galoisinit()

galoisinit(K{,den}): calculate Galois group of number field K; see PARI manual for meaning of den

galoispermtopol()

gamma()

s.gamma(precision): Gamma function at s.

If s is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If s is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).gamma()
1.00000000000000
sage: pari(5).gamma()
24.0000000000000
sage: C.<i> = ComplexField()
sage: pari(1+i).gamma()
0.498015668118356 - 0.154949828301811*I
```

gammah()

s.gammah(): Gamma function evaluated at the argument x+1/2.

If s is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If s is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).gammah()
1.32934038817914
sage: pari(5).gammah()
52.3427777845535
sage: C.<i> = ComplexField()
sage: pari(1+i).gammah()
0.575315188063452 + 0.0882106775440939*I
```

gcd()

gcd(x,y,flag=0): greatest common divisor of x and y. flag is optional, and can be 0: default, 1: use the modular gcd algorithm (x and y must be polynomials), 2 use the subresultant algorithm (x and y must be polynomials)

getattr()

hilbert()

hyperu()

a.hyperu(b,x): U-confluent hypergeometric function.

If a, b, or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).hyperu(2,3)
0.333333333333333
```

idealadd()

idealappr()

idealcoprime()

Given two integral ideals x and y of a pari number field self, return an element a of the field (expressed in the integral basis of self) such that a*x is an integral ideal coprime to y.

EXAMPLES:

```
sage: F = NumberField(x^3-2, 'alpha')
sage: nf = F._pari_()
sage: x = pari('[1, -1, 2]~')
sage: y = pari('[1, -1, 3]~')
sage: nf.idealcoprime(x, y)
[1, 0, 0]~

sage: y = pari('[2, -2, 4]~')
sage: nf.idealcoprime(x, y)
[5/43, 9/43, -1/43]~
```

idealdiv()

idealfactor()

idealhnf()

ideallog()

Return the discrete logarithm of the unit x in (ring of integers)/bid.

INPUT:

- `self` - a pari number field
- `bid` - a big ideal structure (corresponding to an ideal I of self) output by idealstar
- `x` - an element of self with valuation zero at all primes dividing I

OUTPUT:

- the discrete logarithm of x on the generators given in bid[2]

EXAMPLE:

```
sage: F = NumberField(x^3-2, 'alpha')
sage: nf = F._pari_()
sage: I = pari('[1, -1, 2]~')
sage: bid = nf.idealstar(I)
sage: x = pari('5')
sage: nf.ideallog(x, bid)
[25]~
```

idealmul()

idealnorm()

idealred()

idealstar()

Return the big ideal (bid) structure of modulus I.

INPUT:

- `self` - a pari number field
- `I` – an ideal of self, or a row vector whose first component is an ideal and whose second component is a row vector of r_1 0 or 1.
- `flag` - determines the amount of computation and the shape of the output:
  - `1` (default): return a bid structure without generators
  - `2`: return a bid structure with generators (slower)
  - `0` (deprecated): only outputs units of (ring of integers/I) as an abelian group, i.e as a 3-component vector [h,d,g]: h is the order, d is the vector of SNF cyclic components and g the corresponding generators. This flag is deprecated: it is in fact slightly faster to compute a true bid structure, which contains much more information.

EXAMPLE:

```
sage: F = NumberField(x^3-2, 'alpha')
sage: nf = F._pari_()
sage: I = pari('[1, -1, 2]~')
sage: nf.idealstar(I)
[[[43, 9, 5; 0, 1, 0; 0, 0, 1], [0]], [42, [42]], Mat([[43, [9, 1, 0]~, 1, 1, [-5, -9, 1]~],
```

idealtwoelt()

idealval()

imag()

imag(x): Return the imaginary part of x. This function also works component-wise.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari('1+2*I').imag()
2
sage: pari(sqrt(-2)).imag()
1.41421356237310
sage: pari('x+I').imag()
1
sage: pari('x+2*I').imag()
2
sage: pari('(1+I)*x^2+2*I').imag()
x^2 + 2
sage: pari('[1,2,3] + [4*I,5,6]').imag()
[4, 0, 0]
```

incgam()

s.incgam(x, y, precision): incomplete gamma function. y is optional and is the precomputed value of gamma(s).

If s is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If s is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(1+i).incgam(3-i)
-0.0458297859919946 + 0.0433696818726677*I
```

incgamc()

s.incgamc(x): complementary incomplete gamma function.

The arguments x and s are complex numbers such that s is not a pole of Γ and $|x|/(|s|+1)$ is not much larger than 1 (otherwise, the convergence is very slow). The function returns the value of the integral $\int_0^x e^{-t}t^{s-1}dt$.

If s or x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).incgamc(2)
0.864664716763387
```

int_unsafe()

Returns int form of self, but raises an exception if int does not fit into a long integer.

This is about 5 times faster than the usual int conversion.

intformal()
x.intformal(y): formal integration of x with respect to the main variable of y, or to the main variable of x if y is omitted

intvec_unsafe()
Returns Python int list form of entries of self, but raises an exception if int does not fit into a long integer. Here self must be a vector.

EXAMPLES:

```
sage: pari('[3,4,5]').type()
't_VEC'
sage: pari('[3,4,5]').intvec_unsafe()
[3, 4, 5]
sage: type(pari('[3,4,5]').intvec_unsafe()[0])
<type 'int'>
```

TESTS:

```
sage: pari(3).intvec_unsafe()
...
TypeError: gen must be of PARI type t_VEC
sage: pari('[2^150,1]').intvec_unsafe()
...
PariError: impossible assignment I-->S (23)
```

ispower()
Determine whether or not self is a perfect k-th power. If k is not specified, find the largest k so that self is a k-th power.

INPUT:

- k - int (optional)

OUTPUT:

- power - int, what power it is
- g - what it is a power of

EXAMPLES:

```
sage: pari(9).ispower()
(2, 3)
sage: pari(17).ispower()
(1, 17)
sage: pari(17).ispower(2)
(False, None)
sage: pari(17).ispower(1)
(1, 17)
sage: pari(2).ispower()
(1, 2)
```

isprime()
isprime(x, flag=0): Returns True if x is a PROVEN prime number, and False otherwise.

INPUT:

- flag - int 0 (default): use a combination of algorithms. 1: certify primality using the Pocklington-Lehmer Test. 2: certify primality using the APRCL test.

OUTPUT:

- bool - True or False

EXAMPLES:

```
sage: pari(9).isprime()
False
sage: pari(17).isprime()
True
sage: n = pari(561)     # smallest Carmichael number
sage: n.isprime()       # not just a pseudo-primality test!
False
sage: n.isprime(1)
False
sage: n.isprime(2)
False
```

ispseudoprime()

ispseudoprime(x, flag=0): Returns True if x is a pseudo-prime number, and False otherwise.

INPUT:

- `flag` - int 0 (default): checks whether x is a Baillie-Pomerance-Selfridge-Wagstaff pseudo prime (strong Rabin-Miller pseudo prime for base 2, followed by strong Lucas test for the sequence (P,-1), P smallest positive integer such that $P^2 - 4$ is not a square mod x). 0: checks whether x is a strong Miller-Rabin pseudo prime for flag randomly chosen bases (with end-matching to catch square roots of -1).

OUTPUT:

- `bool` - True or False

EXAMPLES:

```
sage: pari(9).ispseudoprime()
False
sage: pari(17).ispseudoprime()
True
sage: n = pari(561)      # smallest Carmichael number
sage: n.ispseudoprime() # not just any old pseudo-primality test!
False
sage: n.ispseudoprime(2)
False
```

issquare()

issquare(x,n): true(1) if x is a square, false(0) if not. If find_root is given, also returns the exact square root if it was computed.

issquarefree()

EXAMPLES:

```
sage: pari(10).issquarefree()
True
sage: pari(20).issquarefree()
False
```

j()

e.j(): return the j-invariant of the elliptic curve e.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.j()
-122023936/161051
sage: _.factor()
[-1, 1; 2, 12; 11, -5; 31, 3]
```

kronecker()

lcm()
Return the least common multiple of x and y. EXAMPLES:

```
sage: pari(10).lcm(15)
30
```

length()

lex()
lex(x,y): Compare x and y lexicographically (1 if xy, 0 if x==y, -1 if xy)

lift()
lift(x,v): Returns the lift of an element of Z/nZ to Z or R[x]/(P) to R[x] for a type R if v is omitted. If v is given, lift only polymods with main variable v. If v does not occur in x, lift only intmods.

INPUT:

- x - gen
- v - (optional) variable

OUTPUT: gen

EXAMPLES:

```
sage: x = pari("x")
sage: a = x.Mod('x^3 + 17*x + 3')
sage: a
Mod(x, x^3 + 17*x + 3)
sage: b = a^4; b
Mod(-17*x^2 - 3*x, x^3 + 17*x + 3)
sage: b.lift()
-17*x^2 - 3*x
```

??? more examples

lindep()

list()

list_str()
Return str that might correctly evaluate to a Python-list.

listinsert()

listput()

lllgram()

lllgramint()

lngamma()
x.lngamma(): logarithm of the gamma function of x.

This function returns the principal branch of the logarithm of the gamma function of x. The function $\log(\Gamma(x))$ is analytic on the complex plane with non-positive integers removed. This function can have much larger inputs than Γ itself.

The p-adic analogue of this function is unfortunately not implemented.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(100).lngamma()
359.134205369575
```

log()
x.log(): natural logarithm of x.

This function returns the principal branch of the natural logarithm of x, i.e., the branch such that $\Im(\log(x)) \in]-\pi, \pi]$. The result is complex (with imaginary part equal to π) if $x \in \mathbf{R}$ and $x < 0$. In general, the algorithm uses the formula

$$\log(x) \simeq \frac{\pi}{2\mathrm{agm}(1, 4/s)} - m\log(2),$$

if $s = x2^m$ is large enough. (The result is exact to B bits provided that $s > 2^{B/2}$.) At low accuracies, this function computes $\log$ using the series expansion near 1.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

Note that p-adic arguments can also be given as input, with the convention that $\log(p) = 0$. Hence, in particular, $\exp(\log(x))/x$ is not in general equal to 1 but instead to a $(p-1)$-st root of unity (or ± 1 if $p = 2$) times a power of p.

EXAMPLES:

```
sage: pari(5).log()
1.60943791243410
sage: C.<i> = ComplexField()
sage: pari(i).log()
0.E-19 + 1.57079632679490*I
```

matadjoint()

matadjoint(x): adjoint matrix of x.

EXAMPLES:

```
sage: pari('[1,2,3; 4,5,6;  7,8,9]').matadjoint()
[-3, 6, -3; 6, -12, 6; -3, 6, -3]
sage: pari('[a,b,c; d,e,f; g,h,i]').matadjoint()
[(i*e - h*f), (-i*b + h*c), (f*b - e*c); (-i*d + g*f), i*a - g*c, -f*a + d*c; (h*d - g*e), -
```

matdet()

Return the determinant of this matrix.

INPUT:

- `flag` - (optional) flag 0: using Gauss-Bareiss. 1: use classical gaussian elimination (slightly better for integer entries)

EXAMPLES:

```
sage: pari('[1,2; 3,4]').matdet(0)
-2
sage: pari('[1,2; 3,4]').matdet(1)
-2
```

matfrobenius()

M.matfrobenius(flag=0): Return the Frobenius form of the square matrix M. If flag is 1, return only the elementary divisors (a list of polynomials). If flag is 2, return a two-components vector [F,B] where F is the Frobenius form and B is the basis change so that $M = B^{-1}FB$.

EXAMPLES:

```
sage: a = pari('[1,2;3,4]')
sage: a.matfrobenius()
[0, 2; 1, 5]
sage: a.matfrobenius(flag=1)
[x^2 - 5*x - 2]
sage: a.matfrobenius(2)
[[0, 2; 1, 5], [1, -1/3; 0, 1/3]]
```

```
sage: v = a.matfrobenius(2)
sage: v[0]
[0, 2; 1, 5]
sage: v[1]^(-1)*v[0]*v[1]
[1, 2; 3, 4]
```

We let t be the matrix of T_2 acting on modular symbols of level 43, which was computed using `ModularSymbols(43,sign=1).T(2).matrix()`:

```
sage: t = pari('[3, -2, 0, 0; 0, -2, 0, 1; 0, -1, -2, 2; 0, -2, 0, 2]')
sage: t.matfrobenius()
[0, 0, 0, -12; 1, 0, 0, -2; 0, 1, 0, 8; 0, 0, 1, 1]
sage: t.charpoly('x')
x^4 - x^3 - 8*x^2 + 2*x + 12
sage: t.matfrobenius(1)
[x^4 - x^3 - 8*x^2 + 2*x + 12]
```

AUTHORS:

- Martin Albrect (2006-04-02)

mathnf()

A.mathnf(flag=0): (upper triangular) Hermite normal form of A, basis for the lattice formed by the columns of A.

INPUT:

- `flag` - optional, value range from 0 to 4 (0 if omitted), meaning : 0: naive algorithm
- `1: Use Batut's algorithm` - output 2-component vector [H,U] such that H is the HNF of A, and U is a unimodular matrix such that xU=H. 3: Use Batut's algorithm. Output [H,U,P] where P is a permutation matrix such that P A U = H. 4: As 1, using a heuristic variant of LLL reduction along the way.

EXAMPLES:

```
sage: pari('[1,2,3; 4,5,6;  7,8,9]').mathnf()
[6, 1; 3, 1; 0, 1]
```

mathnfmod()

Returns the Hermite normal form if d is a multiple of the determinant

Beware that PARI's concept of a hermite normal form is an upper triangular matrix with the same column space as the input matrix.

INPUT:

- `d` - multiple of the determinant of self

EXAMPLES:

```
       sage: M=matrix([[1,2,3],[4,5,6],[7,8,11]])
sage: d=M.det()
sage: pari(M).mathnfmod(d)
       [6, 4, 3; 0, 1, 0; 0, 0, 1]
```

Note that d really needs to be a multiple of the discriminant, not just of the exponent of the cokernel:

```
       sage: M=matrix([[1,0,0],[0,2,0],[0,0,6]])
sage: pari(M).mathnfmod(6)
[1, 0, 0; 0, 1, 0; 0, 0, 6]
sage: pari(M).mathnfmod(12)
[1, 0, 0; 0, 2, 0; 0, 0, 6]
```

mathnfmodid()

Returns the Hermite Normal Form of M concatenated with d*Identity

Beware that PARI's concept of a Hermite normal form is a maximal rank upper triangular matrix with the same column space as the input matrix.

INPUT:

- d - Determines

EXAMPLES:

```
       sage: M=matrix([[1,0,0],[0,2,0],[0,0,6]])
sage: pari(M).mathnfmodid(6)
       [1, 0, 0; 0, 2, 0; 0, 0, 6]
```

This routine is not completely equivalent to mathnfmod:

```
sage: pari(M).mathnfmod(6)
[1, 0, 0; 0, 1, 0; 0, 0, 6]
```

matker()

Return a basis of the kernel of this matrix.

INPUT:

- flag - optional; may be set to 0: default; non-zero: x is known to have integral entries.

EXAMPLES:

```
sage: pari('[1,2,3;4,5,6;7,8,9]').matker()
[1; -2; 1]
```

With algorithm 1, even if the matrix has integer entries the kernel need nto be saturated (which is weird):

```
sage: pari('[1,2,3;4,5,6;7,8,9]').matker(1)
[3; -6; 3]
sage: pari('matrix(3,3,i,j,i)').matker()
[-1, -1; 1, 0; 0, 1]
sage: pari('[1,2,3;4,5,6;7,8,9]*Mod(1,2)').matker()
[Mod(1, 2); Mod(0, 2); 1]
```

matkerint()

Return the integer kernel of a matrix.

This is the LLL-reduced Z-basis of the kernel of the matrix x with integral entries.

INPUT:

- flag - optional, and may be set to 0: default, uses a modified LLL, 1: uses matrixqz.

EXAMPLES:

```
sage: pari('[2,1;2,1]').matker()
[-1/2; 1]
sage: pari('[2,1;2,1]').matkerint()
[-1; 2]
```

This is worrisome (so be careful!):

```
sage: pari('[2,1;2,1]').matkerint(1)
Mat(1)
```

matsnf()

x.matsnf(flag=0): Smith normal form (i.e. elementary divisors) of the matrix x, expressed as a vector d. Binary digits of flag mean 1: returns [u,v,d] where d=u*x*v, otherwise only the diagonal d is returned, 2: allow polynomial entries, otherwise assume x is integral, 4: removes all information corresponding to entries equal to 1 in d.

EXAMPLES:

```
sage: pari('[1,2,3; 4,5,6;  7,8,9]').matsnf()
[0, 3, 1]
```

matsolve()

matsolve(B): Solve the linear system Mx=B for an invertible matrix M

matsolve(B) uses gaussian elimination to solve Mx=B, where M is invertible and B is a column vector.

The corresponding pari library routine is gauss. The gp-interface name matsolve has been given preference here.

INPUT:

- B - a column vector of the same dimension as the square matrix self

EXAMPLES:

```
sage: pari('[1,1;1,-1]').matsolve(pari('[1;0]'))
[1/2; 1/2]
```

mattranspose()

Transpose of the matrix self.

EXAMPLES:

```
sage: pari('[1,2,3; 4,5,6;  7,8,9]').mattranspose()
[1, 4, 7; 2, 5, 8; 3, 6, 9]
```

max()

max(x,y): Return the maximum of x and y.

min()

min(x,y): Return the minumum of x and y.

modreverse()

modreverse(x): reverse polymod of the polymod x, if it exists.

EXAMPLES:

moebius()

moebius(x): Moebius function of x.

ncols()

Return the number of columns of self.

EXAMPLES:

```
sage: pari('matrix(19,8)').ncols()
8
```

newtonpoly()

x.newtonpoly(p): Newton polygon of polynomial x with respect to the prime p.

EXAMPLES:

```
sage: x = pari('y^8+6*y^6-27*y^5+1/9*y^2-y+1')
sage: x.newtonpoly(3)
[1, 1, -1/3, -1/3, -1/3, -1/3, -1/3, -1/3]
```

nextprime()

nextprime(x): smallest pseudoprime = x

EXAMPLES:

```
sage: pari(1).nextprime()
2
sage: pari(2^100).nextprime()
1267650600228229401496703205653
```

nfbasis()

nfbasis_d()

nfbasis_d(x): Return a basis of the number field defined over QQ by x and its discriminant.

EXAMPLES:

```
sage: F = NumberField(x^3-2,'alpha')
sage: F._pari_()[0].nfbasis_d()
([1, x, x^2], -108)

sage: G = NumberField(x^5-11,'beta')
sage: G._pari_()[0].nfbasis_d()
([1, x, x^2, x^3, x^4], 45753125)

sage: pari([-2,0,0,1]).Polrev().nfbasis_d()
([1, x, x^2], -108)
```

nfdisc()

nfdisc(x): Return the discriminant of the number field defined over QQ by x.

EXAMPLES:

```
sage: F = NumberField(x^3-2,'alpha')
sage: F._pari_()[0].nfdisc()
-108

sage: G = NumberField(x^5-11,'beta')
sage: G._pari_()[0].nfdisc()
45753125

sage: f = x^3-2
sage: f._pari_()
x^3 - 2
sage: f._pari_().nfdisc()
-108
```

nfeltreduce()

Given an ideal I in Hermite normal form and an element x of the pari number field self, finds an element r in self such that x-r belongs to the ideal and r is small.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 5)
sage: I = k.ideal(a)
sage: kp = pari(k)
sage: kp.nfeltreduce(12, I.pari_hnf())
[2, 0]~
sage: 12 - k(kp.nfeltreduce(12, I.pari_hnf())) in I
True
```

nffactor()

nfgaloisconj()

Edited from the pari documentation:

nfgaloisconj(nf): list of conjugates of a root of the polynomial x=nf.pol in the same number field.

Uses a combination of Allombert's algorithm and nfroots.

EXAMPLES:

```
sage: x = QQ['x'].0; nf = pari(x^2 + 2).nfinit()
sage: nf.nfgaloisconj()
[-x, x]~
sage: nf = pari(x^3 + 2).nfinit()
sage: nf.nfgaloisconj()
```

```
[x]~
sage: nf = pari(x^4 + 2).nfinit()
sage: nf.nfgaloisconj()
[-x, x]~
```

nfgenerator()

nfinit()

nfisisom()

nfisisom(x, y): Determine if the number fields defined by x and y are isomorphic. According to the PARI documentation, this is much faster if at least one of x or y is a number field. If they are isomorphic, it returns an embedding for the generators. If not, returns 0.

EXAMPLES:

```
sage: F = NumberField(x^3-2,'alpha')
sage: G = NumberField(x^3-2,'beta')
sage: F._pari_().nfisisom(G._pari_())
[x]

sage: GG = NumberField(x^3-4,'gamma')
sage: F._pari_().nfisisom(GG._pari_())
[1/2*x^2]

sage: F._pari_().nfisisom(GG.pari_nf())
[1/2*x^2]

sage: F.pari_nf().nfisisom(GG._pari_()[0])
[x^2]

sage: H = NumberField(x^2-2,'alpha')
sage: F._pari_().nfisisom(H._pari_())
0
```

nfroots()

Return the roots of *poly* in the number field self without multiplicity.

EXAMPLES:

```
sage: y = QQ['yy'].0; _ = pari(y) # pari has variable ordering rules
sage: x = QQ['zz'].0; nf = pari(x^2 + 2).nfinit()
sage: nf.nfroots(y^2 + 2)
[-zz, zz]
sage: nf = pari(x^3 + 2).nfinit()
sage: nf.nfroots(y^3 + 2)
[zz]
sage: nf = pari(x^4 + 2).nfinit()
sage: nf.nfroots(y^4 + 2)
[-zz, zz]
```

nfrootsof1()

nf.nfrootsof1()

number of roots of unity and primitive root of unity in the number field nf.

EXAMPLES:

```
sage: nf = pari('x^2 + 1').nfinit()
sage: nf.nfrootsof1()
[4, [0, 1]~]
```

nfsubfields()

Find all subfields of degree d of number field nf (all subfields if d is null or omitted). Result is a vector of

subfields, each being given by [g,h], where g is an absolute equation and h expresses one of the roots of g in terms of the root x of the polynomial defining nf.

INPUT:

- `self` - nf number field
- `d` - integer

norm()

nrows()

Return the number of rows of self.

EXAMPLES:

```
sage: pari('matrix(19,8)').nrows()
19
```

numbpart()

numbpart(x): returns the number of partitions of x.

EXAMPLES:

```
sage: pari(20).numbpart()
627
sage: pari(100).numbpart()
190569292
```

numdiv()

Return the number of divisors of the integer n.

EXAMPLES:

```
sage: pari(10).numdiv()
4
```

numerator()

numerator(x): Returns the numerator of x.

INPUT:

- `x` - gen

OUTPUT: gen

EXAMPLES:

numtoperm()

numtoperm(k, n): Return the permutation number k (mod n!) of n letters, where n is an integer.

INPUT:

- `k` - gen, integer
- `n` - int

OUTPUT:

- `gen` - vector (permutation of 1,...,n)

EXAMPLES:

omega()

e.omega(): return basis for the period lattice of the elliptic curve e.

EXAMPLES:

```
sage: e = pari([0, -1, 1, -10, -20]).ellinit()
sage: e.omega()
[1.26920930427955, 0.634604652139777 + 1.45881661693850*I]
```

order()

padicappr()
x.padicappr(a): p-adic roots of the polynomial x congruent to a mod p

padicprec()
padicprec(x,p): Return the absolute p-adic precision of the object x.
INPUT:

- x - gen

OUTPUT: int
EXAMPLES:

parent()

permtonum()
permtonum(x): Return the ordinal (between 1 and n!) of permutation vector x. ??? Huh ??? say more. what is a perm vector. 0 to n-1 or 1-n.
INPUT:

- x - gen (vector of integers)

OUTPUT:

- gen - integer

EXAMPLES:

phi()
Return the Euler phi function of n. EXAMPLES:

```
sage: pari(10).phi()
4
```

polcoeff()
EXAMPLES:

```
sage: f = pari("x^2 + y^3 + x*y")
sage: f
x^2 + y*x + y^3
sage: f.polcoeff(1)
y
sage: f.polcoeff(3)
0
sage: f.polcoeff(3, "y")
1
sage: f.polcoeff(1, "y")
x
```

polcompositum()

poldegree()
f.poldegree(var=x): Return the degree of this polynomial.

poldisc()
f.poldist(var=x): Return the discriminant of this polynomial.

poldiscreduced()

polgalois()
f.polgalois(): Galois group of the polynomial f

polhensellift()
self.polhensellift(y, p, e): lift the factorization y of self modulo p to a factorization modulo p^e using Hensel lift. The factors in y must be pairwise relatively prime modulo p.

polinterpolate()
self.polinterpolate(ya,x,e): polynomial interpolation at x according to data vectors self, ya (i.e. return P such that P(self[i]) = ya[i] for all i). Also return an error estimate on the returned value.

polisirreducible()
f.polisirreducible(): Returns True if f is an irreducible non-constant polynomial, or False if f is reducible or constant.

pollead()
self.pollead(v): leading coefficient of polynomial or series self, or self itself if self is a scalar. Error otherwise. With respect to the main variable of self if v is omitted, with respect to the variable v otherwise

polrecip()

polred()

polredabs()

polresultant()

polroots()
polroots(x,flag=0): complex roots of the polynomial x. flag is optional, and can be 0: default, uses Schonhage's method modified by Gourdon, or 1: uses a modified Newton method.

polrootsmod()

polrootspadic()

polrootspadicfast()

polsturm()

polsturm_full()

polsylvestermatrix()

polsym()

polylog()
x.polylog(m,flag=0): m-th polylogarithm of x. flag is optional, and can be 0: default, 1: D_m -modified m-th polylog of x, 2: D_m-modified m-th polylog of x, 3: P_m-modified m-th polylog of x.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

TODO: Add more explanation, copied from the PARI manual.

EXAMPLES:

```
sage: pari(10).polylog(3)
5.64181141475134 - 8.32820207698027*I
sage: pari(10).polylog(3,0)
5.64181141475134 - 8.32820207698027*I
sage: pari(10).polylog(3,1)
0.523778453502411
sage: pari(10).polylog(3,2)
-0.400459056163451
```

precision()
precision(x,n): Change the precision of x to be n, where n is a C-integer). If n is omitted, output the real precision of x.

INPUT:

- x - gen
- n - (optional) int

OUTPUT: nothing or gen if n is omitted

EXAMPLES:

primepi()
Return the number of primes less than or equal to self.

EXAMPLES:

```
sage: pari(7).primepi()
4
sage: pari(100).primepi()
25
sage: pari(1000).primepi()
168
sage: pari(100000).primepi()
9592
sage: pari(0).primepi()
0
sage: pari(-15).primepi()
0
sage: pari(500509).primepi()
41581
```

printtex()

psi()

x.psi(): psi-function at x.

Return the ψ-function of x, i.e., the logarithmic derivative $\Gamma'(x)/\Gamma(x)$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).psi()
-0.577215664901533
```

python()

Return Python eval of self.

Note: is self is a real (type t_REAL) the result will be a RealField element of the equivalent precision; if self is a complex (type t_COMPLEX) the result will be a ComplexField element of precision the minimum precision of the real and imaginary parts.

python_list()

Return a Python list of the PARI gens. This object must be of type t_VEC

INPUT: NoneOUTPUT:

- `list` - Python list whose elements are the elements of the input gen.

EXAMPLES:

```
sage: v=pari([1,2,3,10,102,10])
sage: w = v.python_list()
sage: w
[1, 2, 3, 10, 102, 10]
sage: type(w[0])
<type 'sage.libs.pari.gen.gen'>
sage: pari("[1,2,3]").python_list()
[1, 2, 3]
```

python_list_small()

Return a Python list of the PARI gens. This object must be of type t_VECSMALL, and the resulting list contains python 'int's

EXAMPLES:

```
sage: v=pari([1,2,3,10,102,10]).Vecsmall()
sage: w = v.python_list_small()
sage: w
[1, 2, 3, 10, 102, 10]
```

```
sage: type(w[0])
<type 'int'>
```

qfbhclassno()

Computes the Hurwitz-Kronecker class number of n.

If n is large (more than $5 * 10^5$), the result is conditional upon GRH.

EXAMPLES:

```
sage: pari(-10007).qfbhclassno()
77
sage: pari(-3).qfbhclassno()
1/3
```

qflll()

qflll(x,flag=0): LLL reduction of the vectors forming the matrix x (gives the unimodular transformation matrix). The columns of x must be linearly independent, unless specified otherwise below. flag is optional, and can be 0: default, 1: assumes x is integral, columns may be dependent, 2: assumes x is integral, returns a partially reduced basis, 4: assumes x is integral, returns [K,I] where K is the integer kernel of x and I the LLL reduced image, 5: same as 4 but x may have polynomial coefficients, 8: same as 0 but x may have polynomial coefficients.

qflllgram()

qflllgram(x,flag=0): LLL reduction of the lattice whose gram matrix is x (gives the unimodular transformation matrix). flag is optional and can be 0: default,1: lllgramint algorithm for integer matrices, 4: lllgramkerim giving the kernel and the LLL reduced image, 5: lllgramkerimgen same when the matrix has polynomial coefficients, 8: lllgramgen, same as qflllgram when the coefficients are polynomials.

qfminim()

qfminim(x,bound,maxnum,flag=0): number of vectors of square norm = bound, maximum norm and list of vectors for the integral and definite quadratic form x; minimal non-zero vectors if bound=0. flag is optional, and can be 0: default; 1: returns the first minimal vector found (ignore maxnum); 2: as 0 but uses a more robust, slower implementation, valid for non integral quadratic forms.

qfrep()

qfrep(x,B,flag=0): vector of (half) the number of vectors of norms from 1 to B for the integral and definite quadratic form x. Binary digits of flag mean 1: count vectors of even norm from 1 to 2B, 2: return a t_VECSMALL instead of a t_VEC.

random()

`random(N=2^31)`: Return a pseudo-random integer between 0 and $N - 1$.

INPUT:

-`N` - gen, integer

OUTPUT:

- `gen` - integer

EXAMPLES:

real()

real(x): Return the real part of x.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

reverse()

Return the polynomial obtained by reversing the coefficients of this polynomial.

rnfcharpoly()

rnfdisc()

rnfeltabstorel()

rnfeltreltoabs()

rnfequation()

rnfidealabstorel()

rnfidealdown()

rnfidealdown(rnf,x): finds the intersection of the ideal x with the base field.

EXAMPLES: sage: x = ZZ['xx1'].0; pari(x) xx1 sage: y = ZZ['yy1'].0; pari(y) yy1 sage: nf = pari(y^2 - 6*y + 24).nfinit() sage: rnf = nf.rnfinit(x^2 - pari(y))

This is the relative HNF of the inert ideal (2) in rnf:

sage: P = pari('[[[1, 0]~, [0, 0]~; [0, 0]~, [1, 0]~], [[2, 0; 0, 2], [2, 0; 0, 1/2]]]')

And this is the HNF of the inert ideal (2) in nf:

sage: rnf.rnfidealdown(P) [2, 0; 0, 2]

rnfidealhnf()

rnfidealnormrel()

rnfidealreltoabs()

rnfidealtwoelt()

rnfinit()

EXAMPLES: We construct a relative number field.

```
sage: f = pari('y^3+y+1')
sage: K = f.nfinit()
sage: x = pari('x'); y = pari('y')
sage: g = x^5 - x^2 + y
sage: L = K.rnfinit(g)
```

rnfisfree()

round()

round(x,estimate=False): If x is a real number, returns x rounded to the nearest integer (rounding up). If the optional argument estimate is True, also returns the binary exponent e of the difference between the original and the rounded value (the "fractional part") (this is the integer ceiling of log_2(error)).

When x is a general PARI object, this function returns the result of rounding every coefficient at every level of PARI object. Note that this is different than what the truncate function does (see the example below).

One use of round is to get exact results after a long approximate computation, when theory tells you that the coefficients must be integers.

INPUT:

- x - gen
- `estimate` - (optional) bool, False by default

OUTPUT:

- if estimate is False, return a single gen.
- if estimate is True, return rounded verison of x and error estimate in bits, both as gens.

EXAMPLES:

```
sage: pari('1.5').round()
2
sage: pari('1.5').round(True)
(2, -1)
sage: pari('1.5 + 2.1*I').round()
```

```
2 + 2*I
sage: pari('1.0001').round(True)
(1, -14)
sage: pari('(2.4*x^2 - 1.7)/x').round()
(2*x^2 - 2)/x
sage: pari('(2.4*x^2 - 1.7)/x').truncate()
2.40000000000000*x
```

serconvol()

serlaplace()

serreverse()

serreverse(f): reversion of the power series f.

If f(t) is a series in t with valuation 1, find the series g(t) such that g(f(t)) = t.

EXAMPLES:

```
sage: f = pari('x+x^2+x^3+O(x^4)'); f
x + x^2 + x^3 + O(x^4)
sage: g = f.serreverse(); g
x - x^2 + x^3 + O(x^4)
sage: f.subst('x',g)
x + O(x^4)
sage: g.subst('x',f)
x + O(x^4)
```

shift()

shift(x,n): shift x left n bits if n=0, right -n bits if n0.

shiftmul()

shiftmul(x,n): Return the product of x by 2^n.

sign()

sign(x): Return the sign of x, where x is of type integer, real or fraction.

simplify()

simplify(x): Simplify the object x as much as possible, and return the result.

A complex or quadratic number whose imaginary part is an exact 0 (i.e., not an approximate one such as O(3) or 0.E-28) is converted to its real part, and a a polynomial of degree 0 is converted to its constant term. Simplification occurs recursively.

This function is useful before using arithmetic functions, which expect integer arguments:

EXAMPLES:

```
sage: y = pari('y')
sage: x = pari('9') + y - y
sage: x
9
sage: x.type()
't_POL'
sage: x.factor()
matrix(0,2)
sage: pari('9').factor()
Mat([3, 2])
sage: x.simplify()
9
sage: x.simplify().factor()
Mat([3, 2])
sage: x = pari('1.5 + 0*I')
sage: x.type()
't_COMPLEX'
```

```
sage: x.simplify()
1.50000000000000
sage: y = x.simplify()
sage: y.type()
't_REAL'
```

sin()

x.sin(): The sine of x.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).sin()
0.841470984807897
sage: C.<i> = ComplexField()
sage: pari(1+i).sin()
1.29845758141598 + 0.634963914784736*I
```

sinh()

The hyperbolic sine function.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0).sinh()
0.E-19
sage: C.<i> = ComplexField()
sage: pari(1+i).sinh()
0.634963914784736 + 1.29845758141598*I
```

sizebyte()

sizebyte(x): Return the total number of bytes occupied by the complete tree of the object x. Note that this number depends on whether the computer is 32-bit or 64-bit (see examples).

INPUT:

- x - gen

OUTPUT: int (a Python int)

EXAMPLES:

```
sage: pari('1').sizebyte()
12            # 32-bit
24            # 64-bit
sage: pari('10').sizebyte()
12            # 32-bit
24            # 64-bit
sage: pari('10000000000000').sizebyte()
16            # 32-bit
24            # 64-bit
sage: pari('10^100').sizebyte()
52            # 32-bit
64            # 64-bit
sage: pari('x').sizebyte()
36            # 32-bit
72            # 64-bit
sage: pari('x^20').sizebyte()
264           # 32-bit
```

```
528          # 64-bit
sage: pari('[x, 10^100]').sizebyte()
100          # 32-bit
160          # 64-bit
```

sizedigit()

sizedigit(x): Return a quick estimate for the maximal number of decimal digits before the decimal point of any component of x.

INPUT:

- x - gen

OUTPUT:

- int - Python integer

EXAMPLES:

```
sage: x = pari('10^100')
sage: x.Str().length()
101
sage: x.sizedigit()
101
```

Note that digits after the decimal point are ignored.

```
sage: x = pari('1.234')
sage: x
1.23400000000000
sage: x.sizedigit()
1
```

The estimate can be one too big:

```
sage: pari('7234.1').sizedigit()
4
sage: pari('9234.1').sizedigit()
5
```

sqr()

x.sqr(): square of x. Faster than, and most of the time (but not always - see the examples) identical to x*x.

EXAMPLES:

```
sage: pari(2).sqr()
4
```

For 2-adic numbers, x.sqr() may not be identical to x*x (squaring a 2-adic number increases its precision):

```
sage: pari("1+O(2^5)").sqr()
1 + O(2^6)
sage: pari("1+O(2^5)")*pari("1+O(2^5)")
1 + O(2^5)
```

However:

```
sage: x = pari("1+O(2^5)"); x*x
1 + O(2^6)
```

sqrt()

x.sqrt(precision): The square root of x.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).sqrt()
1.41421356237310
```

sqrtn()

x.sqrtn(n): return the principal branch of the n-th root of x, i.e., the one such that $\arg(\sqrt(x)) \in]-\pi/n, \pi/n]$. Also returns a second argument which is a suitable root of unity allowing one to recover all the other roots. If it was not possible to find such a number, then this second return value is 0. If the argument is present and no square root exists, return 0 instead of raising an error.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

Note: intmods (modulo a prime) and p-adic numbers are allowed as arguments.

INPUT:

- x - gen
- n - integer

OUTPUT:

- gen - principal n-th root of x
- gen - root of unity z that gives the other roots

EXAMPLES:

```
sage: s, z = pari(2).sqrtn(5)
sage: z
0.309016994374947 + 0.951056516295154*I
sage: s
1.14869835499704
sage: s^5
2.00000000000000
sage: z^5
1.00000000000000 + 5.42101086 E-19*I         # 32-bit
1.00000000000000 + 4.87890977618477 E-19*I   # 64-bit
sage: (s*z)^5
2.00000000000000 + 1.409462824 E-18*I        # 32-bit
2.00000000000000 + 7.58941520739853 E-19*I   # 64-bit
```

subst()

EXAMPLES:

```
sage: x = pari("x"); y = pari("y")
sage: f = pari('x^3 + 17*x + 3')
sage: f.subst(x, y)
y^3 + 17*y + 3
sage: f.subst(x, "z")
z^3 + 17*z + 3
sage: f.subst(x, "z")^2
z^6 + 34*z^4 + 6*z^3 + 289*z^2 + 102*z + 9
sage: f.subst(x, "x+1")
x^3 + 3*x^2 + 20*x + 21
sage: f.subst(x, "xyz")
xyz^3 + 17*xyz + 3
sage: f.subst(x, "xyz")^2
xyz^6 + 34*xyz^4 + 6*xyz^3 + 289*xyz^2 + 102*xyz + 9
```

substpol()

sumdiv()

Return the sum of the divisors of n.

EXAMPLES:

```
sage: pari(10).sumdiv()
18
```

sumdivk()

Return the sum of the k-th powers of the divisors of n.

EXAMPLES:

```
sage: pari(10).sumdivk(2)
130
```

tan()

x.tan() - tangent of x

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(2).tan()
-2.18503986326152
sage: C.<i> = ComplexField()
sage: pari(i).tan()
0.E-19 + 0.761594155955765*I
```

tanh()

x.tanh() - hyperbolic tangent of x

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(1).tanh()
0.761594155955765
sage: C.<i> = ComplexField()
sage: pari(i).tanh()
0.E-19 + 1.55740772465490*I
```

taylor()

teichmuller()

teichmuller(x): teichmuller character of p-adic number x.

This is the unique $(p-1)$-st root of unity congruent to $x/p^{v_p(x)}$ modulo p.

EXAMPLES:

```
sage: pari('2+O(7^5)').teichmuller()
2 + 4*7 + 6*7^2 + 3*7^3 + O(7^5)
```

theta()

q.theta(z): Jacobi sine theta-function.

If q or z is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If the arguments are inexact (e.g. real), the smallest of their precisions is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0.5).theta(2)
1.63202590295260
```

thetanullk()

q.thetanullk(k): return the k-th derivative at z=0 of theta(q,z).

If q is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If q is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

EXAMPLES:

```
sage: pari(0.5).thetanullk(1)
0.548978532560341
```

thue()

thueinit()

trace()

Return the trace of this PARI object.

EXAMPLES:

```
sage: pari('[1,2; 3,4]').trace()
5
```

truncate()

truncate(x,estimate=False): Return the truncation of x. If estimate is True, also return the number of error bits.

When x is in the real numbers, this means that the part after the decimal point is chopped away, e is the binary exponent of the difference between the original and truncated value (the "fractional part"). If x is a rational function, the result is the integer part (Euclidean quotient of numerator by denominator) and if requested the error estimate is 0.

When truncate is applied to a power series (in X), it transforms it into a polynomial or a rational function with denominator a power of X, by chopping away the $O(X^k)$. Similarly, when applied to a p-adic number, it transforms it into an integer or a rational number by chopping away the $O(p^k)$.

INPUT:

- x - gen
- estimate - (optional) bool, which is False by default

OUTPUT:

- if estimate is False, return a single gen.
- if estimate is True, return rounded version of x and error estimate in bits, both as gens.

EXAMPLES:

```
sage: pari('(x^2+1)/x').round()
(x^2 + 1)/x
sage: pari('(x^2+1)/x').truncate()
x
sage: pari('1.043').truncate()
1
sage: pari('1.043').truncate(True)
(1, -5)
sage: pari('1.6').truncate()
1
sage: pari('1.6').round()
2
sage: pari('1/3 + 2 + 3^2 + O(3^3)').truncate()
34/3
sage: pari('sin(x+O(x^10))').truncate()
1/362880*x^9 - 1/5040*x^7 + 1/120*x^5 - 1/6*x^3 + x
sage: pari('sin(x+O(x^10))').round()    # each coefficient has abs < 1
x + O(x^10)
```

type()

Return the Pari type of self as a string.

Note: In Cython, it is much faster to simply use typ(self.g) for checking Pari types.

EXAMPLES:

```
sage: pari(7).type()
't_INT'
sage: pari('x').type()
't_POL'
```

valuation()

valuation(x,p): Return the valuation of x with respect to p.

The valuation is the highest exponent of p dividing x.

- If p is an integer, x must be an integer, an intmod whose modulus is divisible by p, a rational number, a p-adic number, or a polynomial or power series in which case the valuation is the minimum of the valuations of the coefficients.
- If p is a polynomial, x must be a polynomial or a rational function. If p is a monomial then x may also be a power series.
- If x is a vector, complex or quadratic number, then the valuation is the minimum of the component valuations.
- If x = 0, the result is $2^31 - 1$ on 32-bit machines or $2^63 - 1$ on 64-bit machines if x is an exact object. If x is a p-adic number or power series, the result is the exponent of the zero.

INPUT:

- x - gen
- p - coercible to gen

OUTPUT:

- gen - integer

EXAMPLES:

```
sage: pari(9).valuation(3)
2
sage: pari(9).valuation(9)
1
sage: x = pari(9).Mod(27); x.valuation(3)
2
sage: pari('5/3').valuation(3)
-1
sage: pari('9 + 3*x + 15*x^2').valuation(3)
1
sage: pari([9,3,15]).valuation(3)
1
sage: pari('9 + 3*x + 15*x^2 + O(x^5)').valuation(3)
1

sage: pari('x^2*(x+1)^3').valuation(pari('x+1'))
3
sage: pari('x + O(x^5)').valuation('x')
1
sage: pari('2*x^2 + O(x^5)').valuation('x')
2

sage: pari(0).valuation(3)
2147483647            # 32-bit
9223372036854775807   # 64-bit
```

variable()

variable(x): Return the main variable of the object x, or p if x is a p-adic number.

This function raises a TypeError exception on scalars, i.e., on objects with no variable associated to them.

INPUT:

- x - gen

OUTPUT: gen

EXAMPLES:

```
sage: pari('x^2 + x -2').variable()
x
sage: pari('1+2^3 + O(2^5)').variable()
2
sage: pari('x+y0').variable()
x
sage: pari('y0+z0').variable()
y0
```

vecextract()

self.vecextract(y,z): extraction of the components of the matrix or vector x according to y and z. If z is omitted, y designates columns, otherwise y corresponds to rows and z to columns. y and z can be vectors (of indices), strings (indicating ranges as in"1..10") or masks (integers whose binary representation indicates the indices to extract, from left to right 1, 2, 4, 8, etc.)

Note: This function uses the PARI row and column indexing, so the first row or column is indexed by 1 instead of 0.

vecmax()

vecmax(x): Return the maximum of the elements of the vector/matrix x.

vecmin()

vecmin(x): Return the maximum of the elements of the vector/matrix x.

weber()

x.weber(flag=0): One of Weber's f functions of x. flag is optional, and can be 0: default, function f(x)=exp(-i*Pi/24)*eta((x+1)/2)/eta(x) such that $j = (f^{24} - 16)^3/f^{24}$, 1: function f1(x)=eta(x/2)/eta(x) such that $j = (f1^24 + 16)^3/f2^{24}$, 2: function f2(x)=sqrt(2)*eta(2*x)/eta(x) such that $j = (f2^{24} + 16)^3/f2^{24}$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

TODO: Add further explanation from PARI manual.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: pari(i).weber()
1.18920711500272 + 0.E-19*I
sage: pari(i).weber(1)
1.09050773266526 + 0.E-19*I
sage: pari(i).weber(2)
1.09050773266526 + 0.E-19*I
```

xgcd()

Returns u,v,d such that d=gcd(x,y) and u*x+v*y=d.

EXAMPLES:

```
sage: pari(10).xgcd(15)
(5, -1, 1)
```

zeta()

zeta(s): zeta function at s with s a complex or a p-adic number.

If s is a complex number, this is the Riemann zeta function $\zeta(s) = \sum_{n\geq 1} n^{-s}$, computed either using the Euler-Maclaurin summation formula (if s is not an integer), or using Bernoulli numbers (if s is a negative integer or an even nonnegative integer), or using modular forms (if s is an odd nonnegative integer).

If s is a p-adic number, this is the Kubota-Leopoldt zeta function, i.e. the unique continuous p-adic function on the p-adic integers that interpolates the values of $(1 - p^{-k})\zeta(k)$ at negative integers k such that $k \equiv 1 \pmod{p-1}$ if p is odd, and at odd k if $p = 2$.

If x is an exact argument, it is first converted to a real or complex number using the optional parameter precision (in bits). If x is inexact (e.g. real), its own precision is used in the computation, and the parameter precision is ignored.

INPUT:

- `s` - gen (real, complex, or p-adic number)

OUTPUT:

- `gen` - value of zeta at s.

EXAMPLES:

```
sage: pari(2).zeta()
1.64493406684823
sage: x = RR(pi)^2/6
sage: pari(x)
1.64493406684823
sage: pari(3).zeta()
1.20205690315959
sage: pari('1+5*7+2*7^2+O(7^3)').zeta()
4*7^-2 + 5*7^-1 + O(7^0)
```

znprimroot()

Return a primitive root modulo self, whenever it exists.

This is a generator of the group $(\mathbf{Z}/n\mathbf{Z})^*$, whenever this group is cyclic, i.e. if $n = 4$ or $n = p^k$ or $n = 2p^k$, where p is an odd prime and k is a natural number.

INPUT:

- `self` - positive integer equal to 4, or a power of an odd prime, or twice a power of an odd prime

OUTPUT: gen

EXAMPLES:

```
sage: pari(4).znprimroot()
Mod(3, 4)
sage: pari(10007^3).znprimroot()
Mod(5, 1002101470343)
sage: pari(2*109^10).znprimroot()
Mod(236736367459211723407, 473472734918423446802)
```

init_pari_stack()

Change the PARI scratch stack space to the given size.

The main application of this command is that you've done some individual PARI computation that used a lot of stack space. As a result the PARI stack may have doubled several times and is now quite large. That object you computed is copied off to the heap, but the PARI stack is never automatically shrunk back down. If you call this function you can shrink it back down.

If you set this too small then it will automatically be increased if it is exceeded, which could make some calculations initially slower (since they have to be redone until the stack is big enough).

INPUT:

- `size` - an integer (default: 8000000)

EXAMPLES:

```
sage: get_memory_usage()                          # random output
'122M+'
sage: a = pari('2^100000000')
sage: get_memory_usage()                          # random output
'157M+'
sage: del a
sage: get_memory_usage()                          # random output
'145M+'
```

Hey, I want my memory back!

```
sage: sage.libs.pari.gen.init_pari_stack()
sage: get_memory_usage()                          # random output
'114M+'
```

Ahh, that's better.

max()
max(x,y): Return the maximum of x and y.

min()
min(x,y): Return the minimum of x and y.

pbw()
Convert from precision expressed in bits to pari real precision expressed in words. Note: this rounds up to the nearest word, adjusts for the two codewords of a pari real, and is architecture-dependent.

EXAMPLES:

```
sage: import sage.libs.pari.gen as gen
sage: gen.prec_bits_to_words(70)
5   # 32-bit
4   # 64-bit


sage: [(32*n,gen.prec_bits_to_words(32*n)) for n in range(1,9)]
[(32, 3), (64, 4), (96, 5), (128, 6), (160, 7), (192, 8), (224, 9), (256, 10)] # 32-bit
[(32, 3), (64, 3), (96, 4), (128, 4), (160, 5), (192, 5), (224, 6), (256, 6)] # 64-bit
```

prec_bits_to_dec()
Convert from precision expressed in bits to precision expressed in decimal.

EXAMPLES:

```
sage: import sage.libs.pari.gen as gen
sage: gen.prec_bits_to_dec(53)
15
sage: [(32*n,gen.prec_bits_to_dec(32*n)) for n in range(1,9)]
[(32, 9),
(64, 19),
(96, 28),
(128, 38),
(160, 48),
(192, 57),
(224, 67),
(256, 77)]
```

prec_bits_to_words()
Convert from precision expressed in bits to pari real precision expressed in words. Note: this rounds up to the nearest word, adjusts for the two codewords of a pari real, and is architecture-dependent.

EXAMPLES:

```
sage: import sage.libs.pari.gen as gen
sage: gen.prec_bits_to_words(70)
5   # 32-bit
4   # 64-bit


sage: [(32*n,gen.prec_bits_to_words(32*n)) for n in range(1,9)]
[(32, 3), (64, 4), (96, 5), (128, 6), (160, 7), (192, 8), (224, 9), (256, 10)] # 32-bit
[(32, 3), (64, 3), (96, 4), (128, 4), (160, 5), (192, 5), (224, 6), (256, 6)] # 64-bit
```

prec_dec_to_bits()

Convert from precision expressed in decimal to precision expressed in bits.

EXAMPLES:

```
sage: import sage.libs.pari.gen as gen
sage: gen.prec_dec_to_bits(15)
49
sage: [(n,gen.prec_dec_to_bits(n)) for n in range(10,100,10)]
[(10, 33),
(20, 66),
(30, 99),
(40, 132),
(50, 166),
(60, 199),
(70, 232),
(80, 265),
(90, 298)]
```

prec_dec_to_words()

Convert from precision expressed in decimal to precision expressed in words. Note: this rounds up to the nearest word, adjusts for the two codewords of a pari real, and is architecture-dependent.

EXAMPLES:

```
sage: import sage.libs.pari.gen as gen
sage: gen.prec_dec_to_words(38)
6   # 32-bit
4   # 64-bit
sage: [(n,gen.prec_dec_to_words(n)) for n in range(10,90,10)]
[(10, 4), (20, 5), (30, 6), (40, 7), (50, 8), (60, 9), (70, 10), (80, 11)] # 32-bit
[(10, 3), (20, 4), (30, 4), (40, 5), (50, 5), (60, 6), (70, 6), (80, 7)] # 64-bit
```

prec_words_to_bits()

Convert from pari real precision expressed in words to precision expressed in bits. Note: this adjusts for the two codewords of a pari real, and is architecture-dependent.

EXAMPLES:

```
sage: import sage.libs.pari.gen as gen
sage: gen.prec_words_to_bits(10)
256   # 32-bit
512   # 64-bit
sage: [(n,gen.prec_words_to_bits(n)) for n in range(3,10)]
[(3, 32), (4, 64), (5, 96), (6, 128), (7, 160), (8, 192), (9, 224)]  # 32-bit
[(3, 64), (4, 128), (5, 192), (6, 256), (7, 320), (8, 384), (9, 448)] # 64-bit
```

prec_words_to_dec()
Convert from precision expressed in words to precision expressed in decimal. Note: this adjusts for the two codewords of a pari real, and is architecture-dependent.

EXAMPLES:

```
sage: import sage.libs.pari.gen as gen
sage: gen.prec_words_to_dec(5)
28   # 32-bit
57   # 64-bit
sage: [(n,gen.prec_words_to_dec(n)) for n in range(3,10)]
[(3, 9), (4, 19), (5, 28), (6, 38), (7, 48), (8, 57), (9, 67)] # 32-bit
[(3, 19), (4, 38), (5, 57), (6, 77), (7, 96), (8, 115), (9, 134)] # 64-bit
```

vecsmall_to_intlist()
INPUT:

- v - a gen of type Vecsmall

OUTPUT: a Python list of Python ints

14.2 Victor Shoup's NTL C++ Library

Sage provides an interface to Victor Shoup's C++ library NTL. Features of this library include *incredibly fast* arithmetic with polynomials and assymptotically fast factorization of polynomials.

14.3 Cremona's mwrank C++ library

mwrank_initprimes(*filename, verb=False*)
mwrank_initprimes(filename, verb=False):

INPUT:

- filename - (string) the name of a file of primes
- verb - (bool: default False) verbose or not?

EXAMPLES:

```
sage: file = tmp_filename()
sage: open(file,'w').write(' '.join([str(p) for p in prime_range(10^6)]))
sage: mwrank_initprimes(file, verb=False)
```

CHAPTER

FIFTEEN

NETWORKING AND GRID COMPUTING

Sage includes custom highly-integrated support for coarse grain distributed (or "grid") computing. See the chapter below on `dsage`.

Sage includes the Moin Moin Wiki by default, which "is an advanced, easy to use and extensible WikiEngine with a large community of users. Said in a few words, it is about collaboration on easily editable web pages."

The Twisted, an event-driven networking framework written in Python, is a networking library that is included with Sage. It is robust, mature, fast, and offers a vast range of networking functionality.

The Sage Notebook (see Chapter *The Sage Notebook*) is another networking application included with Sage.

15.1 Wiki Interactive Web Page.

Sage includes the Moin Moin Wiki interactive web page system standard. To start your own math-typesetting-aware wiki server immediately, just type `wiki()` at the command line.

The Moin Moin Wiki "is an advanced, easy to use and extensible WikiEngine with a large community of users. Said in a few words, it is about collaboration on easily editable web pages."

wiki (*directory='sage_wiki', port=9000, address='localhost', threads=10*)
Create (if necessary) and start up a Moin Moin wiki.

The wiki will be served on the given port.

The moin package contains a modified version of moin moin, which comes with jsmath latex typesetting pre-configured; use dollar signs to typeset.

wiki_create_instance (*directory='sage_wiki'*)

wiki_simple_http (*directory='sage_wiki', port=9000, address='localhost'*)
Create (if necessary) and start up a Moin Moin wiki.

The wiki will be served on the given port.

The moin package contains a modified version of moin moin, which comes with jsmath latex typesetting pre-configured; use dollar signs to typeset.

15.2 DSage: Distributed Sage

Distributed Sage `dsage` is a distributed computing framework suitable for coarse distributed compuatations.

class DistributedSage()
Distributed Sage allows you to do distributed computing in Sage.

To get up and running quickly, run dsage.setup() to run the configuration utility.

Note that configuration files will be stored in the directory `$DOT Sage/dsage`.

QUICK-START

1. Launch sage
2. Run:

```
sage: dsage.setup() #not tested
```

For a really quick start, just hit ENTER on all questions. This will create all the necessary supporting files to get **DSage** running. It will create the databases, set up a private/public key for authentication and create a SSL certificate for the server.

3. Launch a server, monitor and get a connection to the server:

```
sage: D = dsage.start_all() #not tested
```

This will start 2 workers by default. You can change it by passing in the "workers=N" argument where "N" is the number of workers you want.

4. To do a computation, use D just like any other Sage interface. For example:

```
sage: j = D('2+2')  #not tested
sage: j.wait()      #not tested
sage: j             #not tested
4
```

kill_all()
Kills the server and worker.

kill_server()

kill_worker()

server(*blocking=True, port=None, log_level=0, ssl=True, db_file='/home/mvngu/.sage/dsage/db/dsage.db', log_file='/home/mvngu/.sage/dsage/server.log', privkey='/home/mvngu/.sage/dsage/cacert.pem', cert='/home/mvngu/.sage/dsage/pubcert.pem', authenticated_logins=False, failure_threshold=3, verbose=True, testing=False, profile=False*)
Run the Distributed Sage server.

Doing `dsage.server()` will spawn a server process which listens by default on port 8081.

setup(*template=None*)
This is the setup utility which helps you configure dsage.

Type `dsage.setup()` to run the configuration for the server, worker and client. Alternatively, if you want to run the configuration for just one parts, you can launch `dsage.setup_server()`, `dsage.setup worker()` or `dsage.setup()`.

setup_client()
This method runs the configuration utility for the client.

setup_server(**args*)
This method runs the configuration utility for the server.

setup_worker()
This method runs the configuration utility for the worker.

start_all(*port=None, workers=2, log_level=0, poll=1.0, authenticate=False, failure_threshold=3, verbose=True, testing=False*)
Start the server and worker and returns a connection to the server.

worker(*server='localhost', port=8081, workers=2, poll=1.0, username='mvngu', blocking=True, ssl=True, log_level=0, authenticate=True, priority=20, privkey='/home/mvngu/.sage/dsage/dsage_key', pubkey='/home/mvngu/.sage/dsage/dsage_key.pub', log_file='/home/mvngu/.sage/dsage/worker.log', verbose=True*)
Run the Distributed Sage worker.

Typing `sage.worker()` will launch a worker which by default connects to localhost on port 8081 to fetch jobs.

spawn(*cmd, verbose=True, stdout=None, stdin=None*)
Spawns a process and registers it with the Sage.

CHAPTER

SIXTEEN

CRYPTOGRAPHY

16.1 Cryptosystems.

class **Cryptosystem**(*plaintext_space, ciphertext_space, key_space, block_length=1, period=None*)

A cryptosystem is a pair of maps

$$E : \mathcal{K} \to \mathrm{Hom}(\mathcal{M}, \mathcal{C})$$

$$D : \mathcal{K} \to \mathrm{Hom}(\mathcal{C}, \mathcal{M})$$

where $\mathcal{K}$ is the keyspace, $\mathcal{M}$ is the plaintext or message space, and $\mathcal{C}$ is the ciphertext space. In many instances $\mathcal{M} = \mathcal{C}$ and the images will lie in $\mathrm{Aut}(\mathcal{M})$. An element of the image of E is called a cipher.

We may assume that E and D are injective, hence identify a key K in $\mathcal{K}$ with its image $E_K := E(K)$ in $\mathrm{Hom}(\mathcal{M}, \mathcal{C})$.

The cryptosystem has the property that for every encryption key K_1 there is a decryption key K_2 such that $D_{K_2} \circ E_{K_1}$. A cryptosystem with the property that $K := K_2 = K_1$, is called a symmetric cryptosystem. Otherwise, if the key $K_2 \neq K_1$, nor is K_2 easily derived from K_1, we call the cryptosystem asymmetric of public key. In that case, K_1 is called the public key and K_2 is called the private key.

block_length()

cipher_codomain()

cipher_domain()

ciphertext_space()

key_space()

period()

plaintext_space()

class **PublicKeyCryptosystem**(*plaintext_space, ciphertext_space, key_space, block_length=1, period=None*)

class **SymmetricKeyCryptosystem**(*plaintext_space, ciphertext_space, key_space, block_length=1, period=None*)

16.2 Ciphers.

class **Cipher**(*parent, key*)

Cipher class

codomain()

domain()

key()

parent()

class PublicKeyCipher(*parent, key, public=True*)
Public key cipher class

class SymmetricKeyCipher(*parent, key*)
Symmetric key cipher class

16.3 Classical Cryptosystems

class HillCryptosystem(*S, m*)
Create a Hill cryptosystem defined by the m x m matrix space over $\mathbf{Z}/N\mathbf{Z}$, where N is the alphabet size of the string monoid `S`.

INPUT:

- `S` - a string monoid over some alphabet
- `m` - integer > 0; the block length of matrices that specify block permutations

OUTPUT:

- A Hill cryptosystem of block length `m` over the alphabet `S`.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = HillCryptosystem(S,3)
sage: E
Hill cryptosystem on Free alphabetic string monoid on A-Z of block length 3
sage: R = IntegerModRing(26)
sage: M = MatrixSpace(R,3,3)
sage: A = M([[1,0,1],[0,1,1],[2,2,3]])
sage: A
[1 0 1]
[0 1 1]
[2 2 3]
sage: e = E(A)
sage: e
[1 0 1]
[0 1 1]
[2 2 3]
sage: e(S("LAMAISONBLANCHE"))
JYVKSKQPELAYKPV
```

TESTS:

```
sage: S = AlphabeticStrings()
sage: E = HillCryptosystem(S,3)
sage: E == loads(dumps(E))
True
```

block_length()
The row or column dimension of a matrix specifying a block permutation. Encryption and decryption keys of a Hill cipher are square matrices, i.e. the row and column dimensions of an encryption or decryption key are the same. This row/column dimension is referred to as the *block length*.
OUTPUT:

•The block length of an encryption/decryption key.

EXAMPLES:

```
sage: A = AlphabeticStrings()
sage: n = randint(1, A.ngens() - 1)
sage: H = HillCryptosystem(A, n)
sage: H.block_length() == n
True
```

deciphering(*A*, *C*)

Decrypt the ciphertext `C` using the key `A`.

INPUT:

•`A` - a key within the key space of this Hill cipher

•`C` - a string (possibly empty) over the string monoid of this Hill cipher

OUTPUT:

•The plaintext corresponding to the ciphertext `C`.

EXAMPLES:

```
sage: H = HillCryptosystem(AlphabeticStrings(), 3)
sage: K = H.random_key()
sage: M = H.encoding("Good day, mate! How ya going?")
sage: H.deciphering(K, H.enciphering(K, M)) == M
True
```

enciphering(*A*, *M*)

Encrypt the plaintext `M` using the key `A`.

INPUT:

•`A` - a key within the key space of this Hill cipher

•`M` - a string (possibly empty) over the string monoid of this Hill cipher.

OUTPUT:

•The ciphertext corresponding to the plaintext `M`.

EXAMPLES:

```
sage: H = HillCryptosystem(AlphabeticStrings(), 3)
sage: K = H.random_key()
sage: M = H.encoding("Good day, mate! How ya going?")
sage: H.deciphering(K, H.enciphering(K, M)) == M
True
```

encoding(*M*)

The encoding of the string `M` over the string monoid of this Hill cipher. For example, if the string monoid of this Hill cipher is `AlphabeticStringMonoid`, then the encoding of `M` would be its upper-case equivalent stripped of all non-alphabetic characters.

INPUT:

•`M` - a string, possibly empty

OUTPUT:

•The encoding of `M` over the string monoid of this Hill cipher.

EXAMPLES:

```
sage: M = "The matrix cipher by Lester S. Hill."
sage: A = AlphabeticStrings()
sage: H = HillCryptosystem(A, 7)
sage: H.encoding(M) == A.encoding(M)
True
```

inverse_key(*A*)

The inverse key corresponding to the key A.

INPUT:

•A - an invertible matrix of the key space of this Hill cipher

OUTPUT:

•The inverse matrix of A.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = HillCryptosystem(S,3)
sage: A = E.random_key()
sage: B = E.inverse_key(A)
sage: M = S("LAMAISONBLANCHE")
sage: e = E(A)
sage: c = E(B)
sage: c(e(M))
LAMAISONBLANCHE
```

random_key()

A random key within the key space of this Hill cipher. That is, generate a random m x m matrix to be used as a block permutation, where m is the block length of this Hill cipher. If n is the size of the cryptosystem alphabet, then there are n^{m^2} possible keys. However the number of valid keys, i.e. invertible m x m square matrices, is smaller than n^{m^2}.

OUTPUT:

•A random key within the key space of this Hill cipher.

EXAMPLES:

```
sage: A = AlphabeticStrings()
sage: n = 3
sage: H = HillCryptosystem(A, n)
sage: K = H.random_key()
sage: Ki = H.inverse_key(K)
sage: M = "LAMAISONBLANCHE"
sage: e = H(K)
sage: d = H(Ki)
sage: d(e(A(M))) == A(M)
True
```

class **SubstitutionCryptosystem**(*S*)

Create a substitution cryptosystem.

INPUT:

•S - a string monoid over some alphabet

OUTPUT:

•A substitution cryptosystem over the alphabet S.

EXAMPLES:

```
sage: M = AlphabeticStrings()
sage: E = SubstitutionCryptosystem(M)
sage: E
Substitution cryptosystem on Free alphabetic string monoid on A-Z
sage: K = M([ 25-i for i in range(26) ])
sage: K
```

```
ZYXWVUTSRQPONMLKJIHGFEDCBA
sage: e = E(K)
sage: m = M("THECATINTHEHAT")
sage: e(m)
GSVXZGRMGSVSZG
```

TESTS:

```
sage: M = AlphabeticStrings()
sage: E = SubstitutionCryptosystem(M)
sage: E == loads(dumps(E))
True
```

deciphering(*K*, *C*)

Decrypt the ciphertext `C` using the key `K`.

INPUT:

- `K` - a key belonging to the key space of this substitution cipher
- `C` - a string (possibly empty) over the string monoid of this cryptosystem.

OUTPUT:

- The plaintext corresponding to the ciphertext `C`.

EXAMPLES:

```
sage: S = SubstitutionCryptosystem(AlphabeticStrings())
sage: K = S.random_key()
sage: M = S.encoding("Don't substitute me!")
sage: S.deciphering(K, S.enciphering(K, M)) == M
True
```

enciphering(*K*, *M*)

Encrypt the plaintext `M` using the key `K`.

INPUT:

- `K` - a key belonging to the key space of this substitution cipher
- `M` - a string (possibly empty) over the string monoid of this cryptosystem.

OUTPUT:

- The ciphertext corresponding to the plaintext `M`.

EXAMPLES:

```
sage: S = SubstitutionCryptosystem(AlphabeticStrings())
sage: K = S.random_key()
sage: M = S.encoding("Don't substitute me.")
sage: S.deciphering(K, S.enciphering(K, M)) == M
True
```

encoding(*M*)

The encoding of the string `M` over the string monoid of this substitution cipher. For example, if the string monoid of this cryptosystem is `AlphabeticStringMonoid`, then the encoding of `M` would be its upper-case equivalent stripped of all non-alphabetic characters.

INPUT:

- `M` - a string, possibly empty

OUTPUT:

- The encoding of `M` over the string monoid of this cryptosystem.

EXAMPLES:

```
sage: M = "Peter Pan(ning) for gold."
sage: A = AlphabeticStrings()
sage: S = SubstitutionCryptosystem(A)
sage: S.encoding(M) == A.encoding(M)
True
```

inverse_key(*K*)

The inverse key corresponding to the key K. The specified key is a permutation of the cryptosystem alphabet.

INPUT:

- K - a key belonging to the key space of this cryptosystem

OUTPUT:

- The inverse key of K.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = SubstitutionCryptosystem(S)
sage: K = E.random_key()
sage: L = E.inverse_key(K)
sage: M = S("THECATINTHEHAT")
sage: e = E(K)
sage: c = E(L)
sage: c(e(M))
THECATINTHEHAT
```

random_key()

Generate a random key within the key space of this substitution cipher. The generated key is a permutation of the cryptosystem alphabet. Let n be the length of the alphabet. Then there are $n!$ possible keys in the key space.

OUTPUT:

- A random key within the key space of this cryptosystem.

EXAMPLES:

```
sage: A = AlphabeticStrings()
sage: S = SubstitutionCryptosystem(A)
sage: K = S.random_key()
sage: Ki = S.inverse_key(K)
sage: M = "THECATINTHEHAT"
sage: e = S(K)
sage: d = S(Ki)
sage: d(e(A(M))) == A(M)
True
```

class **TranspositionCryptosystem**(*S*, *n*)

Create a transposition cryptosystem of block length n.

INPUT:

- S - a string monoid over some alphabet
- n - integer > 0; a block length of a block permutation

OUTPUT:

- A transposition cryptosystem of block length n over the alphabet S.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = TranspositionCryptosystem(S,14)
sage: E
Transposition cryptosystem on Free alphabetic string monoid on A-Z of block length 14
sage: K = [ 14-i for i in range(14) ]
sage: K
[14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
sage: e = E(K)
sage: e(S("THECATINTHEHAT"))
TAHEHTNITACEHT
```

TESTS:

```
sage: S = AlphabeticStrings()
sage: E = TranspositionCryptosystem(S,14)
sage: E == loads(dumps(E))
True
```

deciphering(*K*, *C*)

Decrypt the ciphertext `C` using the key `K`.

INPUT:

- `K` - a key belonging to the key space of this transposition cipher
- `C` - a string (possibly empty) over the string monoid of this cryptosystem.

OUTPUT:

- The plaintext corresponding to the ciphertext `C`.

EXAMPLES:

```
sage: T = TranspositionCryptosystem(AlphabeticStrings(), 14)
sage: K = T.random_key()
sage: M = T.encoding("The cat in the hat.")
sage: T.deciphering(K, T.enciphering(K, M)) == M
True
```

enciphering(*K*, *M*)

Encrypt the plaintext `M` using the key `K`.

INPUT:

- `K` - a key belonging to the key space of this transposition cipher
- `M` - a string (possibly empty) over the string monoid of this cryptosystem

OUTPUT:

- The ciphertext corresponding to the plaintext `M`.

EXAMPLES:

```
sage: T = TranspositionCryptosystem(AlphabeticStrings(), 14)
sage: K = T.random_key()
sage: M = T.encoding("The cat in the hat.")
sage: T.deciphering(K, T.enciphering(K, M)) == M
True
```

encoding(*M*)

The encoding of the string `M` over the string monoid of this transposition cipher. For example, if the string monoid of this cryptosystem is `AlphabeticStringMonoid`, then the encoding of `M` would be its upper-case equivalent stripped of all non-alphabetic characters.

INPUT:

- `M` - a string, possibly empty

OUTPUT:

•The encoding of M over the string monoid of this cryptosystem.

EXAMPLES:

```
sage: M = "Transposition cipher is not about matrix transpose."
sage: A = AlphabeticStrings()
sage: T = TranspositionCryptosystem(A, 11)
sage: T.encoding(M) == A.encoding(M)
True
```

inverse_key(*K, check=True*)

The inverse key corresponding to the key K.

INPUT:

•K - a key belonging to the key space of this transposition cipher

•check - bool (default: True); check that K belongs to the key space of this cryptosystem.

OUTPUT:

•The inverse key corresponding to K.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = TranspositionCryptosystem(S, 14)
sage: K = E.random_key()
sage: Ki = E.inverse_key(K)
sage: e = E(K)
sage: d = E(Ki)
sage: M = "THECATINTHEHAT"
sage: C = e(S(M))
sage: d(S(C)) == S(M)
True
```

random_key()

Generate a random key within the key space of this transposition cryptosystem. Let $n > 0$ be the block length of this cryptosystem. Then there are $n!$ possible keys.

OUTPUT:

•A random key within the key space of this cryptosystem.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = TranspositionCryptosystem(S, 14)
sage: K = E.random_key()
sage: Ki = E.inverse_key(K)
sage: e = E(K)
sage: d = E(Ki)
sage: M = "THECATINTHEHAT"
sage: C = e(S(M))
sage: d(S(C)) == S(M)
True
```

class **VigenereCryptosystem**(*S, n*)

Create a Vigenere cryptosystem of block length n.

INPUT:

•S– a string monoid over some alphabet

•n - integer > 0; block length of an encryption/decryption key

OUTPUT:

•A Vigenere cryptosystem of block length n over the alphabet S.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = VigenereCryptosystem(S,14)
sage: E
Vigenere cryptosystem on Free alphabetic string monoid on A-Z of period 14
sage: K = S('ABCDEFGHIJKLMN')
sage: K
ABCDEFGHIJKLMN
sage: e = E(K)
sage: e
ABCDEFGHIJKLMN
sage: e(S("THECATINTHEHAT"))
TIGFEYOUBQOSMG
```

TESTS:

```
sage: S = AlphabeticStrings()
sage: E = VigenereCryptosystem(S,14)
sage: E == loads(dumps(E))
True
```

deciphering(*K*, *C*)

Decrypt the ciphertext C using the key K.

INPUT:

•K - a key belonging to the key space of this Vigenere cipher

•C - a string (possibly empty) over the string monoid of this cryptosystem

OUTPUT:

•The plaintext corresponding to the ciphertext C.

EXAMPLES:

```
sage: V = VigenereCryptosystem(AlphabeticStrings(), 24)
sage: K = V.random_key()
sage: M = V.encoding("Jack and Jill went up the hill.")
sage: V.deciphering(K, V.enciphering(K, M)) == M
True
```

enciphering(*K*, *M*)

Encrypt the plaintext M using the key K.

INPUT:

•K - a key belonging to the key space of this Vigenere cipher

•M - a string (possibly empty) over the string monoid of this cryptosystem

OUTPUT:

•The ciphertext corresponding to the plaintext M.

EXAMPLES:

```
sage: V = VigenereCryptosystem(AlphabeticStrings(), 24)
sage: K = V.random_key()
sage: M = V.encoding("Jack and Jill went up the hill.")
sage: V.deciphering(K, V.enciphering(K, M)) == M
True
```

encoding(*M*)

The encoding of the string `M` over the string monoid of this Vigenere cipher. For example, if the string monoid of this cryptosystem is `AlphabeticStringMonoid`, then the encoding of `M` would be its upper-case equivalent stripped of all non-alphabetic characters.

INPUT:

•`M` - a string, possibly empty

OUTPUT:

•The encoding of `M` over the string monoid of this cryptosystem.

EXAMPLES:

```
sage: A = AlphabeticStrings()
sage: V = VigenereCryptosystem(A, 24)
sage: M = "Jack and Jill went up the hill."
sage: V.encoding(M) == A.encoding(M)
True
```

inverse_key(*K*)

The inverse key corresponding to the key `K`.

INPUT:

•`K` - a key within the key space of this Vigenere cryptosystem

OUTPUT:

•The inverse key corresponding to `K`.

EXAMPLES:

```
sage: S = AlphabeticStrings()
sage: E = VigenereCryptosystem(S,14)
sage: K = E.random_key()
sage: L = E.inverse_key(K)
sage: M = S("THECATINTHEHAT")
sage: e = E(K)
sage: c = E(L)
sage: c(e(M))
THECATINTHEHAT
```

random_key()

Generate a random key within the key space of this Vigenere cryptosystem. Let $n > 0$ be the length of the cryptosystem alphabet and let $m > 0$ be the block length of this cryptosystem. Then there are n^m possible keys.

OUTPUT:

•A random key within the key space of this cryptosystem.

EXAMPLES:

```
sage: A = AlphabeticStrings()
sage: V = VigenereCryptosystem(A, 14)
sage: M = "THECATINTHEHAT"
sage: K = V.random_key()
sage: Ki = V.inverse_key(K)
sage: e = V(K)
sage: d = V(Ki)
sage: d(e(A(M))) == A(M)
True
```

16.4 Classical Ciphers.

class **HillCipher**(*parent, key*)
Hill cipher class

inverse()

class **SubstitutionCipher**(*parent, key*)
Substitution cipher class

inverse()

class **TranspositionCipher**(*parent, key*)
Transition cipher class

inverse()

class **VigenereCipher**(*parent, key*)
Vigenere cipher class

inverse()

16.5 Stream Cryptosystems.

class **LFSRCryptosystem**(*field=None*)
Linear feedback shift register cryptosystem class

encoding(*M*)

class **ShrinkingGeneratorCryptosystem**(*field=None*)
Shrinking generator cryptosystem class

encoding(*M*)

16.6 Stream Ciphers

class **LFSRCipher**(*parent, poly, IS*)

connection_polynomial()
The connection polynomial defining the LFSR of the cipher.
EXAMPLE:

```
sage: k = GF(2)
sage: P.<x> = PolynomialRing( k )
sage: LFSR = LFSRCryptosystem( k )
sage: e = LFSR((x^2+x+1,[k(0), k(1)]))
sage: e.connection_polynomial()
x^2 + x + 1
```

initial_state()
The initial state of the LFSR cipher.
EXAMPLE:

```
sage: k = GF(2)
sage: P.<x> = PolynomialRing( k )
sage: LFSR = LFSRCryptosystem( k )
sage: e = LFSR((x^2+x+1,[k(0), k(1)]))
```

```
sage: e.initial_state()
[0, 1]
```

class ShrinkingGeneratorCipher(*parent, e1, e2*)

decimating_cipher()

The LFSR cipher generating the decimating key stream.

EXAMPLE:

```
        sage: FF = FiniteField(2)
        sage: P.<x> = PolynomialRing(FF)
        sage: LFSR = LFSRCryptosystem(FF)
        sage: IS_1 = [ FF(a) for a in [0,1,0,1,0,0,0] ]
        sage: e1 = LFSR((x^7 + x + 1,IS_1))
        sage: IS_2 = [ FF(a) for a in [0,0,1,0,0,0,1,0,1] ]
        sage: e2 = LFSR((x^9 + x^3 + 1,IS_2))
        sage: E = ShrinkingGeneratorCryptosystem()
        sage: e = E((e1,e2))
    sage: e.decimating_cipher()
        (x^9 + x^3 + 1, [0, 0, 1, 0, 0, 0, 1, 0, 1])
```

keystream_cipher()

The LFSR cipher generating the output key stream.

EXAMPLE:

```
        sage: FF = FiniteField(2)
        sage: P.<x> = PolynomialRing(FF)
        sage: LFSR = LFSRCryptosystem(FF)
        sage: IS_1 = [ FF(a) for a in [0,1,0,1,0,0,0] ]
        sage: e1 = LFSR((x^7 + x + 1,IS_1))
        sage: IS_2 = [ FF(a) for a in [0,0,1,0,0,0,1,0,1] ]
        sage: e2 = LFSR((x^9 + x^3 + 1,IS_2))
        sage: E = ShrinkingGeneratorCryptosystem()
        sage: e = E((e1,e2))
    sage: e.keystream_cipher()
        (x^7 + x + 1, [0, 1, 0, 1, 0, 0, 0])
```

16.7 Linear feedback shift register (LFSR) sequence commands.

Stream ciphers have been used for a long time as a source of pseudo-random number generators.

S. Golomb [G] gives a list of three statistical properties a sequence of numbers $\mathbf{a} = \{a_n\}_{n=1}^{\infty}$, $a_n \in \{0, 1\}$, should display to be considered "random". Define the autocorrelation of $\mathbf{a}$ to be

$$C(k) = C(k, \mathbf{a}) = \lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} (-1)^{a_n + a_{n+k}}.$$

In the case where $\mathbf{a}$ is periodic with period P then this reduces to

$$C(k) = \frac{1}{P} \sum_{n=1}^{P} (-1)^{a_n + a_{n+k}}.$$

Assume $\mathbf{a}$ is periodic with period P.

- balance: $|\sum_{n=1}^{P} (-1)^{a_n}| \leq 1$.

- low autocorrelation:

$$C(k) = \begin{cases} 1, & k = 0, \\ \epsilon, & k \neq 0. \end{cases}$$

  (For sequences satisfying these first two properties, it is known that $\epsilon = -1/P$ must hold.)

- proportional runs property: In each period, half the runs have length 1, one-fourth have length 2, etc. Moreover, there are as many runs of 1's as there are of 0's.

A general feedback shift register is a map $f : \mathbf{F}_q^d \to \mathbf{F}_q^d$ of the form

$$f(x_0, ..., x_{n-1}) = (x_1, x_2, ..., x_n),$$
$$x_n = C(x_0, ..., x_{n-1}),$$

where $C : \mathbf{F}_q^d \to \mathbf{F}_q$ is a given function. When C is of the form

$$C(x_0, ..., x_{n-1}) = a_0 x_0 + ... + a_{n-1} x_{n-1},$$

for some given constants $a_i \in \mathbf{F}_q$, the map is called a linear feedback shift register (LFSR).

Example of a LFSR Let

$$f(x) = a_0 + a_1 x + ... + a_n x^n + ...,$$

$$g(x) = b_0 + b_1 x + ... + b_n x^n + ...,$$

be given polynomials in $\mathbf{F}_2[x]$ and let

$$h(x) = \frac{f(x)}{g(x)} = c_0 + c_1 x + ... + c_n x^n + ...$$

We can compute a recursion formula which allows us to rapidly compute the coefficients of $h(x)$ (take $f(x) = 1$):

$$c_n = \sum_{i=1}^{n} \frac{-b_i}{b_0} c_{n-i}.$$

The coefficients of $h(x)$ can, under certain conditions on $f(x)$ and $g(x)$, be considered "random" from certain statistical points of view.

Example: For instance, if

$$f(x) = 1, \quad g(x) = x^4 + x + 1,$$

then

$$h(x) = 1 + x + x^2 + x^3 + x^5 + x^7 + x^8 + ...$$

The coefficients of h are

$$1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1,$$
$$1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, ...$$

The sequence of $0, 1$'s is periodic with period $P = 2^4 - 1 = 15$ and satisfies Golomb's three randomness conditions. However, this sequence of period 15 can be "cracked" (i.e., a procedure to reproduce $g(x)$) by knowing only 8 terms! This is the function of the Berlekamp-Massey algorithm [M], implemented as `berlekamp_massey.py`.

[G] Solomon Golomb, Shift register sequences, Aegean Park Press, Laguna Hills, Ca, 1967

[M] James L. Massey, "Shift-Register Synthesis and BCH Decoding." IEEE Trans. on Information Theory, vol. 15(1), pp. 122-127, Jan 1969.

AUTHORS:

- Timothy Brock

Created 11-24-2005 by wdj. Last updated 12-02-2005.

lfsr_autocorrelation(*L, p, k*)

INPUT:

- L - is a periodic sequence of elements of ZZ or GF(2). L must have length = p
- p - the period of L
- k - k is an integer (0 k p)

OUTPUT: autocorrelation sequence of L

EXAMPLES:

```
sage: F = GF(2)
sage: o = F(0)
sage: l = F(1)
sage: key = [l,o,o,l]; fill = [l,l,o,l]; n = 20
sage: s = lfsr_sequence(key,fill,n)
sage: lfsr_autocorrelation(s,15,7)
4/15
sage: lfsr_autocorrelation(s,int(15),7)
4/15
```

AUTHORS:

- Timothy Brock (2006-04-17)

lfsr_connection_polynomial(*s*)

INPUT:

- s - a sequence of elements of a finite field (F) of even length

OUTPUT:

- C(x) - the connection polynomial of the minimal LFSR.

This implements the algorithm in section 3 of J. L. Massey's article [M].

EXAMPLE:

```
        sage: F = GF(2)
sage: F
Finite Field of size 2
sage: o = F(0); l = F(1)
       sage: key = [l,o,o,l]; fill = [l,l,o,l]; n = 20
       sage: s = lfsr_sequence(key,fill,n); s
       [1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0]
       sage: lfsr_connection_polynomial(s)
       x^4 + x + 1
       sage: berlekamp_massey(s)
       x^4 + x^3 + 1
```

Notice that berlekamp_massey returns the reverse of the connection polynomial (and is potentially must faster than this implementation).

AUTHORS:

- Timothy Brock (2006-04-17)

REFERENCES:

•[M] J. L. Massey, 'Shift-register synthesis and BCH decoding,' IEEE Trans. Inform. Theory, vol. IT-15, pp. 122-127, Jan. 19 69.

lfsr_sequence (*key, fill, n*)

This function creates an lfsr sequence.

INPUT:

•`key` - a list of finite field elements, [c_0,c_1,...,c_k].

•`fill` - the list of the initial terms of the lfsr sequence, [x_0,x_1,...,x_k].

•`n` - number of terms of the sequence that the function returns.

OUTPUT: The lfsr sequence defined by $x_{n+1} = c_k x_n + ... + c_0 x_{n-k}$, for $n \leq k$.

EXAMPLES:

```
sage: F = GF(2); l = F(1); o = F(0)
sage: F = GF(2); S = LaurentSeriesRing(F,'x'); x = S.gen()
sage: fill = [l,l,o,l]; key = [1,o,o,l]; n = 20
sage: L = lfsr_sequence(key,fill,20); L
[1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0]
sage: g = berlekamp_massey(L); g
x^4 + x^3 + 1
sage: (1)/(g.reverse()+O(x^20))
1 + x + x^2 + x^3 + x^5 + x^7 + x^8 + x^11 + x^15 + x^16 + x^17 + x^18 + O(x^20)
sage: (1+x^2)/(g.reverse()+O(x^20))
1 + x + x^4 + x^8 + x^9 + x^10 + x^11 + x^13 + x^15 + x^16 + x^19 + O(x^20)
sage: (1+x^2+x^3)/(g.reverse()+O(x^20))
1 + x + x^3 + x^5 + x^6 + x^9 + x^13 + x^14 + x^15 + x^16 + x^18 + O(x^20)
sage: fill = [l,l,o,l]; key = [l,o,o,o]; n = 20
sage: L = lfsr_sequence(key,fill,20); L
[1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1]
sage: g = berlekamp_massey(L); g
x^4 + 1
sage: (1+x)/(g.reverse()+O(x^20))
1 + x + x^4 + x^5 + x^8 + x^9 + x^12 + x^13 + x^16 + x^17 + O(x^20)
sage: (1+x+x^3)/(g.reverse()+O(x^20))
1 + x + x^3 + x^4 + x^5 + x^7 + x^8 + x^9 + x^11 + x^12 + x^13 + x^15 + x^16 + x^17 + x^19 + O(x
```

AUTHORS:

•Timothy Brock (2005-11): with code modified from Python Cookbook, http://aspn.activestate.com/ASPN/Python/Cookbook/

16.8 Small Scale Variants of the AES (SR) Polynomial System Generator.

Sage supports polynomial system generation for small scale (and full scale) AES variants over $\mathbf{F}_2$ and $\mathbf{F}_{2^e}$. Also, Sage supports both the specification of SR as given in the papers [CMR05] and [CMR06] and a variant of SR* which is equivalent to AES.

AUTHORS:

- Martin Albrecht (2008,2009-01): usability improvements
- Martin Albrecht (2007-09): initial version

EXAMPLES:

We construct SR(1,1,1,4) and study its properties.

```
sage: sr = mq.SR(1, 1, 1, 4)
```

`n` is the number of rounds, `r` the number of rows in the state array, `c` the number of columns in the state array, and `e` the degree of the underlying field.

```
sage: sr.n, sr.r, sr.c, sr.e
(1, 1, 1, 4)
```

By default variables are ordered reverse to as they appear, e.g.:

```
sage: print sr.R.repr_long()
Polynomial Ring
  Base Ring : Finite Field in a of size 2^4
       Size : 20 Variables
   Block  0 : Ordering : degrevlex
              Names    : k100, k101, k102, k103, x100, x101, x102, x103, w100, w101, w102, w103, s00
```

However, this can be prevented by passing in `reverse_variables=False` to the constructor.

For SR(1, 1, 1, 4) the `ShiftRows` matrix isn't that interesting.:

```
sage: sr.ShiftRows
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

Also, the `MixColumns` matrix is the identity matrix.:

```
sage: sr.MixColumns
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

`Lin`, however, is not the identity matrix.:

```
sage: sr.Lin
[          a^2 + 1                 1         a^3 + a^2           a^2 + 1]
[               a                  a                 1 a^3 + a^2 + a + 1]
[          a^3 + a               a^2               a^2                 1]
[               1               a^3             a + 1             a + 1]
```

`M` and `Mstar` are identical for SR(1, 1, 1, 4):

```
sage: sr.M
[          a^2 + 1                 1         a^3 + a^2           a^2 + 1]
[               a                  a                 1 a^3 + a^2 + a + 1]
[          a^3 + a               a^2               a^2                 1]
[               1               a^3             a + 1             a + 1]
```

```
sage: sr.Mstar
[          a^2 + 1                 1         a^3 + a^2           a^2 + 1]
[                a                 a                 1 a^3 + a^2 + a + 1]
[          a^3 + a               a^2               a^2                 1]
[                1               a^3             a + 1             a + 1]
```

We can compute a Groebner basis for the ideals spanned by SR instances to recover all solutions to the system.:

```
sage: sr = mq.SR(1,1,1,4, gf2=True, polybori=True)
sage: K = sr.base_ring()
sage: a = K.gen()
sage: K = [a]
sage: P = [1]
sage: F,s = sr.polynomial_system(P=P, K=K)
sage: F.groebner_basis()
[k100, k101 + 1, k102, k103 + k003,
 x100 + 1, x101 + k003 + 1, x102 + k003 + 1,
 x103 + k003, w100, w101, w102 + 1, w103 + k003 + 1,
 s000 + 1, s001 + k003, s002 + k003, s003 + k003 + 1,
 k000, k001, k002 + 1]
```

Note that the order of `k000`, `k001`, `k002` and `k003` is little endian. Thus the result `k002 + 1, k001, k000` indicates that the key is either a or $a + 1$. We can verify that both keys encrypt P to the same ciphertext:

```
sage: sr(P,[a])
[0]
sage: sr(P,[a+1])
[0]
```

All solutions can easily be recovered using the variety function for ideals.:

```
sage: I = F.ideal()
sage: for V in I.variety():
...     for k,v in sorted(V.iteritems()):
...         print k,v
...     print
k003 0
k002 1
k001 0
k000 0
s003 1
s002 0
s001 0
s000 1
w103 1
w102 1
w101 0
w100 0
x103 0
x102 1
x101 1
x100 1
k103 0
k102 0
k101 1
k100 0
```

```
<BLANKLINE>
k003 1
k002 1
k001 0
k000 0
s003 0
s002 1
s001 1
s000 1
w103 0
w102 1
w101 0
w100 0
x103 1
x102 0
x101 0
x100 1
k103 1
k102 0
k101 1
k100 0
```

Note that the S-Box object for SR can be constructed with a call to `sr.sbox()`:

```
sage: sr = mq.SR(1,1,1,4, gf2=True, polybori=True)
sage: S = sr.sbox()
```

For example, we can now study the difference distribution matrix of `S`:

```
sage: S.difference_distribution_matrix()
[16  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  2  2  2  2  0  0  0  2  0  0  0  2  4  0  0]
[ 0  2  0  4  2  2  2  0  0  2  0  0  0  0  0  2]
[ 0  2  4  0  0  2  0  0  2  2  0  2  0  0  2  0]
[ 0  0  2  0  4  2  0  0  0  0  2  0  2  0  2  2]
[ 0  0  0  2  0  0  0  2  4  2  0  0  2  0  2  2]
[ 0  4  0  0  0  2  0  2  0  2  2  0  2  2  0  0]
[ 0  2  0  0  0  0  2  0  0  0  0  2  4  2  2  2]
[ 0  2  2  0  0  0  2  2  2  0  2  0  0  0  0  4]
[ 0  0  2  2  0  0  0  0  0  2  2  4  0  2  0  2]
[ 0  0  2  0  2  0  2  2  0  4  0  2  2  0  0  0]
[ 0  0  0  0  2  0  2  0  2  2  4  0  0  2  2  0]
[ 0  0  0  2  0  4  2  0  2  0  2  2  2  0  0  0]
[ 0  0  0  0  2  2  0  4  2  0  0  2  0  2  0  2]
[ 0  0  2  2  0  2  4  2  0  0  0  0  0  2  2  0]
[ 0  2  0  2  2  0  0  2  0  0  2  2  0  0  4  0]
```

or use `S` to find alternative polynomial representations for the S-Box.:

```
sage: S.polynomials(degree=3)
[x0*x1 + x1*x2 + x0*x3 + x0*y2 + x1 + y0 + y1 + 1,
 x0*x1 + x0*x2 + x0*y0 + x0*y1 + x0*y2 + x1 + x2 + y0 + y1 + y2,
 x0*x1 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x1*y0 + x0*y1 + x0*y3,
 x0*x1 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x1*y1 + x0*y3 + x1 + y0 + y1 + 1,
 x0*x1 + x0*x2 + x0*y2 + x1*y2 + x0*y3 + x0 + x1,
 x0*x3 + x1*x3 + x0*y1 + x0*y2 + x1*y3 + x0 + x1 + x2 + x3 + y0 + y1 + y3 + 1,
 x0*x1 + x1*x3 + x2*x3 + x0*y0 + x0*y2 + x0*y3 + x2 + y0 + y3,
```

```
x0*x1 + x0*x2 + x0*x3 + x1*x3 + x2*y0 + x0*y2 + x0 + x2 + x3 + y3,
x0*x3 + x1*x3 + x0*y0 + x2*y1 + x0*y2 + x3 + y3,
x0*x1 + x0*x2 + x0*y0 + x0*y1 + x2*y2 + x0*y3 + x1 + y0 + y1 + 1,
x0*x3 + x1*x3 + x0*y0 + x0*y1 + x0*y3 + x2*y3 + y0 + y3,
x0*x1 + x0*x2 + x3*y0 + x0*y1 + x0*y3 + y0,
x0*y0 + x0*y1 + x3*y1 + x0 + x2 + y0 + y3,
x0*y0 + x3*y2 + y0,
x0*x1 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x0*y2 + x3*y3 + y0,
x0*x2 + x0*x3 + x0*y1 + y0*y1 + x0*y3 + x2 + x3 + y3,
x0*x2 + x0*y0 + y0*y2 + x0*y3 + x0 + y0,
x0*x1 + x0*x2 + x1*x3 + x0*y2 + y0*y3 + y0,
x0*x1 + x0*y0 + y1*y2 + x0*y3 + x1 + x2 + y0 + 1,
x0*x2 + x1*x3 + x0*y1 + x0*y2 + x0*y3 + y1*y3 + x0 + y0 + y3,
x0*x1 + x0*x2 + x0*x3 + x0*y1 + x0*y2 + x0*y3 + y2*y3 + x0 + x1 + x2 + x3 + y1 + y3 + 1,
x0*x1*x2 + x0*x3 + x0*y0 + x0*y1 + x0*y2 + x0,
x0*x1*x3 + x0*x2 + x0*x3 + x0*y1 + x0*y3 + x0,
x0*x1*y0 + x0*x1 + x0*y0 + x0,
x0*x1*y1,
x0*x1*y2 + x0*x2 + x0*y2 + x0*y3 + x0,
x0*x1*y3 + x0*x1 + x0*x3 + x0*y0 + x0*y1 + x0*y2 + x0,
x0*x2*x3 + x0*x1 + x0*x3 + x0*y1 + x0*y2 + x0*y3 + x0,
x0*x2*y0 + x0*x1 + x0*x2 + x0*x3 + x0*y1 + x0*y2,
x0*x2*y1 + x0*x2 + x0*x3 + x0*y0 + x0*y1 + x0*y2 + x0,
x0*x2*y2 + x0*x2 + x0*y3 + x0,
x0*x2*y3 + x0*x2 + x0*y3 + x0,
x0*x3*y0 + x0*x1 + x0*x2 + x0*y0 + x0*y1 + x0*y3,
x0*x3*y1 + x0*x2 + x0*y1 + x0*y3 + x0,
x0*x3*y2,
x0*x3*y3 + x0*x1 + x0*y1 + x0*y2 + x0*y3 + x0,
x0*y0*y1 + x0*y1,
x0*y0*y2 + x0*x2 + x0*y3 + x0,
x0*y0*y3 + x0*x1 + x0*x3 + x0*y0 + x0*y1 + x0*y2 + x0*y3 + x0,
x0*y1*y2 + x0*x2 + x0*y3 + x0,
x0*y1*y3 + x0*x3 + x0*y0 + x0*y2 + x0*y3,
x0*y2*y3 + x0*y2,
x1*x2*x3 + x0*x1 + x1*x3 + x0*y0 + x0*y1 + x2 + x3 + y3,
x1*x2*y0 + x0*x1 + x1*x3 + x0*y0 + x0*y1 + x2 + x3 + y3,
x1*x2*y1 + x0*x1 + x1*x3 + x0*y0 + x1 + x2 + x3 + y0 + y1 + y3 + 1,
x1*x2*y2 + x0*x1 + x0*y0 + x0*y1 + x0 + x1 + y0 + y1 + 1,
x1*x2*y3 + x0*x1 + x1*x3 + x0*y0 + x1 + x2 + x3 + y0 + y1 + y3 + 1,
x1*x3*y0 + x0*x1 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x0*y1 + x0*y3,
x1*x3*y1 + x0*x2 + x0*x3 + x0*y3 + x2 + x3 + y3,
x1*x3*y2 + x0*x2 + x0*x3 + x1*x3 + x0*y1 + x0*y3 + x0,
x1*x3*y3 + x0*x1 + x0*x2 + x0*x3 + x0*y0 + x0*y1 + x0*y3,
x1*y0*y1 + x0*x2 + x0*x3 + x0*y3 + x2 + x3 + y3,
x1*y0*y2 + x0*x2 + x0*x3 + x1*x3 + x0*y1 + x0*y3 + x0,
x1*y0*y3,
x1*y1*y2 + x0*x1 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x0*y3 + x1 + y0 + y1 + 1,
x1*y1*y3 + x0*x1 + x1*x3 + x0*y0 + x1 + x2 + x3 + y0 + y1 + y3 + 1,
x1*y2*y3 + x0*x1 + x0*x2 + x1*x3 + x0*y0 + x0*y2 + x0*y3 + x0 + x1 + x2 + x3 + y0 + y1 + y3 + 1,
x2*x3*y0 + x0*x1 + x0*x3 + x1*x3 + x0*y2 + x0*y3 + x2 + x3 + y3,
x2*x3*y1 + x0*y1 + x0*y2 + x0*y3 + x3 + y0,
x2*x3*y2 + x1*x3 + x0*y1 + x0 + x2 + x3 + y3,
x2*x3*y3,
x2*y0*y1 + x0*x2 + x0*x3 + x0*y0 + x0*y1 + x0*y2 + x0,
x2*y0*y2 + x0*x2 + x1*x3 + x0*y1 + x0*y3 + x2 + x3 + y3,
x2*y0*y3 + x0*x2 + x0*y3 + x0,
x2*y1*y2 + x0*x1 + x0*x2 + x1*x3 + x0*y0 + x0*y3 + x0 + x1 + x2 + x3 + y0 + y1 + y3 + 1,
```

```
 x2*y1*y3 + x0*x3 + x1*x3 + x0*y0 + x0*y1 + x0*y3 + y0 + y3,
 x2*y2*y3 + x0*x1 + x0*x2 + x1*x3 + x0*y0 + x0*y3 + x0 + x1 + x2 + x3 + y0 + y1 + y3 + 1,
 x3*y0*y1 + x0*x3 + x0*y1 + x0 + x2 + x3 + y3,
 x3*y0*y2 + x0*y0 + y0,
 x3*y0*y3 + x1*x3 + x0*y1 + x0*y2 + x0*y3 + y0,
 x3*y1*y2 + x0*x2 + x0*x3 + x0*y3 + x2 + x3 + y3,
 x3*y1*y3 + x0*x2 + x0*x3 + x0*y0 + x0*y2 + x0,
 x3*y2*y3 + x0*x2 + x0*x3 + x1*x3 + x0*y0 + x0*y1 + x0*y3 + x0 + y0,
 y0*y1*y2 + x0*x3 + x0 + x2 + x3 + y3,
 y0*y1*y3 + x0*x3 + x0*y0 + x0*y2 + x0*y3,
 y0*y2*y3 + x0*x3 + x1*x3 + x0*y0 + x0*y1 + y0,
 y1*y2*y3 + x0*x1 + x0*x2 + x1*x3 + x0*y0 + x0*y3 + x0 + x1 + x2 + x3 + y0 + y1 + y3 + 1]
```

```
sage: S.interpolation_polynomial()
(a^2 + 1)*x^14 + x^13 + (a^3 + a^2)*x^11 + (a^2 + 1)*x^7 + a^2 + a
```

The SR_gf2_2 gives an example how use alternative polynomial representations of the S-Box for construction of polynomial systems.

TESTS:

```
sage: sr == loads(dumps(sr))
True
```

REFERENCES:

class AllowZeroInversionsContext(*sr*)

Temporarily allow zero inversion.

SR(*n=1, r=1, c=1, e=4, star=False, **kwargs*)

Return a small scale variant of the AES polynomial system constructor subject to the following conditions:

INPUT:

- n - the number of rounds (default: 1)
- r - the number of rows in the state array (default: 1)
- c - the number of columns in the state array (default: 1)
- e - the exponent of the finite extension field (default: 4)
- star - determines if SR* or SR should be constructed (default: False)
- aes_mode - as the SR key schedule specification differs slightly from the AES key schedule, this parameter controls which schedule to use (default: True)
- gf2 - generate polynomial systems over $\mathbf{F}_2$ rather than over $\mathbf{F}_{2^e}$ (default: False)
- polybori - use the BooleanPolynomialRing as polynomial representation (default: False, $\mathbf{F}_2$ only)
- order - a string to specify the term ordering of the variables
- postfix - a string which is appended after the variable name (default: '')
- allow_zero_inversions - a boolean to control whether zero inversions raise an exception (default: False)
- correct_only - only include correct inversion polynomials (default: False, $\mathbf{F}_2$ only)
- biaffine_only - only include bilinear and biaffine inversion polynomials (default: True, $\mathbf{F}_2$ only)

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 4)
sage: ShiftRows = sr.shift_rows_matrix()
sage: MixColumns = sr.mix_columns_matrix()
sage: Lin = sr.lin_matrix()
sage: M = MixColumns * ShiftRows * Lin
sage: print sr.hex_str_matrix(M)
 5 1 C 5
 2 2 1 F
 A 4 4 1
 1 8 3 3

sage: sr = mq.SR(1, 2, 1, 4)
sage: ShiftRows = sr.shift_rows_matrix()
sage: MixColumns = sr.mix_columns_matrix()
sage: Lin = sr.lin_matrix()
sage: M = MixColumns * ShiftRows * Lin
sage: print sr.hex_str_matrix(M)
 F 3 7 F A 2 B A
 A A 5 6 8 8 4 9
 7 8 8 2 D C C 3
 4 6 C C 5 E F F
 A 2 B A F 3 7 F
 8 8 4 9 A A 5 6
 D C C 3 7 8 8 2
 5 E F F 4 6 C C

sage: sr = mq.SR(1, 2, 2, 4)
sage: ShiftRows = sr.shift_rows_matrix()
sage: MixColumns = sr.mix_columns_matrix()
sage: Lin = sr.lin_matrix()
sage: M = MixColumns * ShiftRows * Lin
sage: print sr.hex_str_matrix(M)
 F 3 7 F 0 0 0 0 0 0 0 0 A 2 B A
 A A 5 6 0 0 0 0 0 0 0 0 8 8 4 9
 7 8 8 2 0 0 0 0 0 0 0 0 D C C 3
 4 6 C C 0 0 0 0 0 0 0 0 5 E F F
 A 2 B A 0 0 0 0 0 0 0 0 F 3 7 F
 8 8 4 9 0 0 0 0 0 0 0 0 A A 5 6
 D C C 3 0 0 0 0 0 0 0 0 7 8 8 2
 5 E F F 0 0 0 0 0 0 0 0 4 6 C C
 0 0 0 0 A 2 B A F 3 7 F 0 0 0 0
 0 0 0 0 8 8 4 9 A A 5 6 0 0 0 0
 0 0 0 0 D C C 3 7 8 8 2 0 0 0 0
 0 0 0 0 5 E F F 4 6 C C 0 0 0 0
 0 0 0 0 F 3 7 F A 2 B A 0 0 0 0
 0 0 0 0 A A 5 6 8 8 4 9 0 0 0 0
 0 0 0 0 7 8 8 2 D C C 3 0 0 0 0
 0 0 0 0 4 6 C C 5 E F F 0 0 0 0
```

class **SR_generic**(*n=1, r=1, c=1, e=4, star=False, **kwargs*)

add_round_key(*d, key*)

Perform the `AddRoundKey` operation on `d` using `key`.

INPUT:

- `d` - state array or something coercible to a state array
- `key` - state array or something coercible to a state array

EXAMPLE:

```
sage: sr = mq.SR(10, 4, 4, 4)
sage: D = sr.random_state_array()
sage: K = sr.random_state_array()
sage: sr.add_round_key(D, K) == K + D
True
```

base_ring()

Return the base field of self as determined by `self.e`.

EXAMPLE:

```
sage: sr = mq.SR(10, 2, 2, 4)
sage: sr.base_ring().polynomial()
a^4 + a + 1
```

The Rijndael polynomial:

```
sage: sr = mq.SR(10, 4, 4, 8)
sage: sr.base_ring().polynomial()
a^8 + a^4 + a^3 + a + 1
```

block_order()

Return a block order for self where each round is a block.

EXAMPLE:

```
sage: sr = mq.SR(2, 1, 1, 4)
sage: sr.block_order()
degrevlex(16),degrevlex(16),degrevlex(4) term order

sage: P = sr.ring(order='block')
sage: print P.repr_long()
Polynomial Ring
  Base Ring : Finite Field in a of size 2^4
       Size : 36 Variables
   Block  0 : Ordering : degrevlex
              Names    : k200, k201, k202, k203, x200, x201, x202, x203, w200, w201, w202, w
   Block  1 : Ordering : degrevlex
              Names    : k100, k101, k102, k103, x100, x101, x102, x103, w100, w101, w102, w
   Block  2 : Ordering : degrevlex
              Names    : k000, k001, k002, k003
```

hex_str(*M*, *typ='matrix'*)

Return a hex string for the provided AES state array/matrix.

INPUT:

- `M` - state array
- `typ` - controls what to return, either 'matrix' or 'vector' (default: `'matrix'`)

EXAMPLE:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: k = sr.base_ring()
sage: A = matrix(k, 2, 2, [1, k.gen(), 0, k.gen()^2])
sage: sr.hex_str(A)
' 1 2 \n 0 4 \n'

sage: sr.hex_str(A, typ='vector')
'1024'
```

hex_str_matrix(*M*)

Return a two-dimensional AES-like representation of the matrix M.

That is, show the finite field elements as hex strings.

INPUT:

- M - an AES state array

EXAMPLE:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: k = sr.base_ring()
sage: A = matrix(k, 2, 2, [1, k.gen(), 0, k.gen()^2])
sage: sr.hex_str_matrix(A)
' 1 2 \n 0 4 \n'
```

hex_str_vector(*M*)

Return a one-dimensional AES-like representation of the matrix M.

That is, show the finite field elements as hex strings.

INPUT:

- M - an AES state array

EXAMPLE:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: k = sr.base_ring()
sage: A = matrix(k, 2, 2, [1, k.gen(), 0, k.gen()^2])
sage: sr.hex_str_vector(A)
'1024'
```

is_state_array(*d*)

Return `True` if `d` is a state array, i.e. has the correct dimensions and base field.

EXAMPLE:

```
sage: sr = mq.SR(2, 2, 4, 8)
sage: k = sr.base_ring()
sage: sr.is_state_array( matrix(k, 2, 4) )
True

sage: sr = mq.SR(2, 2, 4, 8)
sage: k = sr.base_ring()
sage: sr.is_state_array( matrix(k, 4, 4) )
False
```

key_schedule(*kj*, *i*)

Return k_i for a given i and k_j with $j = i - 1$.

EXAMPLES:

```
sage: sr = mq.SR(10, 4, 4, 8, star=True, allow_zero_inversions=True)
sage: ki = sr.state_array()
sage: for i in range(10):
...   ki = sr.key_schedule(ki, i+1)
sage: print sr.hex_str_matrix(ki)
B4 3E 23 6F
EF 92 E9 8F
5B E2 51 18
CB 11 CF 8E
```

key_schedule_polynomials(*i*)

Return polynomials for the i-th round of the key schedule.

INPUT:

- i - round ($0 \leq i \leq n$)

EXAMPLE:

```
sage: sr = mq.SR(1, 1, 1, 4, gf2=True)
```

The 0-th subkey is the user provided key, so only conjugacy relations are added.

```
sage: sr.key_schedule_polynomials(0)
(k000^2 + k000, k001^2 + k001, k002^2 + k002, k003^2 + k003)
```

The 1-th subkey is derived from the user provided key according to the key schedule which is non-linear.

```
sage: sr.key_schedule_polynomials(1)
(k100 + s000 + s002 + s003,
 k101 + s000 + s001 + s003 + 1,
 k102 + s000 + s001 + s002 + 1,
 k103 + s001 + s002 + s003 + 1,
 k100^2 + k100, k101^2 + k101, k102^2 + k102, k103^2 + k103,
 s000^2 + s000, s001^2 + s001, s002^2 + s002, s003^2 + s003,
 s000*k000 + s003*k000 + s002*k001 + s001*k002 + s000*k003,
 s000*k000 + s001*k000 + s000*k001 + s003*k001 + s002*k002 + s001*k003,
 s001*k000 + s002*k000 + s000*k001 + s001*k001 + s000*k002 + s003*k002 + s002*k003,
 s000*k000 + s002*k000 + s003*k000 + s000*k001 + s001*k001 + s002*k002 + s000*k003 + k000,
 s001*k000 + s003*k000 + s001*k001 + s002*k001 + s000*k002 + s003*k002 + s001*k003 + k001,
 s000*k000 + s002*k000 + s000*k001 + s002*k001 + s003*k001 + s000*k002 + s001*k002 + s002*k0
 s001*k000 + s002*k000 + s000*k001 + s003*k001 + s001*k002 + s003*k003 + k003,
 s000*k000 + s001*k000 + s003*k000 + s001*k001 + s000*k002 + s002*k002 + s000*k003 + s000,
 s002*k000 + s000*k001 + s001*k001 + s003*k001 + s001*k002 + s000*k003 + s002*k003 + s001,
 s000*k000 + s001*k000 + s002*k000 + s002*k001 + s000*k002 + s001*k002 + s003*k002 + s001*k0
 s001*k000 + s000*k001 + s002*k001 + s000*k002 + s001*k003 + s003*k003 + s003,
 s002*k000 + s001*k001 + s000*k002 + s003*k003 + 1)
```

mix_columns(*d*)

Perform the MixColumns operation on d.

INPUT:

- d - state array or something coercible to a state array

EXAMPLES:

```
sage: sr = mq.SR(10, 4, 4, 4)
sage: E = sr.state_array() + 1; E
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: sr.mix_columns(E)
[    a a + 1     1     1]
[    1     a a + 1     1]
[    1     1     a a + 1]
[a + 1     1     1     a]
```

new_generator(***kwds*)

Return a new SR instance equal to this instance except for the parameters passed explicitly to this function.

INPUT:

- **kwds - see the SR constructor for accepted parameters

EXAMPLE:

```
sage: sr = mq.SR(2,1,1,4); sr
SR(2,1,1,4)
sage: sr.ring().base_ring()
Finite Field in a of size 2^4
```

```
sage: sr2 = sr.new_generator(gf2=True); sr2
SR(2,1,1,4)
sage: sr2.ring().base_ring()
Finite Field of size 2
sage: sr3 = sr2.new_generator(correct_only=True)
sage: len(sr2.inversion_polynomials_single_sbox())
20
sage: len(sr3.inversion_polynomials_single_sbox())
19
```

polynomial_system(*P=None, K=None, C=None*)

Return a polynomial system for this small scale AES variant for a given plaintext-key pair.

If neither `P`, `K` nor `C` are provided, a random pair (`P`, `K`) will be generated. If `P` and `C` are provided no `K` needs to be provided.

INPUT:

- `P` - vector, list, or tuple (default: `None`)
- `K` - vector, list, or tuple (default: `None`)
- `C` - vector, list, or tuple (default: `None`)

EXAMPLE:

```
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=True)
sage: P = sr.vector([0, 0, 1, 0])
sage: K = sr.vector([1, 0, 0, 1])
sage: F, s = sr.polynomial_system(P, K)
```

This returns a polynomial system:

```
sage: F
Polynomial System with 36 Polynomials in 20 Variables
```

and a solution:

```
sage: s # random -- maybe we need a better doctest here?
{k000: 1, k001: 0, k003: 1, k002: 0}
```

This solution is not the only solution that we can learn from the Groebner basis of the system.

```
sage: F.groebner_basis()[-3:]
[k000 + 1, k001, k003 + 1]
```

In particular we have two solutions:

```
sage: len(F.ideal().variety())
2
```

In the following example we provide `C` explicitly:

```
sage: C = sr(P,K)
sage: F,s = sr.polynomial_system(P=P, C=C)
sage: F
Polynomial System with 36 Polynomials in 20 Variables
```

Alternatively, we can use symbols for the `P` and `C`. First, we have to create a polynomial ring:

```
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=True)
sage: R = sr.R
sage: vn = sr.varstrs("P",0,1,4) + R.variable_names() + sr.varstrs("C",0,1,4)
sage: R = BooleanPolynomialRing(len(vn),vn)
sage: sr.R = R
```

Now, we can construct the purely symbolic equation system:

```
sage: C = sr.vars("C",0); C
(C000, C001, C002, C003)
sage: P = sr.vars("P",0)
sage: F,s = sr.polynomial_system(P=P,C=C)
sage: [(k,v) for k,v in sorted(s.iteritems())] # this can be ignored
[(k003, 1), (k002, 1), (k001, 0), (k000, 0)]
sage: F
Polynomial System with 36 Polynomials in 28 Variables
sage: F.round(0)
(P000 + w100 + k000, P001 + w101 + k001, P002 + w102 + k002, P003 + w103 + k003)
sage: F.round(-2)
(k100 + x100 + x102 + x103 + C000, k101 + x100 + x101 + x103 + C001 + 1, ...)
```

random_element(*elem_type='vector'*)

Return a random element for self.

INPUT:

- `elem_type` - either 'vector' or 'state array' (default: `'vector'`)

EXAMPLE:

```
sage: sr = mq.SR()
sage: sr.random_element()
[  a^3 + a + 1]
[      a^3 + 1]
[a^3 + a^2 + 1]
[a^3 + a^2 + a]
sage: sr.random_element('state_array')
[a + 1]
```

random_state_array()

Return a random element in `MatrixSpace(self.base_ring(), self.r, self.c)`.

EXAMPLE:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: sr.random_state_array()
[a^3 + a + 1       a + 1]
[      a + 1         a^2]
```

random_vector()

Return a random vector as it might appear in the algebraic expression of self.

EXAMPLE:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: sr.random_vector()
[  a^3 + a + 1]
[      a^3 + 1]
[a^3 + a^2 + 1]
[a^3 + a^2 + a]
[        a + 1]
[      a^2 + 1]
[            a]
[          a^2]
[        a + 1]
[      a^2 + 1]
[            a]
[          a^2]
[          a^2]
[        a + 1]
```

```
[      a^2 + 1]
[            a]
```

Note: ϕ was already applied to the result.

ring(*order=None, reverse_variables=None*)

Construct a ring as a base ring for the polynomial system.

By default, variables are ordered in the reverse of their natural ordering, i.e. the reverse of as they appear.

INPUT:

- `order` - a monomial ordering (default: `None`)
- `reverse_variables` - reverse rounds of variables (default: `True`)

The variable assignment is as follows:

- $k_{i,j,l}$ - subkey round i word j conjugate/bit l
- $s_{i,j,l}$ - subkey inverse round i word j conjugate/bit l
- $w_{i,j,l}$ - inversion input round i word j conjugate/bit l
- $x_{i,j,l}$ - inversion output round i word j conjugate/bit l

Note that the variables are ordered in column major ordering in the state array and that the bits are ordered in little endian ordering.

For example, if $x_{0,1,0}$ is a variable over $\mathbf{F}_2$ for $r = 2$ and $c = 2$ then refers to the *most* significant bit of the entry in the position (1,0) in the state array matrix.

EXAMPLE:

```
sage: sr = mq.SR(2, 1, 1, 4)
sage: P = sr.ring(order='block')
sage: print P.repr_long()
Polynomial Ring
  Base Ring : Finite Field in a of size 2^4
       Size : 36 Variables
   Block  0 : Ordering : degrevlex
              Names    : k200, k201, k202, k203, x200, x201, x202, x203, w200, w201, w202, w
   Block  1 : Ordering : degrevlex
              Names    : k100, k101, k102, k103, x100, x101, x102, x103, w100, w101, w102, w
   Block  2 : Ordering : degrevlex
              Names    : k000, k001, k002, k003
```

round_polynomials(*i, plaintext=None, ciphertext=None*)

Return list of polynomials for a given round i.

If `i == 0` a plaintext must be provided, if `i == n` a ciphertext must be provided.

INPUT:

- `i` - round number
- `plaintext` - optional plaintext (mandatory in first round)
- `ciphertext` - optional ciphertext (mandatory in last round)

OUTPUT: MPolynomialRoundSystem

EXAMPLE:

```
sage: sr = mq.SR(1, 1, 1, 4)
sage: k = sr.base_ring()
sage: p = [k.random_element() for _ in range(sr.r*sr.c)]
sage: sr.round_polynomials(0, plaintext=p)
(w100 + k000 + (a^2 + 1), w101 + k001 + (a), w102 + k002 + (a^2), w103 + k003 + (a + 1))
```

sbox(*inversion_only=False*)

Return an S-Box object for this SR instance.

INPUT:

•`inversion_only` - do not include the $\mathbf{F}_2$ affine map when computing the S-Box (default: `False`)

EXAMPLE:

```
sage: sr = mq.SR(1,2,2,4, allow_zero_inversions=True)
sage: S = sr.sbox(); S
(6, 11, 5, 4, 2, 14, 7, 10, 9, 13, 15, 12, 3, 1, 0, 8)

sage: sr.sub_byte(0)
a^2 + a
sage: sage_eval(str(sr.sub_byte(0)), {'a':2})
6
sage: S(0)
6

sage: sr.sub_byte(1)
a^3 + a + 1
sage: sage_eval(str(sr.sub_byte(1)), {'a':2})
11
sage: S(1)
11

sage: sr = mq.SR(1,2,2,8, allow_zero_inversions=True)
sage: S = sr.sbox(); S
(99, 124, 119, 123, 242, 107, 111, 197, 48, 1, 103, 43,
254, 215, 171, 118, 202, 130, 201, 125, 250, 89, 71, 240,
173, 212, 162, 175, 156, 164, 114, 192, 183, 253, 147, 38,
54, 63, 247, 204, 52, 165, 229, 241, 113, 216, 49, 21, 4,
199, 35, 195, 24, 150, 5, 154, 7, 18, 128, 226, 235, 39,
178, 117, 9, 131, 44, 26, 27, 110, 90, 160, 82, 59, 214,
179, 41, 227, 47, 132, 83, 209, 0, 237, 32, 252, 177, 91,
106, 203, 190, 57, 74, 76, 88, 207, 208, 239, 170, 251,
67, 77, 51, 133, 69, 249, 2, 127, 80, 60, 159, 168, 81,
163, 64, 143, 146, 157, 56, 245, 188, 182, 218, 33, 16,
255, 243, 210, 205, 12, 19, 236, 95, 151, 68, 23, 196,
167, 126, 61, 100, 93, 25, 115, 96, 129, 79, 220, 34, 42,
144, 136, 70, 238, 184, 20, 222, 94, 11, 219, 224, 50, 58,
10, 73, 6, 36, 92, 194, 211, 172, 98, 145, 149, 228, 121,
231, 200, 55, 109, 141, 213, 78, 169, 108, 86, 244, 234,
101, 122, 174, 8, 186, 120, 37, 46, 28, 166, 180, 198,
232, 221, 116, 31, 75, 189, 139, 138, 112, 62, 181, 102,
72, 3, 246, 14, 97, 53, 87, 185, 134, 193, 29, 158, 225,
248, 152, 17, 105, 217, 142, 148, 155, 30, 135, 233, 206,
85, 40, 223, 140, 161, 137, 13, 191, 230, 66, 104, 65,
153, 45, 15, 176, 84, 187, 22)

sage: sr.sub_byte(0)
a^6 + a^5 + a + 1

sage: sage_eval(str(sr.sub_byte(0)), {'a':2})
99
sage: S(0)
99

sage: sr.sub_byte(1)
a^6 + a^5 + a^4 + a^3 + a^2

sage: sage_eval(str(sr.sub_byte(1)), {'a':2})
124
```

```
sage: S(1)
124

sage: sr = mq.SR(1,2,2,4, allow_zero_inversions=True)
sage: S = sr.sbox(inversion_only=True); S
(0, 1, 9, 14, 13, 11, 7, 6, 15, 2, 12, 5, 10, 4, 3, 8)

sage: S(0)
0
sage: S(1)
1

sage: S(sr.k.gen())
a^3 + 1
```

sbox_constant()

Return the S-Box constant which is added after $L(x^{-1})$ was performed. That is `0x63` if `e == 8` or `0x6` if `e == 4`.

EXAMPLE:

```
sage: sr = mq.SR(10, 1, 1, 8)
sage: sr.sbox_constant()
a^6 + a^5 + a + 1
```

shift_rows(*d*)

Perform the `ShiftRows` operation on `d`.

INPUT:

- `d` - state array or something coercible to a state array

EXAMPLES:

```
sage: sr = mq.SR(10, 4, 4, 4)
sage: E = sr.state_array() + 1; E
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]

sage: sr.shift_rows(E)
[1 0 0 0]
[1 0 0 0]
[1 0 0 0]
[1 0 0 0]
```

state_array(*d=None*)

Convert the parameter to a state array.

INPUT:

- `d` - a matrix, a list, or a tuple (default: `None`)

EXAMPLES:

```
sage: sr = mq.SR(2, 2, 2, 4)
sage: k = sr.base_ring()
sage: e1 = [k.fetch_int(e) for e in range(2*2)]; e1
[0, 1, a, a + 1]
sage: e2 = sr.phi( Matrix(k, 2*2, 1, e1) )
sage: sr.state_array(e1) # note the column major ordering
[    0     a]
```

```
[    1 a + 1]
sage: sr.state_array(e2)
[    0     a]
[    1 a + 1]

sage: sr.state_array()
[0 0]
[0 0]
```

sub_byte(*b*)

Perform SubByte on a single byte/halfbyte b.

A ZeroDivision exception is raised if an attempt is made to perform an inversion on the zero element. This can be disabled by passing allow_zero_inversion=True to the constructor. A zero inversion can result in an inconsistent equation system.

INPUT:

- b - an element in self.base_ring()

EXAMPLE:

The S-Box table for $\mathbf{F}_{2^4}$:

```
sage: sr = mq.SR(1, 1, 1, 4, allow_zero_inversions=True)
sage: for e in sr.base_ring():
...     print '% 20s % 20s'%(e, sr.sub_byte(e))
                   0                a^2 + a
                   a                a^2 + 1
                 a^2                    a
                 a^3                a^3 + 1
               a + 1                    a^2
             a^2 + a            a^2 + a + 1
           a^3 + a^2                  a + 1
         a^3 + a + 1              a^3 + a^2
             a^2 + 1          a^3 + a^2 + a
             a^3 + a      a^3 + a^2 + a + 1
         a^2 + a + 1                a^3 + a
       a^3 + a^2 + a                    0
a^3 + a^2 + a + 1                   a^3
       a^3 + a^2 + 1                    1
             a^3 + 1          a^3 + a^2 + 1
                   1            a^3 + a + 1
```

sub_bytes(*d*)

Perform the non-linear transform on d.

INPUT:

- d - state array or something coercible to a state array

EXAMPLE:

```
sage: sr = mq.SR(2, 1, 2, 8, gf2=True)
sage: k = sr.base_ring()
sage: A = Matrix(k, 1, 2 , [k(1), k.gen()])
sage: sr.sub_bytes(A)
[  a^6 + a^5 + a^4 + a^3 + a^2 a^6 + a^5 + a^4 + a^2 + a + 1]
```

varformatstr(*name, n=None, rc=None, e=None*)

Return a format string which is understood by print et al.

If a numerical value is omitted, the default value of self is used. The numerical values (n, rc, e) are used to determine the width of the respective fields in the format string.

INPUT:

- name - name of the variable
- n - number of rounds (default: None)
- rc - number of rows * number of cols (default: None)
- e - exponent of base field (default: None)

EXAMPLE:

```
sage: sr = mq.SR(1, 2, 2, 4)
sage: sr.varformatstr('x')
'x%01d%01d%01d'
sage: sr.varformatstr('x', n=1000)
'x%03d%03d%03d'
```

variable_dict()

Return a dictionary to access variables in self.R by their names.

EXAMPLE:

```
sage: sr = mq.SR(1,1,1,4)
sage: sr.variable_dict()
{'x101': x101, 'x100': x100, 'x103': x103, 'x102': x102,
 's002': s002, 'w100': w100, 'w101': w101, 'w102': w102,
 'w103': w103, 'k100': k100, 'k101': k101, 'k102': k102,
 'k103': k103, 's003': s003, 's001': s001, 'k002': k002,
 'k001': k001, 'k000': k000, 'k003': k003, 's000': s000}

sage: sr = mq.SR(1,1,1,4,gf2=True)
sage: sr.variable_dict()
{'x101': x101, 'x100': x100, 'x103': x103, 'x102': x102,
 's002': s002, 'w100': w100, 'w101': w101, 'w102': w102,
 'w103': w103, 'k100': k100, 'k101': k101, 'k102': k102,
 'k103': k103, 's003': s003, 's001': s001, 'k002': k002,
 'k001': k001, 'k000': k000, 'k003': k003, 's000': s000}
```

vars(*name, nr, rc=None, e=None*)

Return a list of variables in self.

INPUT:

- name - variable name
- nr - number of round to create variable strings for
- rc - number of rounds * number of columns in the state array (default: None)
- e - exponent of base field (default: None)

EXAMPLE:

```
sage: sr = mq.SR(10, 1, 2, 4)
sage: sr.vars('x', 2)
(x200, x201, x202, x203, x210, x211, x212, x213)
```

varstr(*name, nr, rc, e*)

Return a string representing a variable for the small scale AES subject to the given constraints.

INPUT:

- name - variable name
- nr - number of round to create variable strings for
- rc - row*column index in state array
- e - exponent of base field

EXAMPLE:

```
sage: sr = mq.SR(10, 1, 2, 4)
sage: sr.varstr('x', 2, 1, 1)
'x211'
```

varstrs (*name, nr, rc=None, e=None*)

Return a list of strings representing variables in `self`.

INPUT:

- `name` - variable name
- `nr` - number of round to create variable strings for
- `rc` - number of rows * number of columns in the state array (default: `None`)
- `e` - exponent of base field (default: `None`)

EXAMPLE:

```
sage: sr = mq.SR(10, 1, 2, 4)
sage: sr.varstrs('x', 2)
('x200', 'x201', 'x202', 'x203', 'x210', 'x211', 'x212', 'x213')
```

class SR_gf2 (*n=1, r=1, c=1, e=4, star=False, **kwargs*)

antiphi (*l*)

The operation ϕ^{-1} from [MR02] or the inverse of `self.phi`.

INPUT:

- `l` - a vector in the sense of `self.is_vector`

EXAMPLE:

```
sage: sr = mq.SR(gf2=True)
sage: A = sr.random_state_array()
sage: A
[a^3 + a + 1]
sage: sr.antiphi(sr.phi(A)) == A
True
```

field_polynomials (*name, i, l=None*)

Return list of field polynomials for a given round `i` and name `name`.

INPUT:

- `name` - variable name
- `i` - round number
- `l` - length of variable list (default: `None` = r*c)

EXAMPLE:

```
sage: sr = mq.SR(3, 1, 1, 8, gf2=True)
sage: sr.field_polynomials('x', 2)
[x200^2 + x200, x201^2 + x201,
 x202^2 + x202, x203^2 + x203,
 x204^2 + x204, x205^2 + x205,
 x206^2 + x206, x207^2 + x207]

sage: sr = mq.SR(3, 1, 1, 8, gf2=True, polybori=True)
sage: sr.field_polynomials('x', 2)
[]
```

inversion_polynomials (*xi, wi, length*)

Return polynomials to represent the inversion in the AES S-Box.

INPUT:

- `xi` - output variables
- `wi` - input variables
- `length` - length of both lists

EXAMPLE:

```
sage: sr = mq.SR(1, 1, 1, 8, gf2=True)
sage: xi = sr.vars('x', 1)
sage: wi = sr.vars('w', 1)
sage: sr.inversion_polynomials(xi, wi, len(xi))[:3]
[x100*w100 + x102*w100 + x103*w100 + x107*w100 + x101*w101
+ x102*w101 + x106*w101 + x100*w102 + x101*w102 +
x105*w102 + x100*w103 + x104*w103 + x103*w104 + x102*w105
+ x101*w106 + x100*w107, x101*w100 + x103*w100 + x104*w100
+ x100*w101 + x102*w101 + x103*w101 + x107*w101 +
x101*w102 + x102*w102 + x106*w102 + x100*w103 + x101*w103
+ x105*w103 + x100*w104 + x104*w104 + x103*w105 +
x102*w106 + x101*w107, x102*w100 + x104*w100 + x105*w100 +
x101*w101 + x103*w101 + x104*w101 + x100*w102 + x102*w102
+ x103*w102 + x107*w102 + x101*w103 + x102*w103 +
x106*w103 + x100*w104 + x101*w104 + x105*w104 + x100*w105
+ x104*w105 + x103*w106 + x102*w107]
```

inversion_polynomials_single_sbox(*x=None, w=None, biaffine_only=None, correct_only=None*)

Return inversion polynomials of a single S-Box.

INPUT:

- `xi` - output variables
- `wi` - input variables
- `length` - length of both lists

EXAMPLES:

```
sage: sr = mq.SR(1, 1, 1, 8, gf2=True)
sage: len(sr.inversion_polynomials_single_sbox())
24
sage: len(sr.inversion_polynomials_single_sbox(correct_only=True))
23
sage: len(sr.inversion_polynomials_single_sbox(biaffine_only=False))
40
sage: len(sr.inversion_polynomials_single_sbox(biaffine_only=False, correct_only=True))
39


sage: sr = mq.SR(1, 1, 1, 8, gf2=True)
sage: l0 = sr.inversion_polynomials_single_sbox(); len(l0)
24
sage: l1 = sr.inversion_polynomials_single_sbox(correct_only=True); len(l1)
23
sage: l2 = sr.inversion_polynomials_single_sbox(biaffine_only=False); len(l2)
40
sage: l3 = sr.inversion_polynomials_single_sbox(biaffine_only=False, correct_only=True); len
39

sage: set(l0) == set(sr._inversion_polynomials_single_sbox())
True
sage: set(l1) == set(sr._inversion_polynomials_single_sbox(correct_only=True))
True
sage: set(l2) == set(sr._inversion_polynomials_single_sbox(biaffine_only=False))
True
```

```
sage: set(l3) == set(sr._inversion_polynomials_single_sbox(biaffine_only=False, correct_only
True

sage: sr = mq.SR(1, 1, 1, 4, gf2=True)
sage: l0 = sr.inversion_polynomials_single_sbox(); len(l0)
12
sage: l1 = sr.inversion_polynomials_single_sbox(correct_only=True); len(l1)
11
sage: l2 = sr.inversion_polynomials_single_sbox(biaffine_only=False); len(l2)
20
sage: l3 = sr.inversion_polynomials_single_sbox(biaffine_only=False, correct_only=True); len
19

sage: set(l0) == set(sr._inversion_polynomials_single_sbox())
True
sage: set(l1) == set(sr._inversion_polynomials_single_sbox(correct_only=True))
True
sage: set(l2) == set(sr._inversion_polynomials_single_sbox(biaffine_only=False))
True
sage: set(l3) == set(sr._inversion_polynomials_single_sbox(biaffine_only=False, correct_only
True
```

is_vector(*d*)

Return `True` if the given matrix satisfies the conditions for a vector as it appears in the algebraic expression of `self`.

INPUT:

- `d` - matrix

EXAMPLE:

```
sage: sr = mq.SR(gf2=True)
sage: sr
SR(1,1,1,4)
sage: k = sr.base_ring()
sage: A = Matrix(k, 1, 1, [k.gen()])
sage: B = sr.vector(A)
sage: sr.is_vector(A)
False
sage: sr.is_vector(B)
True
```

lin_matrix(*length=None*)

Return the `Lin` matrix.

If no `length` is provided, the standard state space size is used. The key schedule calls this method with an explicit length argument because only `self.r` S-Box applications are performed in the key schedule.

INPUT:

- `length` - length of state space (default: `None`)

EXAMPLE:

```
sage: sr = mq.SR(1, 1, 1, 4, gf2=True)
sage: sr.lin_matrix()
[1 0 1 1]
[1 1 0 1]
[1 1 1 0]
[0 1 1 1]
```

mix_columns_matrix()

Return the `MixColumns` matrix.

EXAMPLE:

```
sage: sr = mq.SR(1, 2, 2, 4, gf2=True)
sage: s = sr.random_state_array()
sage: r1 = sr.mix_columns(s)
sage: r2 = sr.state_array(sr.mix_columns_matrix() * sr.vector(s))
sage: r1 == r2
True
```

phi (*l, diffusion_matrix=False*)

The operation ϕ from [MR02]

Given a list/matrix of elements in $\mathbf{F}_{2^e}$, return a matching list/matrix of elements in $\mathbf{F}_2$.

INPUT:

- `l` - element to perform ϕ on.
- `diffusion_matrix` - if `True`, the given matrix `l` is transformed to a matrix which performs the same operation over $\mathbf{F}_2$ as `l` over $\mathbf{F}_{2^n}$ (default: `False`).

EXAMPLE:

```
sage: sr = mq.SR(2, 1, 2, 4, gf2=True)
sage: k = sr.base_ring()
sage: A = matrix(k, 1, 2, [k.gen(), 0] )
sage: sr.phi(A)
[0 0]
[0 0]
[1 0]
[0 0]
```

shift_rows_matrix()

Return the `ShiftRows` matrix.

EXAMPLE:

```
sage: sr = mq.SR(1, 2, 2, 4, gf2=True)
sage: s = sr.random_state_array()
sage: r1 = sr.shift_rows(s)
sage: r2 = sr.state_array( sr.shift_rows_matrix() * sr.vector(s) )
sage: r1 == r2
True
```

vector (*d=None*)

Constructs a vector suitable for the algebraic representation of SR.

INPUT:

- `d` - values for vector (default: `None`)

EXAMPLE:

```
sage: sr = mq.SR(gf2=True)
sage: sr
SR(1,1,1,4)
sage: k = sr.base_ring()
sage: A = Matrix(k, 1, 1, [k.gen()])
sage: sr.vector(A)
[0]
[0]
[1]
[0]
```

class SR_gf2_2 (*n=1, r=1, c=1, e=4, star=False, **kwargs*)

This is an example how to customize the SR constructor.

In this example, we replace the S-Box inversion polynomials by the polynomials generated by the S-Box class.

inversion_polynomials_single_sbox(*x=None, w=None, biaffine_only=None, correct_only=None, groebner=False*)

Return inversion polynomials of a single S-Box.

INPUT:

- `x` - output variables (default: `None`)
- `w` - input variables (default: `None`)
- `biaffine_only` - ignored (always `False`)
- `correct_only` - ignored (always `True`)
- `groebner` - precompute the Groebner basis for this S-Box (default: `False`).

EXAMPLES:

```
sage: from sage.crypto.mq.sr import SR_gf2_2
sage: e = 4
sage: sr = SR_gf2_2(1, 1, 1, e)
sage: P = PolynomialRing(GF(2),['x%d'%i for i in range(e)] + ['w%d'%i for i in range(e)],orc
sage: X,W = P.gens()[:e],P.gens()[e:]
sage: sr.inversion_polynomials_single_sbox(X, W, groebner=True)
[x0 + w0*w1*w2 + w0*w1 + w0*w2 + w0*w3 + w0 + w1 + w2,
 x1 + w0*w1*w3 + w0*w3 + w0 + w1*w3 + w1 + w2*w3,
 x2 + w0*w2*w3 + w0*w2 + w0 + w1*w2 + w1*w3 + w2*w3,
 x3 + w0*w1*w2 + w0 + w1*w2*w3 + w1*w2 + w1*w3 + w1 + w2 + w3]

sage: from sage.crypto.mq.sr import SR_gf2_2
sage: e = 4
sage: sr = SR_gf2_2(1, 1, 1, e)
sage: sr.inversion_polynomials_single_sbox()
[w3*w1 + w3*w0 + w3*x2 + w3*x1 + w3 + w2*w1 + w1 + x3 + x2 + x1,
 w3*w2 + w3*w1 + w3*x3 + w2 + w1 + x3,
 w3*w2 + w3*w1 + w3*x2 + w3 + w2*x3 + x2 + x1,
 w3*w2 + w3*w1 + w3*x3 + w3*x2 + w3*x1 + w3 + w2*x2 + w0 + x3 + x2 + x1 + x0,
 w3*w2 + w3*w1 + w3*x1 + w3*x0 + w2*x1 + w0 + x3 + x0,
 w3*w2 + w3*w1 + w3*w0 + w3*x2 + w3*x1 + w2*w0 + w2*x0 + w0 + x3 + x2 + x1 + x0,
 w3*w2 + w3*x1 + w3 + w2*w0 + w1*w0 + w1 + x3 + x2,
 w3*w2 + w3*w1 + w3*x1 + w1*x3 + x3 + x2 + x1,
 w3*x3 + w3*x2 + w3*x0 + w3 + w1*x2 + w1 + w0 + x2 + x0,
 w3*w2 + w3*w1 + w3*x2 + w3*x1 + w1*x1 + w1 + w0 + x2 + x0,
 w3*w2 + w3*w1 + w3*w0 + w3*x3 + w3*x1 + w2*w0 + w1*x0 + x3 + x2,
 w3*w2 + w3*w1 + w3*x2 + w3*x1 + w3*x0 + w3 + w1 + w0*x3 + x3 + x2,
 w3*w2 + w3*w1 + w3*w0 + w3*x3 + w3 + w2*w0 + w1 + w0*x2 + x3 + x2,
 w3*w0 + w3*x2 + w2*w0 + w0*x1 + w0 + x3 + x1 + x0,
 w3*w0 + w3*x3 + w3*x0 + w2*w0 + w1 + w0*x0 + w0 + x3 + x2,
 w3*w2 + w3 + w1 + x3*x2 + x3 + x1,
 w3*w2 + w3*x3 + w1 + x3*x1 + x3 + x2,
 w3*w2 + w3*w0 + w3*x3 + w3*x2 + w3*x1 + w0 + x3*x0 + x1 + x0,
 w3*w2 + w3*w1 + w3*w0 + w3*x3 + w1 + w0 + x2*x1 + x2 + x0,
 w3*w2 + w2*w0 + w1 + x3 + x2*x0,
 w3*x3 + w3*x1 + w2*w0 + w1 + x3 + x2 + x1*x0 + x1]
```

class SR_gf2n(*n=1, r=1, c=1, e=4, star=False, **kwargs*)

Small Scale Variants of the AES polynomial system constructor over $\mathbf{F}_{2^n}$.

antiphi(*l*)

The operation ϕ^{-1} from [MR02] or the inverse of `self.phi`.

INPUT:

- `l` - a vector in the sense of `self.is_vector`

EXAMPLE:

```
sage: sr = mq.SR()
sage: A = sr.random_state_array()
sage: A
[a^3 + a + 1]
sage: sr.antiphi(sr.phi(A)) == A
True
```

field_polynomials(*name, i, l=None*)

Return list of conjugacy polynomials for a given round `i` and name `name`.

INPUT:

- `name` - variable name
- `i` - round number
- `l` - r*c (default: `None`)

EXAMPLE:

```
sage: sr = mq.SR(3, 1, 1, 8)
sage: sr.field_polynomials('x', 2)
[x200^2 + x201,
x201^2 + x202,
x202^2 + x203,
x203^2 + x204,
x204^2 + x205,
x205^2 + x206,
x206^2 + x207,
x207^2 + x200]
```

inversion_polynomials(*xi, wi, length*)

Return polynomials to represent the inversion in the AES S-Box.

INPUT:

- `xi` - output variables
- `wi` - input variables
- `length` - length of both lists

EXAMPLE:

```
sage: sr = mq.SR(1, 1, 1, 8)
sage: R = sr.ring()
sage: xi = Matrix(R, 8, 1, sr.vars('x', 1))
sage: wi = Matrix(R, 8, 1, sr.vars('w', 1))
sage: sr.inversion_polynomials(xi, wi, 8)
[x100*w100 + 1,
x101*w101 + 1,
x102*w102 + 1,
x103*w103 + 1,
x104*w104 + 1,
x105*w105 + 1,
x106*w106 + 1,
x107*w107 + 1]
```

is_vector(*d*)

Return `True` if `d` can be used as a vector for `self`.

EXAMPLE:

```
sage: sr = mq.SR()
sage: sr
SR(1,1,1,4)
sage: k = sr.base_ring()
```

```
sage: A = Matrix(k, 1, 1, [k.gen()])
sage: B = sr.vector(A)
sage: sr.is_vector(A)
False
sage: sr.is_vector(B)
True
```

lin_matrix(*length=None*)

Return the `Lin` matrix.

If no `length` is provided, the standard state space size is used. The key schedule calls this method with an explicit length argument because only `self.r` S-Box applications are performed in the key schedule.

INPUT:

- `length` - length of state space (default: `None`)

EXAMPLE:

```
sage: sr = mq.SR(1, 1, 1, 4)
sage: sr.lin_matrix()
[          a^2 + 1                 1         a^3 + a^2            a^2 + 1]
[               a                  a                 1 a^3 + a^2 + a + 1]
[          a^3 + a              a^2               a^2                  1]
[               1              a^3             a + 1              a + 1]
```

mix_columns_matrix()

Return the `MixColumns` matrix.

EXAMPLE:

```
sage: sr = mq.SR(1, 2, 2, 4)
sage: s = sr.random_state_array()
sage: r1 = sr.mix_columns(s)
sage: r2 = sr.state_array(sr.mix_columns_matrix() * sr.vector(s))
sage: r1 == r2
True
```

phi(*l*)

The operation ϕ from [MR02]

Projects state arrays to their algebraic representation.

INPUT:

- `l` - element to perform ϕ on.

EXAMPLE:

```
sage: sr = mq.SR(2, 1, 2, 4)
sage: k = sr.base_ring()
sage: A = matrix(k, 1, 2, [k.gen(), 0] )
sage: sr.phi(A)
[      a       0]
[    a^2       0]
[  a + 1       0]
[a^2 + 1       0]
```

shift_rows_matrix()

Return the `ShiftRows` matrix.

EXAMPLE:

```
sage: sr = mq.SR(1, 2, 2, 4)
sage: s = sr.random_state_array()
sage: r1 = sr.shift_rows(s)
```

```
sage: r2 = sr.state_array( sr.shift_rows_matrix() * sr.vector(s) )
sage: r1 == r2
True
```

vector(*d=None*)

Constructs a vector suitable for the algebraic representation of SR, i.e. BES.

INPUT:

- d - values for vector, must be understood by `self.phi` (default:None)

EXAMPLE:

```
sage: sr = mq.SR()
sage: sr
SR(1,1,1,4)
sage: k = sr.base_ring()
sage: A = Matrix(k, 1, 1, [k.gen()])
sage: sr.vector(A)
[      a]
[    a^2]
[  a + 1]
[a^2 + 1]
```

test_consistency(*max_n=2, **kwargs*)

Test all combinations of `r`, `c`, `e` and `n` in `(1, 2)` for consistency of random encryptions and their polynomial systems. $\mathbf{F}_2$ and $\mathbf{F}_{2^e}$ systems are tested. This test takes a while.

INPUT:

- max_n - maximal number of rounds to consider (default: 2)
- kwargs - are passed to the SR constructor

TESTS:

```
sage: from sage.crypto.mq.sr import test_consistency
sage: test_consistency(1) # long time -- calling w/ max_n = 2 requires a LOT of RAM (>> 2GB, evi
True
```

The above doctest used to fail on a machine with "only" 2GB RAM. Using `max_n = 1` appears to be a more reasonable memory usage.

16.9 Multivariate Polynomial Systems.

We call a finite set of multivariate polynomials an `MPolynomialSystem`.

In many other computer algebra systems (cf. Singular) this class would be called `Ideal` but an ideal is a very distinct object from its generators and thus this is not an ideal in Sage.

The idea of polynomial systems is specialized to systems which consist of several "rounds" in Sage. These kind of polynomial systems arise naturally in algebraic cryptanalysis of symmetric cryptographic primitives. The most prominent examples of these systems are: the small scale variants of the AES [CMR05] (cf. `mq.SR`) and Flurry/Curry [BPW06].

AUTHORS:

- Martin Albrecht (2007ff): initial version
- Martin Albrecht (2009): refactoring, clean-up, new functions

EXAMPLES:

As an example consider a small scale variant of the AES:

```
sage: sr = mq.SR(2,1,2,4,gf2=True,polybori=True)
sage: sr
SR(2,1,2,4)
```

We can construct a polynomial system for a random plaintext-ciphertext pair and study it:

```
sage: F,s = sr.polynomial_system()
sage: F
Polynomial System with 112 Polynomials in 64 Variables

sage: r2 = F.round(2); r2
(w200 + k100 + x100 + x102 + x103,
 w201 + k101 + x100 + x101 + x103 + 1,
 w202 + k102 + x100 + x101 + x102 + 1,
 w203 + k103 + x101 + x102 + x103,
 w210 + k110 + x110 + x112 + x113,
 w211 + k111 + x110 + x111 + x113 + 1,
 w212 + k112 + x110 + x111 + x112 + 1,
 w213 + k113 + x111 + x112 + x113,
 w100*x100 + w100*x103 + w101*x102 + w102*x101 + w103*x100,
 w100*x100 + w100*x101 + w101*x100 + w101*x103 + w102*x102 + w103*x101,
 w100*x101 + w100*x102 + w101*x100 + w101*x101 + w102*x100 + w102*x103 + w103*x102,
 w100*x100 + w100*x101 + w100*x103 + w101*x101 + w102*x100 + w102*x102 + w103*x100 + x100,
 w100*x102 + w101*x100 + w101*x101 + w101*x103 + w102*x101 + w103*x100 + w103*x102 + x101,
 w100*x100 + w100*x101 + w100*x102 + w101*x102 + w102*x100 + w102*x101 + w102*x103 + w103*x101 + x102
 w100*x101 + w101*x100 + w101*x102 + w102*x100 + w103*x101 + w103*x103 + x103,
 w100*x100 + w100*x102 + w100*x103 + w101*x100 + w101*x101 + w102*x102 + w103*x100 + w100,
 w100*x101 + w100*x103 + w101*x101 + w101*x102 + w102*x100 + w102*x103 + w103*x101 + w101,
 w100*x100 + w100*x102 + w101*x100 + w101*x102 + w101*x103 + w102*x100 + w102*x101 + w103*x102 + w102
 w100*x101 + w100*x102 + w101*x100 + w101*x103 + w102*x101 + w103*x103 + w103,
 w100*x102 + w101*x101 + w102*x100 + w103*x103 + 1,
 w110*x110 + w110*x113 + w111*x112 + w112*x111 + w113*x110,
 w110*x110 + w110*x111 + w111*x110 + w111*x113 + w112*x112 + w113*x111,
 w110*x111 + w110*x112 + w111*x110 + w111*x111 + w112*x110 + w112*x113 + w113*x112,
 w110*x110 + w110*x111 + w110*x113 + w111*x111 + w112*x110 + w112*x112 + w113*x110 + x110,
 w110*x112 + w111*x110 + w111*x111 + w111*x113 + w112*x111 + w113*x110 + w113*x112 + x111,
 w110*x110 + w110*x111 + w110*x112 + w111*x112 + w112*x110 + w112*x111 + w112*x113 + w113*x111 + x112
 w110*x111 + w111*x110 + w111*x112 + w112*x110 + w113*x111 + w113*x113 + x113,
 w110*x110 + w110*x112 + w110*x113 + w111*x110 + w111*x111 + w112*x112 + w113*x110 + w110,
 w110*x111 + w110*x113 + w111*x111 + w111*x112 + w112*x110 + w112*x113 + w113*x111 + w111,
 w110*x110 + w110*x112 + w111*x110 + w111*x112 + w111*x113 + w112*x110 + w112*x111 + w113*x112 + w112
 w110*x111 + w110*x112 + w111*x110 + w111*x113 + w112*x111 + w113*x113 + w113,
 w110*x112 + w111*x111 + w112*x110 + w113*x113 + 1)

 sage: type(r2)
 <class 'sage.crypto.mq.mpolynomialsystem.MPolynomialRoundSystem_generic'>
```

As an example, we separate the system in independent subsystems or compute the coefficient matrix:

```
sage: C = mq.MPolynomialSystem(r2).connected_components(); C
[Polynomial System with 16 Polynomials in 16 Variables,
 Polynomial System with 16 Polynomials in 16 Variables]

sage: C[0].groebner_basis()
```

```
[x111*x110 + w113*x110 + w113*x112 + w113*x113 + w113*w111 + w113*w112 + x111 + x113 + w110 + w111 +
 x112*x110 + w113*x112 + w113*x113 + w113*w110 + w113*w112 + x110 + x112 + 1,
 x112*x111 + w113*w110 + x110 + x111 + x113 + w111 + w113,
 x113*x110 + w113*x110 + w113*x112 + w113*w110 + w113*w111 + w113*w112 + x110 + x111 + x113 + w110 +
 x113*x111 + w113*x110 + w113*x112 + w113*x113 + w113*w112 + x111 + x112 + x113 + w111 + w112,
 x113*x112 + w113*x112 + w113*x113 + w113*w110 + x111 + w110 + w112 + 1,
 w110*x110 + w113*x110 + w113*w110 + w113*w111 + x110 + x113 + w111 + w112 + 1,
 w110*x111 + w113*x112 + w113*w110 + w113*w111 + x110 + x112 + x113 + w113,
 w110*x112 + w113*x110 + w113*x112 + w113*x113 + x112 + x113 + w110 + w111 + w112,
 w110*x113 + w113*x110 + w113*x113 + x111 + x113 + w111 + w113 + 1,
 w111*x110 + w113*x110 + w113*x112 + x110 + x111 + w110 + w113 + 1,
 w111*x111 + w113*x110 + w113*x113 + w113*w110 + w113*w111 + x110 + x111 + x112 + x113,
 w111*x112 + w113*x110 + w113*x112 + w113*w110 + w113*w111 + x110 + x111 + x112 + w110 + w112,
 w111*x113 + w113*x113 + w113*w110 + w113*w111 + x111 + x112 + w110 + w111 + w112 + 1,
 w111*w110 + w113*x110 + w113*x112 + w113*x113 + w113*w111 + w113*w112 + x110 + x111 + x112 + x113 +
 w112*x110 + w113*x112 + w113*x113 + w113*w110 + w113*w111 + x110 + x111 + w110 + w111 + w112 + 1,
 w112*x111 + w113*x112 + w113*x113 + x112 + w110 + w113,
 w112*x112 + w113*x113 + x110 + x111 + w110 + w111 + 1,
 w112*x113 + w113*x110 + w113*x112 + w113*x113 + w113*w110 + w113*w111 + x111 + x112 + x113 + w110,
 w112*w110 + w113*x112 + w113*x113 + w113*w110 + w113*w112 + x110 + x111 + x112 + w111 + 1,
 w112*w111 + w113*x110 + w113*x112 + w113*w110 + w113*w111 + w113*w112 + x110 + w113 + 1,
 w113*x111 + w113*w110 + w113*w111 + x111 + w110 + w111,
 w210 + k110 + x110 + x112 + x113,
 w211 + k111 + x110 + x111 + x113 + 1,
 w212 + k112 + x110 + x111 + x112 + 1,
 w213 + k113 + x111 + x112 + x113]

sage: A,v = mq.MPolynomialSystem(r2).coefficient_matrix()
sage: A.rank()
32
```

Using these building blocks we can implement a simple XL algorithm easily:

```
sage: sr = mq.SR(1,1,1,4,gf2=True,polybori=True,order='lex')
sage: F,s = sr.polynomial_system()

sage: monomials = [a*b for a in F.variables() for b in F.variables() if a<b]
sage: len(monomials)
190
sage: F2 = mq.MPolynomialSystem(map(mul, cartesian_product_iterator((monomials, F))))
sage: A,v = F2.coefficient_matrix(sparse=False)
sage: A.echelonize()
sage: A
6840 x 4474 dense matrix over Finite Field of size 2
sage: A.rank()
4056
sage: A[4055]*v
(k002*k003)
```

The last output tells us that `k002` and `k003` can't both be 1.

TEST:

```
sage: P.<x,y> = PolynomialRing(QQ)
sage: I = [[x^2 + y^2], [x^2 - y^2]]
sage: F = mq.MPolynomialSystem(P,I)
sage: loads(dumps(F)) == F
True
```

REFERENCES:

MPolynomialRoundSystem(*R, gens*)

Construct an object representing the equations of a single round e.g. of a block cipher or however a "round" is defined.

INPUT:

- `R` - the base ring
- `gens` - list (default: `[]`)

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(GF(2),3)
sage: mq.MPolynomialRoundSystem(P,[x*y +1, z + 1])
(x*y + 1, z + 1)
```

class **MPolynomialRoundSystem_generic**(*R, gens*)

gens()

Return list of polynomials in in the system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: R1 = F.round(1)
sage: l = R1.gens()
sage: l[0]
k000^2 + k000
```

monomials()

Return an unordered list of monomials appearing in polynomials in this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: R1 = F.round(1)
sage: sorted(R1.monomials())
[k003, k002, k001, k000, k003^2, k002^2, k001^2, k000^2]
```

ngens()

Return number of polynomials in the system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: R0 = F.round(0)
sage: R0.ngens()
4
```

ring()

Return the polynomial ring the members of this system live in.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: R0 = F.round(0)
sage: print R0.ring().repr_long()
Polynomial Ring
```

```
Base Ring : Finite Field of size 2
     Size : 20 Variables
 Block  0 : Ordering : degrevlex
            Names    : k100, k101, k102, k103, x100, x101, x102, x103, w100, w101, w102, w1
 Block  1 : Ordering : degrevlex
            Names    : k000, k001, k002, k003
```

subs(**args, **kwargs*)

Substitute variables for every polynomial in this system and return a new system. See `MPolynomial.subs` for calling convention.

INPUT:

- `args` - arguments to be passed to `MPolynomial.subs`
- `kwargs` - keyword arguments to be passed to `MPolynomial.subs`

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: R1 = F.round(1)
sage: R1 = R1.subs(s) # the solution
sage: R1
(0, 0, 0, 0)
```

variables()

Return an unordered list of variables appearing in polynomials in this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: R1 = F.round(1)
sage: sorted(R1.variables())
[k003, k002, k001, k000]
```

MPolynomialSystem(*arg1, arg2=None*)

Construct a new polynomial system object.

INPUT:

- `arg1` - a multivariate polynomial ring or an ideal
- `arg2` - an iterable object of rounds, preferable `MPolynomialRoundSystem`, or polynomials (default:`None`)

EXAMPLES:

```
sage: P.<a,b,c,d> = PolynomialRing(GF(127),4)
sage: I = sage.rings.ideal.Katsura(P)
```

If a list of MPolynomialRoundSystems is provided, those form the rounds:

```
sage: mq.MPolynomialSystem(I.ring(), [mq.MPolynomialRoundSystem(I.ring(),I.gens())])
Polynomial System with 4 Polynomials in 4 Variables
```

If an ideal is provided, the generators are used:

```
sage: mq.MPolynomialSystem(I)
Polynomial System with 4 Polynomials in 4 Variables
```

If a list of polynomials is provided, the system has only one round:

```
sage: mq.MPolynomialSystem(I.ring(), I.gens())
Polynomial System with 4 Polynomials in 4 Variables
```

class MPolynomialSystem_generic(*R, rounds*)

coefficient_matrix(*sparse=True*)

Return tuple `(A,v)` where `A` is the coefficient matrix of this system and `v` the matching monomial vector. Thus value of `A[i,j]` corresponds the coefficient of the monomial `v[j]` in the `i`-th polynomial in this system.

Monomials are order w.r.t. the term ordering of `self.ring()` in reverse order, i.e. such that the smallest entry comes last.

INPUT:

- `sparse` - construct a sparse matrix (default: `True`)

EXAMPLE:

```
sage: P.<a,b,c,d> = PolynomialRing(GF(127),4)
sage: I = sage.rings.ideal.Katsura(P)
sage: I.gens()
(a + 2*b + 2*c + 2*d - 1, a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a, 2*a*b + 2*b*c
+ 2*c*d - b, b^2 + 2*a*c + 2*b*d - c)

sage: F = mq.MPolynomialSystem(I)
sage: A,v = F.coefficient_matrix()
sage: A
[  0   0   0   0   0   0   0   0   0   1   2   2   2 126]
[  1   0   2   0   0   2   0   0   2 126   0   0   0   0]
[  0   2   0   0   2   0   0   2   0   0 126   0   0   0]
[  0   0   1   2   0   0   2   0   0   0   0 126   0   0]

sage: v
[a^2]
[a*b]
[b^2]
[a*c]
[b*c]
[c^2]
[b*d]
[c*d]
[d^2]
[  a]
[  b]
[  c]
[  d]
[  1]

sage: A*v
[        a + 2*b + 2*c + 2*d - 1]
[a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a]
[      2*a*b + 2*b*c + 2*c*d - b]
[        b^2 + 2*a*c + 2*b*d - c]
```

connected_components()

Split the polynomial system in systems which do not share any variables.

EXAMPLE:

As an example consider one round of AES, which naturally splits into four subsystems which are independent:

```
sage: sr = mq.SR(2,4,4,8,gf2=True,polybori=True)
sage: F,s = sr.polynomial_system()
sage: Fz = mq.MPolynomialSystem(F.round(2))
sage: Fz.connected_components()
[Polynomial System with 128 Polynomials in 128 Variables,
 Polynomial System with 128 Polynomials in 128 Variables,
 Polynomial System with 128 Polynomials in 128 Variables,
 Polynomial System with 128 Polynomials in 128 Variables]
```

connection_graph()

Return the graph which has the variables of this system as vertices and edges between two variables if they appear in the same polynomial.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing()
sage: F = mq.MPolynomialSystem([x*y + y + 1, z + 1])
sage: F.connection_graph()
Graph on 3 vertices
```

gen(*ij*)

Return an element in this system indexed by `ij`.

INPUT:

- `ij` - tuple, slice, integer

EXAMPLES:

```
sage: P.<a,b,c,d> = PolynomialRing(GF(127),4)
sage: F = mq.MPolynomialSystem(sage.rings.ideal.Katsura(P))
```

`ij`-th polynomial overall:

```
sage: F[0] # indirect doctest
a + 2*b + 2*c + 2*d - 1
```

`i`-th to `j`-th polynomial overall:

```
sage: F[0:2]
(a + 2*b + 2*c + 2*d - 1, a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a)
```

`i`-th round, `j`-th polynomial:

```
sage: F[0,1]
a^2 + 2*b^2 + 2*c^2 + 2*d^2 - a
```

gens()

Return tuple of polynomials in this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: l = F.gens()
sage: len(l), type(l)
(40, <type 'tuple'>)
```

groebner_basis(**args, **kwargs*)

Compute and return a Groebner basis for the ideal spanned by the polynomials in this system (`self.gens()`).

INPUT:

- **`args` - list of arguments passed to** `MPolynomialIdeal.groebner_basis` call
- **`kwargs` - dictionary of arguments passed to** `MPolynomialIdeal.groebner_basis` call

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: gb = F.groebner_basis()
sage: Ideal(gb).basis_is_groebner()
True
```

ideal()

Return Sage ideal spanned by the elements of this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: P = F.ring()
sage: I = F.ideal()
sage: I.elimination_ideal(P('s000*s001*s002*s003*w100*w101*w102*w103*x100*x101*x102*x103'))
Ideal (k002 + (a^2)*k003 + 1,
       k001 + (a^3)*k003 + (a + 1),
       k000 + (a^3 + a^2 + a)*k003 + (a^3 + a^2 + a),
       k103 + (a^3)*k003 + (a^2 + 1),
       k102 + (a^3 + a^2 + a)*k003 + (a^3 + 1),
       k101 + k003 + (a^2 + a),
       k100 + (a^2)*k003 + (a^2 + a),
       k003^2 + (a^3 + a^2 + a)*k003 + (a^3 + a^2 + a))
of Multivariate Polynomial Ring in k100, k101, k102, k103,
x100, x101, x102, x103, w100, w101, w102, w103, s000,
s001, s002, s003, k000, k001, k002, k003 over Finite Field
in a of size 2^4
```

monomials()

Return an unordered tuple of monomials in this polynomial system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: len(F.monomials())
49
```

ngens()

Return number polynomials in this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: F.ngens()
56
```

nmonomials()

Return the number of monomials present in this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.nmonomials()
49
```

nrounds()

Return number of rounds of this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.nrounds()
4
```

nvariables()

Return number of variables present in this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.nvariables()
20
```

ring()

Return base ring.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True,gf2=True,order='block')
sage: F,s = sr.polynomial_system()
sage: print F.ring().repr_long()
Polynomial Ring
 Base Ring : Finite Field of size 2
      Size : 20 Variables
  Block  0 : Ordering : degrevlex
             Names    : k100, k101, k102, k103, x100, x101, x102, x103, w100, w101, w102, w1
  Block  1 : Ordering : degrevlex
             Names    : k000, k001, k002, k003
```

round(*i*)

Return `i`-th round of this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: R0 = F.round(1)
sage: R0
(k000^2 + k001, k001^2 + k002, k002^2 + k003, k003^2 + k000)
```

rounds()

Return a tuple of rounds of this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: l = F.rounds()
sage: len(l)
4
```

subs(**args, **kwargs*)

Substitute variables for every polynomial in this system and return a new system. See `MPolynomial.subs` for calling convention.

INPUT:

- `args` - arguments to be passed to `MPolynomial.subs`
- `kwargs` - keyword arguments to be passed to `MPolynomial.subs`

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system(); F
Polynomial System with 40 Polynomials in 20 Variables
sage: F = F.subs(s); F
Polynomial System with 40 Polynomials in 16 Variables
```

variables()

Return all variables present in this system. This tuple may or may not be equal to the generators of the ring of this system.

EXAMPLE:

```
sage: sr = mq.SR(allow_zero_inversions=True)
sage: F,s = sr.polynomial_system()
sage: F.variables()[:10]
(k003, k002, k001, k000, s003, s002, s001, s000, w103, w102)
```

class MPolynomialSystem_gf2(*R, rounds*)

Polynomial Systems over $\mathbb{F}_2$.

eliminate_linear_variables(*maxlength=3, skip=<function <lambda> at 0x2dc9f50>*)

Return a new system where "linear variables" are eliminated.

In this function we call a variable "linear" if it appears as a leading term of a linear polynomial.

INPUT:

- `maxlength` - an optional upper bound on the number of monomials by which a variable is replaced.
- `skip` - an optional callable to skip eliminations. It must accept two parameters and return either `True` or `False`. The two parameters are the leading term and the tail of a polynomial (default: `lambda lm,tail: False`).

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing()
sage: F = mq.MPolynomialSystem([c + d + b + 1, a + c + d, a*b + c, b*c*d + c])
sage: F.eliminate_linear_variables().gens() # everything vanishes
()
sage: F.eliminate_linear_variables(maxlength=2).gens()
(b + c + d + 1, b*c + b*d + c, b*c*d + c)
sage: F.eliminate_linear_variables(skip=lambda lm,tail: str(lm)=='a').gens()
(a + c + d, a*c + a*d + a + c, c*d + c)
```

Note: This is called "massaging" in [CB07].

REFERENCES:

class MPolynomialSystem_gf2e(*R, rounds*)

MPolynomialSystem over $\mathbb{F}_{2^e}$.

change_ring(*k*)

Project self onto `k` using the Weil restriction of scalars.

INPUT:

- `k` - GF(2) (parameter only for compatible syntax)

EXAMPLE:

```
sage: k.<a> = GF(2^2)
sage: P.<x,y> = PolynomialRing(k,2)
sage: a = P.base_ring().gen()
sage: F = mq.MPolynomialSystem(P,[x*y + 1, a*x + 1])
sage: F
Polynomial System with 2 Polynomials in 2 Variables
sage: F2 = F.change_ring(GF(2)); F2
```

```
doctest... DeprecationWarning: The use of this function is deprecated please use the weil_re
Polynomial System with 8 Polynomials in 4 Variables
sage: F2.gens()
(x1*y0 + x0*y1 + x1*y1,
x0*y0 + x1*y1 + 1,
x0 + x1,
x1 + 1,
x0^2 + x0,
x1^2 + x1,
y0^2 + y0,
y1^2 + y1)
```

Note: This function is deprecated use the `weil_restriction()` function instead.

weil_restriction()

Project this polynomial system to $\mathbb{F}_2$.

That is, compute the Weil restriction of scalars for the variety corresponding to this polynomial system and express it as a polynomial system over $\mathbb{F}_2$.

EXAMPLE:

```
sage: k.<a> = GF(2^2)
sage: P.<x,y> = PolynomialRing(k,2)
sage: a = P.base_ring().gen()
sage: F = mq.MPolynomialSystem(P,[x*y + 1, a*x + 1])
sage: F
Polynomial System with 2 Polynomials in 2 Variables
sage: F2 = F.weil_restriction(); F2
Polynomial System with 8 Polynomials in 4 Variables
sage: F2.gens()
(x1*y0 + x0*y1 + x1*y1,
x0*y0 + x1*y1 + 1,
x0 + x1,
x1 + 1,
x0^2 + x0,
x1^2 + x1,
y0^2 + y0,
y1^2 + y1)
```

Another bigger example for a small scale AES:

```
sage: sr = mq.SR(1,1,1,4,gf2=False)
sage: F,s = sr.polynomial_system(); F
Polynomial System with 40 Polynomials in 20 Variables
sage: F2 = F.weil_restriction(); F2
Polynomial System with 240 Polynomials in 80 Variables
```

is_MPolynomialRoundSystem(F)

Return `True` if `F` is an `MPolynomialRoundSystem`.

INPUT:

- `F` - anything

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQ)
sage: I = [[x^2 + y^2], [x^2 - y^2]]
sage: F = mq.MPolynomialSystem(P,I); F
Polynomial System with 2 Polynomials in 2 Variables
sage: mq.is_MPolynomialRoundSystem(F.round(0))
True
```

is_MPolynomialSystem(*F*)

Return `True` if `F` is an `MPolynomialSystem`.

INPUT:

- ``F`` - anything

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQ)
sage: I = [[x^2 + y^2], [x^2 - y^2]]
sage: F = mq.MPolynomialSystem(P,I); F
Polynomial System with 2 Polynomials in 2 Variables
sage: mq.is_MPolynomialSystem(F)
True
```

16.10 S-Boxes and Their Algebraic Representations

class SBox(**args, **kwargs*)

A substitution box or S-box is one of the basic components of symmetric key cryptography. In general, an S-box takes `m` input bits and transforms them into `n` output bits. This is called an `mxn` S-box and is often implemented as a lookup table. These S-boxes are carefully chosen to resist linear and differential cryptanalysis [Heys02].

This module implements an S-box class which allows an algebraic treatment.

EXAMPLE:

We consider the S-box of the block cipher PRESENT [PRESENT07]:

```
sage: S = mq.SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2); S
(12, 5, 6, 11, 9, 0, 10, 13, 3, 14, 15, 8, 4, 7, 1, 2)
sage: S(1)
5
```

Note that by default bits are interpreted in big endian order. This is not consistent with the rest of Sage, which has a strong bias towards little endian, but is consistent with most cryptographic literature:

```
sage: S([0,0,0,1])
[0, 1, 0, 1]

sage: S = mq.SBox(12,5,6,11,9,0,10,13,3,14,15,8,4,7,1,2, big_endian=False)
sage: S(1)
5
sage: S([0,0,0,1])
[1, 1, 0, 0]
```

Now we construct an `SBox` object for the 4-bit small scale AES S-Box (cf. `sage.crypto.mq.sr`):

```
sage: sr = mq.SR(1,1,1,4, allow_zero_inversions=True)
sage: S = mq.SBox([sr.sub_byte(e) for e in list(sr.k)])
sage: S
(6, 5, 2, 9, 4, 7, 3, 12, 14, 15, 10, 0, 8, 1, 13, 11)
```

REFERENCES:

difference_distribution_matrix()

Return difference distribution matrix `A` for this S-box.

The rows of `A` encode the differences `Delta I` of the input and the columns encode the difference `Delta O` for the output. The bits are ordered according to the endianess of this S-box. The value at `A[Delta`

I,Delta O] encodes how often Delta O is the actual output difference given Delta I as input difference.

See [Heys02] for an introduction to differential cryptanalysis.

EXAMPLE:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: S.difference_distribution_matrix()
[8 0 0 0 0 0 0 0]
[0 2 2 0 2 0 0 2]
[0 0 2 2 0 0 2 2]
[0 2 0 2 2 0 2 0]
[0 2 0 2 0 2 0 2]
[0 0 2 2 2 2 0 0]
[0 2 2 0 0 2 2 0]
[0 0 0 0 2 2 2 2]
```

from_bits(*x*, *n=None*)

Return integer for bitstring x of length n.

INPUT:

- x - a bitstring
- n - bit length (optional)

EXAMPLE:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: S.from_bits( [1,1,0])
6

sage: S( S.from_bits( [1,1,0] ) )
1
sage: S.from_bits( S( [1,1,0] ) )
1
```

interpolation_polynomial(*k=None*)

Return a univariate polynomial over an extension field representing this S-box.

If m is the input length of this S-box then the extension field is of degree m.

If the output length does not match the input length then a TypeError is raised.

INPUT:

- k - an instance of $\mathbf{F}_{2^m}$ (default: None)

EXAMPLE:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: f = S.interpolation_polynomial()
sage: f
x^6 + a*x^5 + (a + 1)*x^4 + (a^2 + a + 1)*x^3
  + (a^2 + 1)*x^2 + (a + 1)*x + a^2 + a + 1

sage: a = f.base_ring().gen()

sage: f(0), S(0)
(a^2 + a + 1, 7)

sage: f(a^2 + 1), S(5)
(a^2 + 1, 5)
```

is_permutation()

Return True if this S-Box is a permutation.

EXAMPLE:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: S.is_permutation()
True

sage: S = mq.SBox(3,2,0,0,2,1,1,3)
sage: S.is_permutation()
False
```

linear_approximation_matrix()

Return linear approximation matrix `A` for this S-box.

Let `i_b` be the `b`-th bit of `i` and `o_b` the `b`-th bit of `o`. Then `v = A[i,o]` encodes the bias of the equation `sum( i_b * x_i ) = sum( o_b * y_i )` if `x_i` and `y_i` represent the input and output variables of the S-box.

See [Heys02] for an introduction to linear cryptanalysis.

EXAMPLE:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: S.linear_approximation_matrix()
[ 4  0  0  0  0  0  0  0]
[ 0  0  0  0  2  2  2 -2]
[ 0  0 -2 -2 -2  2  0  0]
[ 0  0 -2  2  0  0 -2 -2]
[ 0  2  0  2 -2  0  2  0]
[ 0 -2  0  2  0  2  0  2]
[ 0 -2 -2  0  0 -2  2  0]
[ 0 -2  2  0 -2  0  0 -2]
```

According to this matrix the first bit of the input is equal to the third bit of the output 6 out of 8 times:

```
sage: for i in srange(8): print S.to_bits(i)[0] == S.to_bits(S(i))[2]
False
True
True
True
False
True
True
True
```

maximal_difference_probability()

Return the difference probability of the difference with the highest probability in the range between 0.0 and 1.0 indicating 0% or 100% respectively.

EXAMPLE:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: S.maximal_difference_probability()
0.25
```

maximal_difference_probability_absolute()

Return the difference probability of the difference with the highest probability in absolute terms, i.e. how often it occurs in total.

EXAMPLE:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: S.maximal_difference_probability_absolute()
2
```

Note: This code is mainly called internally.

maximal_linear_bias_absolute()

Return maximal linear bias, i.e. how often the linear approximation with the highest bias is true or false minus 2^{n-1}.

EXAMPLE:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: S.maximal_linear_bias_absolute()
2
```

maximal_linear_bias_relative()

Return maximal bias of all linear approximations of this S-box.

EXAMPLE:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: S.maximal_linear_bias_relative()
0.25
```

polynomials(*X=None, Y=None, degree=2, groebner=False*)

Return a list of polynomials satisfying this S-box.

First, a simple linear fitting is performed for the given `degree` (cf. for example [BC03]). If `groebner=True` a Groebner basis is also computed for the result of that process.

INPUT:

- `X` - input variables
- `Y` - output variables
- `degree` - integer > 0 (default: `2`)
- `groebner` - calculate a reduced Groebner basis of the spanning polynomials to obtain more polynomials (default: `False`)

EXAMPLES:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: P = S.ring()
```

By default, this method returns an indirect representation:

```
sage: S.polynomials()
[x0*x2 + x1 + y1 + 1,
 x0*x1 + x1 + x2 + y0 + y1 + y2 + 1,
 x0*y1 + x0 + x2 + y0 + y2,
 x0*y0 + x0*y2 + x1 + x2 + y0 + y1 + y2 + 1,
 x1*x2 + x0 + x1 + x2 + y2 + 1,
 x0*y0 + x1*y0 + x0 + x2 + y1 + y2,
 x0*y0 + x1*y1 + x1 + y1 + 1,
 x1*y2 + x1 + x2 + y0 + y1 + y2 + 1,
 x0*y0 + x2*y0 + x1 + x2 + y1 + 1,
 x2*y1 + x0 + y1 + y2,
 x2*y2 + x1 + y1 + 1,
 y0*y1 + x0 + x2 + y0 + y1 + y2,
 y0*y2 + x1 + x2 + y0 + y1 + 1,
 y1*y2 + x2 + y0]
```

We can get a direct representation by computing a lexicographical Groebner basis with respect to the right variable ordering, i.e. a variable ordering where the output bits are greater than the input bits:

```
sage: P.<y0,y1,y2,x0,x1,x2> = PolynomialRing(GF(2),6,order='lex')
sage: S.polynomials([x0,x1,x2],[y0,y1,y2], groebner=True)
[y0 + x0*x1 + x0*x2 + x0 + x1*x2 + x1 + 1,
 y1 + x0*x2 + x1 + 1,
 y2 + x0 + x1*x2 + x1 + x2 + 1]
```

REFERENCES:

ring()

Create, return and cache a polynomial ring for S-box polynomials.

EXAMPLE:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: S.ring()
Multivariate Polynomial Ring in x0, x1, x2, y0, y1, y2 over Finite Field of size 2
```

solutions(*X=None, Y=None*)

Return a dictionary of solutions to this S-box.

INPUT:

- `X` - input variables (default: `None`)
- `Y` - output variables (default: `None`)

EXAMPLE:

```
sage: S = mq.SBox([7,6,0,4,2,5,1,3])
sage: F = S.polynomials()
sage: s = S.solutions()
sage: any(f.subs(_s) for f in F for _s in s)
False
```

to_bits(*x, n=None*)

Return bitstring of length `n` for integer `x`. The returned bitstring is guaranteed to have length `n`.

INPUT:

- `x` - an integer
- `n` - bit length (optional)

EXAMPLE:

```
sage: S = mq.SBox(7,6,0,4,2,5,1,3)
sage: S.to_bits(6)
[1, 1, 0]

sage: S.to_bits( S(6) )
[0, 0, 1]

sage: S( S.to_bits( 6 ) )
[0, 0, 1]
```

CHAPTER

SEVENTEEN

COMBINATORICS

17.1 Combinatorial Functions.

AUTHORS:

- David Joyner (2006-07): initial implementation.
- William Stein (2006-07): editing of docs and code; many optimizations, refinements, and bug fixes in corner cases
- David Joyner (2006-09): bug fix for combinations, added permutations_iterator, combinations_iterator from Python Cookbook, edited docs.
- David Joyner (2007-11): changed permutations, added hadamard_matrix
- Florent Hivert (2009-02): combinatorial class cleanup

This module implements some combinatorial functions, as listed below. For a more detailed description, see the relevant docstrings.

Sequences:

- Bell numbers, `bell_number`
- Bernoulli numbers, `bernoulli_number` (though PARI's bernoulli is better)
- Catalan numbers, `catalan_number` (not to be confused with the Catalan constant)
- Eulerian/Euler numbers, `euler_number` (Maxima)
- Fibonacci numbers, `fibonacci` (PARI) and `fibonacci_number` (GAP) The PARI version is better.
- Lucas numbers, `lucas_number1`, `lucas_number2`.
- Stirling numbers, `stirling_number1`, `stirling_number2`.

Set-theoretic constructions:

- Combinations of a multiset, `combinations`, `combinations_iterator`, and `number_of_combinations`. These are unordered selections without repetition of k objects from a multiset S.
- Arrangements of a multiset, `arrangements` and `number_of_arrangements` These are ordered selections without repetition of k objects from a multiset S.

- Derangements of a multiset, `derangements` and `number_of_derangements`.
- Tuples of a multiset, `tuples` and `number_of_tuples`. An ordered tuple of length k of set S is a ordered selection with repetitions of S and is represented by a sorted list of length k containing elements from S.
- Unordered tuples of a set, `unordered_tuple` and `number_of_unordered_tuples`. An unordered tuple of length k of set S is an unordered selection with repetitions of S and is represented by a sorted list of length k containing elements from S.
- Permutations of a multiset, `permutations`, `permutations_iterator`, `number_of_permutations`. A permutation is a list that contains exactly the same elements but possibly in different order.

Partitions:

- Partitions of a set, `partitions_set`, `number_of_partitions_set`. An unordered partition of set S is a set of pairwise disjoint nonempty sets with union S and is represented by a sorted list of such sets.
- Partitions of an integer, `partitions_list`, `number_of_partitions_list`. An unordered partition of n is an unordered sum $n = p_1 + p_2 + \ldots + p_k$ of positive integers and is represented by the list $p = [p_1, p_2, \ldots, p_k]$, in nonincreasing order, i.e., $p1 \geq p_2 ... \geq p_k$.
- Ordered partitions of an integer, `ordered_partitions`, `number_of_ordered_partitions`. An ordered partition of n is an ordered sum $n = p_1 + p_2 + \ldots + p_k$ of positive integers and is represented by the list $p = [p_1, p_2, \ldots, p_k]$, in nonincreasing order, i.e., $p1 \geq p_2 ... \geq p_k$.
- Restricted partitions of an integer, `partitions_restricted`, `number_of_partitions_restricted`. An unordered restricted partition of n is an unordered sum $n = p_1 + p_2 + \ldots + p_k$ of positive integers p_i belonging to a given set S, and is represented by the list $p = [p_1, p_2, \ldots, p_k]$, in nonincreasing order, i.e., $p1 \geq p_2 ... \geq p_k$.
- `partitions_greatest` implements a special type of restricted partition.
- `partitions_greatest_eq` is another type of restricted partition.
- Tuples of partitions, `partition_tuples`, `number_of_partition_tuples`. A k-tuple of partitions is represented by a list of all k-tuples of partitions which together form a partition of n.
- Powers of a partition, `partition_power(pi, k)`. The power of a partition corresponds to the k-th power of a permutation with cycle structure π.
- Sign of a partition, `partition_sign( pi )` This means the sign of a permutation with cycle structure given by the partition pi.
- Associated partition, `partition_associated( pi )` The "associated" (also called "conjugate" in the literature) partition of the partition pi which is obtained by transposing the corresponding Ferrers diagram.
- Ferrers diagram, `ferrers_diagram`. Analogous to the Young diagram of an irreducible representation of S_n.

Related functions:

- Bernoulli polynomials, `bernoulli_polynomial`

Implemented in other modules (listed for completeness):

The `sage.rings.arith` module contains the following combinatorial functions:

- binomial the binomial coefficient (wrapped from PARI)

- factorial (wrapped from PARI)
- partition (from the Python Cookbook) Generator of the list of all the partitions of the integer n.
- `number_of_partitions` (wrapped from PARI) the *number* of partitions:
- `falling_factorial` Definition: for integer $a \geq 0$ we have $x(x-1)\cdots(x-a+1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+1)}{\Gamma(x-a+1)}$.
- `rising_factorial` Definition: for integer $a \geq 0$ we have $x(x+1)\cdots(x+a-1)$. In all other cases we use the GAMMA-function: $\frac{\Gamma(x+a)}{\Gamma(x)}$.
- gaussian_binomial the gaussian binomial

$$\binom{n}{k}_q = \frac{(1-q^m)(1-q^{m-1})\cdots(1-q^{m-r+1})}{(1-q)(1-q^2)\cdots(1-q^r)}.$$

The `sage.groups.perm_gps.permgroup_elements` contains the following combinatorial functions:

- matrix method of PermutationGroupElement yielding the permutation matrix of the group element.

```
TODO:
   GUAVA commands:
    * MOLS returns a list of n Mutually Orthogonal Latin Squares (MOLS).
    * VandermondeMat
    * GrayMat returns a list of all different vectors of length n over
      the field F, using Gray ordering.
   Not in GAP:
    * Rencontres numbers
      http://en.wikipedia.org/wiki/Rencontres_number
```

REFERENCES:

- http://en.wikipedia.org/wiki/Twelvefold_way (general reference)

class CombinatorialClass()

cardinality()

Default implementation of cardinality which just goes through the iterator of the combinatorial class to count the number of objects.

EXAMPLES:

```
sage: class C(CombinatorialClass):
...     def __iter__(self):
...         return iter([1,2,3])
...
sage: C().cardinality() #indirect doctest
3
```

count()

Deprecated ! Please use `.cardinality` instead.

TEST:

```
sage: class C(CombinatorialClass):
...     def __iter__(self):
...         return iter([1,2,3])
...
sage: C().count() #indirect doctest
doctest:1: DeprecationWarning: The usage of count for combinatorial classes is deprecated. P
3
```

filter (*f*, *name=None*)

Returns the combinatorial subclass of f which consists of the elements x of self such that f(x) is True.

EXAMPLES:

```
sage: P = Permutations(3).filter(lambda x: x.avoids([1,2]))
sage: P.list()
[[3, 2, 1]]
```

first ()

Default implementation for first which uses iterator.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.first() # indirect doctest
1
```

iterator ()

Iterator is deprecated for combinatorial classes.

EXAMPLES:

```
sage: p5 = Partitions(3)
sage: it = p5.iterator()
doctest:1: DeprecationWarning: The usage of iterator for combinatorial classes is deprecated
sage: [i for i in it]
[[3], [2, 1], [1, 1, 1]]
sage: [i for i in p5]
[[3], [2, 1], [1, 1, 1]]
```

last ()

Default implementation for first which uses iterator.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.last() # indirect doctest
3
```

list ()

The default implementation of list which builds the list from the iterator.

EXAMPLES:

```
sage: class C(CombinatorialClass):
...     def __iter__(self):
...         return iter([1,2,3])
...
sage: C().list() #indirect doctest
[1, 2, 3]
```

map (*f*, *name=None*)

Returns the image $\{f(x) x in self f\}$ of this combinatorial class by f, as a combinatorial class.

f is supposed to be injective.

EXAMPLES:

```
sage: R = Permutations(3).map(attrcall('reduced_word')); R
Image of Standard permutations of 3 by *.reduced_word()
sage: R.cardinality()
6
sage: R.list()
[[], [2], [1], [1, 2], [2, 1], [2, 1, 2]]
sage: [ r for r in R]
[[], [2], [1], [1, 2], [2, 1], [2, 1, 2]]

If the function is not injective, then there may be repeated elements:
sage: P = Partitions(4)
sage: P.list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: P.map(len).list()
[1, 2, 2, 3, 4]
```

TESTS:

```
sage: R = Permutations(3).map(attrcall('reduced_word'))
sage: R == loads(dumps(R))
True
```

next(*obj*)

Default implementation for next which uses iterator.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.next(2) # indirect doctest
3
```

object_class()

previous(*obj*)

Default implementation for next which uses iterator.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.previous(2) # indirect doctest
1
```

random()

Deprecated. Use self.random_element() instead.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.random()
...
NotImplementedError: Deprecated: use random_element() instead
```

random_element()

Default implementation of random which uses unrank.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
```

```
sage: C.random_element()
1
```

rank(*obj*)

Default implementation of rank which uses iterator.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.rank(3) # indirect doctest
2
```

union(*right_cc*, *name=None*)

Returns the combinatorial class representing the union of self and right_cc.

EXAMPLES:

```
sage: P = Permutations(2).union(Permutations(1))
sage: P.list()
[[1, 2], [2, 1], [1]]
```

unrank(*r*)

Default implementation of unrank which goes through the iterator.

EXAMPLES:

```
sage: C = CombinatorialClass()
sage: C.list = lambda: [1,2,3]
sage: C.unrank(1) # indirect doctest
2
```

class **CombinatorialObject**(*l*)

index(*key*)

EXAMPLES:

```
sage: c = CombinatorialObject([1,2,3])
sage: c.index(1)
0
sage: c.index(3)
2
```

class **FilteredCombinatorialClass**(*combinatorial_class*, *f*, *name=None*)

cardinality()

EXAMPLES:

```
sage: P = Permutations(3).filter(lambda x: x.avoids([1,2]))
sage: P.cardinality()
1
```

class **InfiniteAbstractCombinatorialClass**()

This is an internal Class this should not be used directly. A class wich inerits from InfiniteAbstractCombinatorialClass inherits the standard methods list and count.

If self._infinite_cclass_slice exists then self.__iter__ returns an iterator for self, otherwise raise NotImplementedError. The method self._infinite_cclass_slice is supposed to accept any integer as an argument and return something which is iterable.

cardinality()

Counts the elements of the combinatorial class.

EXAMPLES: sage: R = InfiniteAbstractCombinatorialClass() sage: R.cardinality() +Infinity

list()

Returns an error since self is an infinite combinatorial class.

EXAMPLES: sage: R = InfiniteAbstractCombinatorialClass() sage: R.list() Traceback (most recent call last): ... NotImplementedError: infinite list

class MapCombinatorialClass(*cc, f, name=None*)

A MapCombinatorialClass models the image of a combinatorial class through a function which is assumed to be injective

See CombinatorialClass.map for examples

cardinality()

Returns the cardinality of this combinatorial class

EXAMPLES: sage: R = Permutations(10).map(attrcall('reduced_word')) sage: R.cardinality() 3628800

class UnionCombinatorialClass(*left_cc, right_cc, name=None*)

cardinality()

EXAMPLES:

```
sage: P = Permutations(3).union(Permutations(2))
sage: P.cardinality()
8
```

first()

EXAMPLES:

```
sage: P = Permutations(3).union(Permutations(2))
sage: P.first()
[1, 2, 3]
```

last()

EXAMPLES:

```
sage: P = Permutations(3).union(Permutations(2))
sage: P.last()
[2, 1]
```

list()

EXAMPLES:

```
sage: P = Permutations(3).union(Permutations(2))
sage: P.list()
[[1, 2, 3],
 [1, 3, 2],
 [2, 1, 3],
 [2, 3, 1],
 [3, 1, 2],
 [3, 2, 1],
 [1, 2],
 [2, 1]]
```

rank(*x*)

EXAMPLES:

```
sage: P = Permutations(3).union(Permutations(2))
sage: P.rank(Permutation([2,1]))
7
sage: P.rank(Permutation([1,2,3]))
0
```

unrank(*x*)

EXAMPLES:

```
sage: P = Permutations(3).union(Permutations(2))
sage: P.unrank(7)
[2, 1]
sage: P.unrank(0)
[1, 2, 3]
```

arrangements(*mset, k*)

An arrangement of mset is an ordered selection without repetitions and is represented by a list that contains only elements from mset, but maybe in a different order.

`arrangements` returns the set of arrangements of the multiset mset that contain k elements.

IMPLEMENTATION: Wraps GAP's Arrangements.

Warning: Wraps GAP - hence mset must be a list of objects that have string representations that can be interpreted by the GAP interpreter. If mset consists of at all complicated Sage objects, this function does *not* do what you expect. A proper function should be written! (TODO!)

EXAMPLES:

```
sage: mset = [1,1,2,3,4,4,5]
sage: arrangements(mset,2)
[[1, 1],
 [1, 2],
 [1, 3],
 [1, 4],
 [1, 5],
 [2, 1],
 [2, 3],
 [2, 4],
 [2, 5],
 [3, 1],
 [3, 2],
 [3, 4],
 [3, 5],
 [4, 1],
 [4, 2],
 [4, 3],
 [4, 4],
 [4, 5],
 [5, 1],
 [5, 2],
 [5, 3],
 [5, 4]]
 sage: arrangements( ["c","a","t"], 2 )
 ['ac', 'at', 'ca', 'ct', 'ta', 'tc']
 sage: arrangements( ["c","a","t"], 3 )
 ['act', 'atc', 'cat', 'cta', 'tac', 'tca']
```

bell_number(*n*)

Returns the n-th Bell number (the number of ways to partition a set of n elements into pairwise disjoint nonempty subsets).

If $n \leq 0$, returns 1.

Wraps GAP's Bell.

EXAMPLES:

```
sage: bell_number(10)
115975
sage: bell_number(2)
2
sage: bell_number(-10)
1
sage: bell_number(1)
1
sage: bell_number(1/3)
...
TypeError: no conversion of this rational to integer
```

bell_polynomial (*n, k*)

This function returns the Bell Polynomial

$$B_{n,k}(x_1, x_2, \ldots, x_{n-k+1}) = \sum_{\sum j_i = k, \sum i j_i = n} \frac{n!}{j_1! j_2! \ldots} \frac{x_1}{1!}^{j_1} \frac{x_2}{2!}^{j_2} \cdots$$

INPUT:

- n - integer
- k - integer

OUTPUT:

- polynomial expression (SymbolicArithmetic)

EXAMPLES:

```
sage: bell_polynomial(6,2)
10*x_3^2 + 15*x_2*x_4 + 6*x_1*x_5
sage: bell_polynomial(6,3)
15*x_2^3 + 60*x_1*x_2*x_3 + 15*x_1^2*x_4
```

REFERENCES:

- E.T. Bell, "Partition Polynomials"

AUTHORS:

- Blair Sutton (2009-01-26)

bernoulli_polynomial (*x, n*)

Return the nth Bernoulli polynomial as a polynomial in x. In particular, if x is anything other than a variable, this will simply be the nth Bernoulli polynomial evaluated at x.

The generating function for the Bernoulli polynomials is

$$\frac{te^{xt}}{e^t - 1} = \sum_{n=0}^{\infty} B_n(x) \frac{t^n}{n!},$$

and they are given directly by

$$B_n(x) = \sum_{i=0}^{n} \binom{n}{i} B_{n-i} x^i.$$

One has $B_n(x) = -n\zeta(1-n, x)$, where $\zeta(s, x)$ is the Hurwitz zeta function. Thus, in a certain sense, the Hurwitz zeta generalizes the Bernoulli polynomials to non-integer values of n.

EXAMPLES:

```
sage: y = QQ['y'].0
sage: bernoulli_polynomial(y,5)
y^5 - 5/2*y^4 + 5/3*y^3 - 1/6*y
sage: bernoulli_polynomial(y,5)(12)
199870
sage: bernoulli_polynomial(12,5)
199870
```

REFERENCES:

- http://en.wikipedia.org/wiki/Bernoulli_polynomials

catalan_number(n)

Returns the n-th Catalan number

Catalan numbers: The n-th Catalan number is given directly in terms of binomial coefficients by

$$C_n = \frac{1}{n+1}\binom{2n}{n} = \frac{(2n)!}{(n+1)!\,n!} \qquad \text{for } n \geq 0.$$

Consider the set $S = \{1, ..., n\}$. A noncrossing partition of S is a partition in which no two blocks "cross" each other, i.e., if a and b belong to one block and x and y to another, they are not arranged in the order axby. C_n is the number of noncrossing partitions of the set S. There are many other interpretations (see REFERENCES).

When $n = -1$, this function raises a ZeroDivisionError; for other $n < 0$ it returns 0.

EXAMPLES:

```
sage: [catalan_number(i) for i in range(7)]
[1, 1, 2, 5, 14, 42, 132]
sage: maxima.eval("-(1/2)*taylor (sqrt (1-4*x^2), x, 0, 15)")
'-1/2+x^2+x^4+2*x^6+5*x^8+14*x^10+42*x^12+132*x^14'
sage: [catalan_number(i) for i in range(-7,7) if i != -1]
[0, 0, 0, 0, 0, 0, 1, 1, 2, 5, 14, 42, 132]
sage: catalan_number(-1)
...
ZeroDivisionError: Rational division by zero
```

REFERENCES:

- http://en.wikipedia.org/wiki/Catalan_number
- http://www-history.mcs.st-andrews.ac.uk/~history/Miscellaneous/CatalanNumbers/catalan.html

combinations(*mset, k*)

A combination of a multiset (a list of objects which may contain the same object several times) mset is an unordered selection without repetitions and is represented by a sorted sublist of mset. Returns the set of all combinations of the multiset mset with k elements.

Warning: Wraps GAP's Combinations. Hence mset must be a list of objects that have string representations that can be interpreted by the GAP interpreter. If mset consists of at all complicated Sage objects, this function does *not* do what you expect. A proper function should be written! (TODO!)

EXAMPLES:

```
sage: mset = [1,1,2,3,4,4,5]
sage: combinations(mset,2)
[[1, 1],
 [1, 2],
 [1, 3],
```

```
 [1, 4],
 [1, 5],
 [2, 3],
 [2, 4],
 [2, 5],
 [3, 4],
 [3, 5],
 [4, 4],
 [4, 5]]
 sage: mset = ["d","a","v","i","d"]
 sage: combinations(mset,3)
 ['add', 'adi', 'adv', 'aiv', 'ddi', 'ddv', 'div']
```

Note: For large lists, this raises a string error.

combinations_iterator(*mset, n=None*)

Posted by Raymond Hettinger, 2006/03/23, to the Python Cookbook: http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/474124

Much faster than combinations.

EXAMPLES:

```
sage: X = combinations_iterator([1,2,3,4,5],3)
sage: [x for x in X]
[[1, 2, 3],
 [1, 2, 4],
 [1, 2, 5],
 [1, 3, 4],
 [1, 3, 5],
 [1, 4, 5],
 [2, 3, 4],
 [2, 3, 5],
 [2, 4, 5],
 [3, 4, 5]]
```

cyclic_permutations(*mset*)

Returns a list of all cyclic permutations of mset. Treats mset as a list, not a set, i.e. entries with the same value are distinct.

AUTHORS:

- Emily Kirkman

EXAMPLES:

```
sage: from sage.combinat.combinat import cyclic_permutations, cyclic_permutations_iterator
sage: cyclic_permutations(range(4))
[[0, 1, 2, 3], [0, 1, 3, 2], [0, 2, 1, 3], [0, 2, 3, 1], [0, 3, 1, 2], [0, 3, 2, 1]]
sage: for cycle in cyclic_permutations(['a', 'b', 'c']):
...       print cycle
['a', 'b', 'c']
['a', 'c', 'b']
```

Note that lists with repeats are not handled intuitively:

```
sage: cyclic_permutations([1,1,1])
[[1, 1, 1], [1, 1, 1]]
```

cyclic_permutations_iterator(*mset*)

Iterates over all cyclic permutations of mset in cycle notation. Treats mset as a list, not a set, i.e. entries with the same value are distinct.

AUTHORS:

- Emily Kirkman

EXAMPLES:

```
sage: from sage.combinat.combinat import cyclic_permutations, cyclic_permutations_iterator
sage: cyclic_permutations(range(4))
[[0, 1, 2, 3], [0, 1, 3, 2], [0, 2, 1, 3], [0, 2, 3, 1], [0, 3, 1, 2], [0, 3, 2, 1]]
sage: for cycle in cyclic_permutations(['a', 'b', 'c']):
...       print cycle
['a', 'b', 'c']
['a', 'c', 'b']
```

Note that lists with repeats are not handled intuitively:

```
sage: cyclic_permutations([1,1,1])
[[1, 1, 1], [1, 1, 1]]
```

derangements(*mset*)

A derangement is a fixed point free permutation of list and is represented by a list that contains exactly the same elements as mset, but possibly in different order. Derangements returns the set of all derangements of a multiset.

Wraps GAP's Derangements.

Warning: Wraps GAP - hence mset must be a list of objects that have string representations that can be interpreted by the GAP interpreter. If mset consists of at all complicated Sage objects, this function does *not* do what you expect. A proper function should be written! (TODO!)

EXAMPLES:

```
sage: mset = [1,2,3,4]
sage: derangements(mset)
[[2, 1, 4, 3],
 [2, 3, 4, 1],
 [2, 4, 1, 3],
 [3, 1, 4, 2],
 [3, 4, 1, 2],
 [3, 4, 2, 1],
 [4, 1, 2, 3],
 [4, 3, 1, 2],
 [4, 3, 2, 1]]
 sage: derangements(["c","a","t"])
 ['atc', 'tca']
```

euler_number(*n*)

Returns the n-th Euler number

IMPLEMENTATION: Wraps Maxima's euler.

EXAMPLES:

```
sage: [euler_number(i) for i in range(10)]
[1, 0, -1, 0, 5, 0, -61, 0, 1385, 0]
sage: maxima.eval("taylor (2/(exp(x)+exp(-x)), x, 0, 10)")
'1-x^2/2+5*x^4/24-61*x^6/720+277*x^8/8064-50521*x^10/3628800'
```

```
sage: [euler_number(i)/factorial(i) for i in range(11)]
[1, 0, -1/2, 0, 5/24, 0, -61/720, 0, 277/8064, 0, -50521/3628800]
sage: euler_number(-1)
...
ValueError: n (=-1) must be a nonnegative integer
```

REFERENCES:

- http://en.wikipedia.org/wiki/Euler_number

fibonacci(*n, algorithm='pari'*)

Returns the n-th Fibonacci number. The Fibonacci sequence F_n is defined by the initial conditions $F_1 = F_2 = 1$ and the recurrence relation $F_{n+2} = F_{n+1} + F_n$. For negative n we define $F_n = (-1)^{n+1}F_{-n}$, which is consistent with the recurrence relation.

INPUT:

- `algorithm` - string:
- `"pari"` - (default) - use the PARI C library's fibo function.
- `"gap"` - use GAP's Fibonacci function

Note: PARI is tens to hundreds of times faster than GAP here; moreover, PARI works for every large input whereas GAP doesn't.

EXAMPLES:

```
sage: fibonacci(10)
55
sage: fibonacci(10, algorithm='gap')
55
```

```
sage: fibonacci(-100)
-354224848179261915075
sage: fibonacci(100)
354224848179261915075
```

```
sage: fibonacci(0)
0
sage: fibonacci(1/2)
...
TypeError: no conversion of this rational to integer
```

fibonacci_sequence(*start, stop=None, algorithm=None*)

Returns an iterator over the Fibonacci sequence, for all fibonacci numbers f_n from `n = start` up to (but not including) `n = stop`

INPUT:

- `start` - starting value
- `stop` - stopping value
- `algorithm` - default (None) - passed on to fibonacci function (or not passed on if None, i.e., use the default).

EXAMPLES:

```
sage: fibs = [i for i in fibonacci_sequence(10, 20)]
sage: fibs
[55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181]
```

```
sage: sum([i for i in fibonacci_sequence(100, 110)])
699193769230753087300013
```

See Also:

fibonacci_xrange()

AUTHORS:

- Bobby Moretti

fibonacci_xrange(*start, stop=None, algorithm='pari'*)

Returns an iterator over all of the Fibonacci numbers in the given range, including `f_n = start` up to, but not including, `f_n = stop`.

EXAMPLES:

```
sage: fibs_in_some_range =  [i for i in fibonacci_xrange(10^7, 10^8)]
sage: len(fibs_in_some_range)
4
sage: fibs_in_some_range
[14930352, 24157817, 39088169, 63245986]
```

```
sage: fibs = [i for i in fibonacci_xrange(10, 100)]
sage: fibs
[13, 21, 34, 55, 89]
```

```
sage: list(fibonacci_xrange(13, 34))
[13, 21]
```

A solution to the second Project Euler problem:

```
sage: sum([i for i in fibonacci_xrange(10^6) if is_even(i)])
1089154
```

See Also:

fibonacci_sequence()

AUTHORS:

- Bobby Moretti

hadamard_matrix(*n*)

Returns an n x n Hadamard matrix of order n, if possible.

If the construction of this matrix is not implemented in GUAVA or there is no such matrix, raises a NotImplementedError.

EXAMPLES:

```
sage: hadamard_matrix(4)
[ 1  1  1  1]
[ 1 -1  1 -1]
[ 1  1 -1 -1]
[ 1 -1 -1  1]
sage: hadamard_matrix(6)
...
NotImplementedError: Hadamard matrix of order 6 does not exist or is not implemented yet.
```

hurwitz_zeta(*s, x, N*)

Returns the value of the $\zeta(s,x)$ to N decimals, where s and x are real.

The Hurwitz zeta function is one of the many zeta functions. It defined as

$$\zeta(s,x) = \sum_{k=0}^{\infty} (k+x)^{-s}.$$

When $x = 1$, this coincides with Riemann's zeta function. The Dirichlet L-functions may be expressed as a linear combination of Hurwitz zeta functions.

Note that if you use floting point inputs, then the results may be slightly off.

EXAMPLES:

```
sage: hurwitz_zeta(3,1/2,6)
8.41439000000000
sage: hurwitz_zeta(11/10,1/2,6)
12.1041000000000
sage: hurwitz_zeta(11/10,1/2,50)
12.103813495683755105709077412966680619033648618088
```

REFERENCES:

- http://en.wikipedia.org/wiki/Hurwitz_zeta_function

lucas_number1(*n, P, Q*)

Returns the n-th Lucas number "of the first kind" (this is not standard terminology). The Lucas sequence $L_n^{(1)}$ is defined by the initial conditions $L_1^{(1)} = 0$, $L_2^{(1)} = 1$ and the recurrence relation $L_{n+2}^{(1)} = P * L_{n+1}^{(1)} - Q * L_n^{(1)}$.

Wraps GAP's Lucas(...)[1].

P=1, Q=-1 gives the Fibonacci sequence.

INPUT:

- `n` - integer
- `P, Q` - integer or rational numbers

OUTPUT: integer or rational number

EXAMPLES:

```
sage: lucas_number1(5,1,-1)
5
sage: lucas_number1(6,1,-1)
8
sage: lucas_number1(7,1,-1)
13
sage: lucas_number1(7,1,-2)
43


sage: lucas_number1(5,2,3/5)
229/25
sage: lucas_number1(5,2,1.5)
...
TypeError: no canonical coercion from Real Field with 53 bits of precision to Rational Field
```

There was a conjecture that the sequence L_n defined by $L_{n+2} = L_{n+1} + L_n$, $L_1 = 1$, $L_2 = 3$, has the property that n prime implies that L_n is prime.

```
sage: lucas = lambda n:(5/2)*lucas_number1(n,1,-1)+(1/2)*lucas_number2(n,1,-1)
sage: [[lucas(n),is_prime(lucas(n)),n+1,is_prime(n+1)] for n in range(15)]
[[1, False, 1, False],
 [3, True, 2, True],
 [4, False, 3, True],
 [7, True, 4, False],
 [11, True, 5, True],
 [18, False, 6, False],
 [29, True, 7, True],
 [47, True, 8, False],
 [76, False, 9, False],
 [123, False, 10, False],
 [199, True, 11, True],
 [322, False, 12, False],
 [521, True, 13, True],
 [843, False, 14, False],
 [1364, False, 15, False]]
```

Can you use Sage to find a counterexample to the conjecture?

lucas_number2 (*n, P, Q*)

Returns the n-th Lucas number "of the second kind" (this is not standard terminology). The Lucas sequence $L_n^{(2)}$ is defined by the initial conditions $L_1^{(2)} = 2$, $L_2^{(2)} = P$ and the recurrence relation $L_{n+2}^{(2)} = P * L_{n+1}^{(2)} - Q * L_n^{(2)}$.

Wraps GAP's Lucas(...)[2].

INPUT:

- `n` - integer
- `P, Q` - integer or rational numbers

OUTPUT: integer or rational number

EXAMPLES:

```
sage: [lucas_number2(i,1,-1) for i in range(10)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
sage: [fibonacci(i-1)+fibonacci(i+1) for i in range(10)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]


sage: n = lucas_number2(5,2,3); n
2
sage: type(n)
<type 'sage.rings.integer.Integer'>
sage: n = lucas_number2(5,2,-3/9); n
418/9
sage: type(n)
<type 'sage.rings.rational.Rational'>
```

The case P=1, Q=-1 is the Lucas sequence in Brualdi's Introductory Combinatorics, 4th ed., Prentice-Hall, 2004:

```
sage: [lucas_number2(n,1,-1) for n in range(10)]
[2, 1, 3, 4, 7, 11, 18, 29, 47, 76]
```

number_of_arrangements (*mset, k*)

Returns the size of arrangements(mset,k). Wraps GAP's NrArrangements.

EXAMPLES:

```
sage: mset = [1,1,2,3,4,4,5]
sage: number_of_arrangements(mset,2)
22
```

number_of_combinations(*mset, k*)

Returns the size of combinations(mset,k). IMPLEMENTATION: Wraps GAP's NrCombinations.

Note: `mset` must be a list of integers or strings (i.e., this is very restricted).

EXAMPLES:

```
sage: mset = [1,1,2,3,4,4,5]
sage: number_of_combinations(mset,2)
12
```

number_of_derangements(*mset*)

Returns the size of derangements(mset). Wraps GAP's NrDerangements.

EXAMPLES:

```
sage: mset = [1,2,3,4]
sage: number_of_derangements(mset)
9
```

number_of_permutations(*mset*)

Do not use this function. It will be deprecated in future version of Sage and eventually removed. Use Permutations instead; instead of

number_of_permutations(mset)

use

Permutations(mset).cardinality().

If you insist on using this now:

Returns the size of permutations(mset).

AUTHORS:

- Robert L. Miller

EXAMPLES:

```
sage: mset = [1,1,2,2,2]
sage: number_of_permutations(mset)
10
```

number_of_tuples(*S, k*)

Returns the size of tuples(S,k). Wraps GAP's NrTuples.

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: number_of_tuples(S,2)
25
sage: S = [1,1,2,3,4,5]
sage: number_of_tuples(S,2)
25
```

number_of_unordered_tuples(*S, k*)

Returns the size of unordered_tuples(S,k). Wraps GAP's NrUnorderedTuples.

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: number_of_unordered_tuples(S,2)
15
```

permutations(*mset*)

A permutation is represented by a list that contains exactly the same elements as mset, but possibly in different order. If mset is a proper set there are $|mset|!$ such permutations. Otherwise if the first elements appears k_1 times, the second element appears k_2 times and so on, the number of permutations is $|mset|!/(k_1!k_2!\ldots)$, which is sometimes called a multinomial coefficient.

permutations returns the set of all permutations of a multiset. Calls a function written by Mike Hansen, not GAP.

EXAMPLES:

```
sage: mset = [1,1,2,2,2]
sage: permutations(mset)
[[1, 1, 2, 2, 2],
 [1, 2, 1, 2, 2],
 [1, 2, 2, 1, 2],
 [1, 2, 2, 2, 1],
 [2, 1, 1, 2, 2],
 [2, 1, 2, 1, 2],
 [2, 1, 2, 2, 1],
 [2, 2, 1, 1, 2],
 [2, 2, 1, 2, 1],
 [2, 2, 2, 1, 1]]
sage: MS = MatrixSpace(GF(2),2,2)
sage: A = MS([1,0,1,1])
sage: permutations(A.rows())
[[(1, 0), (1, 1)], [(1, 1), (1, 0)]]
```

permutations_iterator(*mset, n=None*)

Do not use this function. It will be deprecated in future version of Sage and eventually removed. Use Permutations instead; instead of

for p in permutations_iterator(range(1, m+1), n)

use

for p in Permutations(m, n).

Note that Permutations, unlike this function, treats repeated elements as identical.

If you insist on using this now:

Returns an iterator (http://docs.python.org/lib/typeiter.html) which can be used in place of permutations(mset) if all you need it for is a 'for' loop.

Posted by Raymond Hettinger, 2006/03/23, to the Python Cookbook: http://aspn.activestate.com/ASPN/Cookbook/Python/Recipe/474124

Note- This function considers repeated elements as different entries, so for example:

```
sage: from sage.combinat.combinat import permutations, permutations_iterator
sage: mset = [1,2,2]
sage: permutations(mset)
[[1, 2, 2], [2, 1, 2], [2, 2, 1]]
sage: for p in permutations_iterator(mset): print p
[1, 2, 2]
[1, 2, 2]
[2, 1, 2]
```

```
[2, 2, 1]
[2, 1, 2]
[2, 2, 1]
```

EXAMPLES:

```
sage: X = permutations_iterator(range(3),2)
sage: [x for x in X]
[[0, 1], [0, 2], [1, 0], [1, 2], [2, 0], [2, 1]]
```

stirling_number1(*n, k*)

Returns the n-th Stilling number $S_1(n,k)$ of the first kind (the number of permutations of n points with k cycles). Wraps GAP's Stirling1.

EXAMPLES:

```
sage: stirling_number1(3,2)
3
sage: stirling_number1(5,2)
50
sage: 9*stirling_number1(9,5)+stirling_number1(9,4)
269325
sage: stirling_number1(10,5)
269325
```

Indeed, $S_1(n,k) = S_1(n-1,k-1) + (n-1)S_1(n-1,k)$.

stirling_number2(*n, k*)

Returns the n-th Stirling number $S_2(n,k)$ of the second kind (the number of ways to partition a set of n elements into k pairwise disjoint nonempty subsets). (The n-th Bell number is the sum of the $S_2(n,k)$'s, $k = 0, ..., n$.) Wraps GAP's Stirling2.

EXAMPLES: Stirling numbers satisfy $S_2(n,k) = S_2(n-1,k-1) + kS_2(n-1,k)$:

```
sage: 5*stirling_number2(9,5) + stirling_number2(9,4)
42525
sage: stirling_number2(10,5)
42525
```

```
sage: n = stirling_number2(20,11); n
1900842429486
sage: type(n)
<type 'sage.rings.integer.Integer'>
```

tuples(*S, k*)

An ordered tuple of length k of set is an ordered selection with repetition and is represented by a list of length k containing elements of set. tuples returns the set of all ordered tuples of length k of the set.

EXAMPLES:

```
sage: S = [1,2]
sage: tuples(S,3)
[[1, 1, 1], [2, 1, 1], [1, 2, 1], [2, 2, 1], [1, 1, 2], [2, 1, 2], [1, 2, 2], [2, 2, 2]]
sage: mset = ["s","t","e","i","n"]
sage: tuples(mset,2)
[['s', 's'], ['t', 's'], ['e', 's'], ['i', 's'], ['n', 's'], ['s', 't'], ['t', 't'],
 ['e', 't'], ['i', 't'], ['n', 't'], ['s', 'e'], ['t', 'e'], ['e', 'e'], ['i', 'e'],
 ['n', 'e'], ['s', 'i'], ['t', 'i'], ['e', 'i'], ['i', 'i'], ['n', 'i'], ['s', 'n'],
 ['t', 'n'], ['e', 'n'], ['i', 'n'], ['n', 'n']]
```

The Set(...) comparisons are necessary because finite fields are not enumerated in a standard order.

```
sage: K.<a> = GF(4, 'a')
sage: mset = [x for x in K if x!=0]
sage: tuples(mset,2)
[[a, a], [a + 1, a], [1, a], [a, a + 1], [a + 1, a + 1], [1, a + 1], [a, 1], [a + 1, 1], [1, 1]]
```

AUTHORS:

- Jon Hanke (2006-08)

unordered_tuples(*S*, *k*)

An unordered tuple of length k of set is a unordered selection with repetitions of set and is represented by a sorted list of length k containing elements from set.

unordered_tuples returns the set of all unordered tuples of length k of the set. Wraps GAP's UnorderedTuples.

Warning: Wraps GAP - hence mset must be a list of objects that have string representations that can be interpreted by the GAP intepreter. If mset consists of at all complicated Sage objects, this function does *not* do what you expect. A proper function should be written! (TODO!)

EXAMPLES:

```
sage: S = [1,2]
sage: unordered_tuples(S,3)
[[1, 1, 1], [1, 1, 2], [1, 2, 2], [2, 2, 2]]
sage: unordered_tuples(["a","b","c"],2)
['aa', 'ab', 'ac', 'bb', 'bc', 'cc']
```

17.2 Functions that compute some of the sequences in Sloane's tables

EXAMPLES:

Type sloane.[tab] to see a list of the sequences that are defined.

```
sage: a = sloane.A000005; a
 The integer sequence tau(n), which is the number of divisors of n.
 sage: a(1)
 1
 sage: a(6)
 4
 sage: a(100)
 9
```

Type `d._eval??` to see how the function that computes an individual term of the sequence is implemented.

The input must be a positive integer:

```
sage: a(0)
...
ValueError: input n (=0) must be a positive integer
sage: a(1/3)
...
TypeError: input must be an int, long, or Integer
```

You can also change how a sequence prints:

```
sage: a = sloane.A000005; a
The integer sequence tau(n), which is the number of divisors of n.
sage: a.rename('(..., tau(n), ...)')
sage: a
(..., tau(n), ...)
sage: a.reset_name()
sage: a
The integer sequence tau(n), which is the number of divisors of n.
```

TESTS:

```
sage: a = sloane.A000001;
sage: a == loads(dumps(a))
True
```

AUTHORS:

- William Stein: framework
- Jaap Spies: most sequences
- Nick Alexander: updated framework

class A000001()

class A000004()

class A000005()

class A000007()

class A000008()

class A000009()

cf()
EXAMPLES:

```
sage: it = sloane.A000009.cf()
sage: [it.next() for i in range(14)]
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18]
```

list(*n*)
EXAMPLES:

```
sage: sloane.A000009.list(14)
[1, 1, 1, 2, 2, 3, 4, 5, 6, 8, 10, 12, 15, 18]
```

class A000010()

class A000012()

class A000015()

class A000016()

class A000027()

class A000030()

class A000032()

class A000035()

class A000040()

class A000041()

class A000043()

class A000045()

fib()
Returns a generator over all Fibonacci numbers, starting with 0.
EXAMPLES:

```
sage: it = sloane.A000045.fib()
sage: [it.next() for i in range(10)]
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

list(*n*)
EXAMPLES:

```
sage: sloane.A000045.list(10)
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

class A000069()

class A000073()

list(*n*)
EXAMPLES:

```
sage: sloane.A000073.list(10)
[0, 0, 1, 1, 2, 4, 7, 13, 24, 44]
```

class A000079()

class A000085()

class A000100()

class A000108()

class A000110()

class A000120()

f(*n*)
EXAMPLES:

```
sage: [sloane.A000120.f(n) for n in range(10)]
[0, 1, 1, 2, 1, 2, 2, 3, 1, 2]
```

class A000124()

class A000129()

class A000142()

class A000153()

class A000165()

class A000166()

class A000169()

class **A000203**()

class **A000204**()

class **A000213**()

list(*n*)

EXAMPLES:

```
sage: sloane.A000213.list(10)
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105]
```

class **A000217**()

class **A000225**()

class **A000244**()

class **A000255**()

class **A000261**()

class **A000272**()

class **A000290**()

class **A000292**()

class **A000302**()

class **A000312**()

class **A000326**()

class **A000330**()

class **A000396**()

class **A000578**()

class **A000583**()

class **A000587**()

class **A000668**()

class **A000670**()

class **A000720**()

class **A000796**()

list(*n*)

EXAMPLES:

```
sage: sloane.A000796.list(10)
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
```

pi()

Based on an algorithm of Lambert Meertens The ABC-programming language!!!

EXAMPLES:

```
sage: it = sloane.A000796.pi()
sage: [it.next() for i in range(10)]
[3, 1, 4, 1, 5, 9, 2, 6, 5, 3]
```

class **A000961**()

list(*n*)
EXAMPLES:

```
sage: sloane.A000961.list(10)
[1, 2, 3, 4, 5, 7, 8, 9, 11, 13]
```

class A000984()

class A001006()

class A001045()

class A001055()

nwf(*n*, *m*)
EXAMPLES:

```
sage: sloane.A001055.nwf(4,1)
0
sage: sloane.A001055.nwf(4,2)
1
sage: sloane.A001055.nwf(4,3)
1
sage: sloane.A001055.nwf(4,4)
2
```

class A001109()

class A001110()

g(*k*)
EXAMPLES:

```
sage: sloane.A001110.g(2)
2
sage: sloane.A001110.g(1)
0
```

class A001147()

class A001157()

class A001189()

class A001221()

class A001222()

class A001227()

class A001333()

class A001358()

list(*n*)
EXAMPLES:

```
sage: sloane.A001358.list(9)
[4, 6, 9, 10, 14, 15, 21, 22, 25]
```

class A001405()

class A001477()

class A001694()

is_powerful(*n*)

This function returns True if and only if n is a Powerful Number:

A positive integer n is powerful if for every prime p dividing n, p^2 also divides n. See Sloane's OEIS A001694.

INPUT:

- n - integer

OUTPUT:

- True - if n is a Powerful number, else False

EXAMPLES:

```
sage: a = sloane.A001694
sage: a.is_powerful(2500)
True
sage: a.is_powerful(20)
False
```

AUTHORS:

- Jaap Spies (2006-12-07)

list(*n*)

EXAMPLES:

```
sage: sloane.A001694.list(9)
[1, 4, 8, 9, 16, 25, 27, 32, 36]
```

class A001836()

list(*n*)

EXAMPLES:

```
sage: sloane.A001836.list(9)
[53, 83, 158, 263, 293, 368, 578, 683, 743]
```

class A001906()

class A001909()

class A001910()

class A001969()

class A002110()

class A002113()

list(*n*)

EXAMPLES:

```
sage: sloane.A002113.list(15)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 22, 33, 44, 55]
```

class A002275()

class A002378()

class A002620()

class **A002808**()

list(*n*)
EXAMPLES:

```
sage: sloane.A002808.list(10)
[4, 6, 8, 9, 10, 12, 14, 15, 16, 18]
```

class **A003418**()

class **A004086**()

class **A004526**()

class **A005100**()

list(*n*)
EXAMPLES:

```
sage: sloane.A005100.list(10)
[1, 2, 3, 4, 5, 7, 8, 9, 10, 11]
```

class **A005101**()

list(*n*)
EXAMPLES:

```
sage: sloane.A005101.list(10)
[12, 18, 20, 24, 30, 36, 40, 42, 48, 54]
```

class **A005117**()

list(*n*)
EXAMPLES:

```
sage: sloane.A005117.list(10)
[1, 2, 3, 5, 6, 7, 10, 11, 13, 14]
```

class **A005408**()

class **A005843**()

class **A006318**()

class **A006530**()

class **A006882**()

df()
Double factorials n!!: a(n)=n*a(n-2).
EXAMPLES:

```
sage: it = sloane.A006882.df()
sage: [it.next() for i in range(10)]
[1, 1, 2, 3, 8, 15, 48, 105, 384, 945]
```

list(*n*)
EXAMPLES:

```
sage: sloane.A006882.list(10)
[1, 1, 2, 3, 8, 15, 48, 105, 384, 945]
```

class A007318()

class A008275()

s(*n*, *k*)

EXAMPLES:

```
sage: sloane.A008275.s(4,2)
11
sage: sloane.A008275.s(5,2)
-50
sage: sloane.A008275.s(5,3)
35
```

class A008277()

s2(*n*, *k*)

Returns the Stirling number S2(n,k) of the 2nd kind.

EXAMPLES:

```
sage: sloane.A008277.s2(4,2)
7
```

class A008683()

class A010060()

class A015521()

class A015523()

class A015530()

class A015531()

class A015551()

class A018252()

class A020639()

list(*n*)

EXAMPLES:

```
sage: sloane.A020639.list(10)
[1, 2, 3, 2, 5, 2, 7, 2, 3, 2]
```

class A046660(*offset=1*)

Excess of n = number of prime divisors (with multiplicity) - number of prime divisors (without multiplicity).

$\Omega(n) - \omega(n)$.

INPUT:

- `n` - positive integer

OUTPUT:

- `integer` - function value

EXAMPLES:

```
sage: a = sloane.A046660; a
Excess of n = Bigomega (with multiplicity) - omega (without multiplicity).
sage: a(0)
...
ValueError: input n (=0) must be a positive integer
sage: a(1)
0
sage: a(8)
2
sage: a(41)
0
sage: a(84792)
2
sage: a.list(12)
[0, 0, 0, 1, 0, 0, 0, 2, 1, 0, 0, 1]
```

AUTHORS:

- Jaap Spies (2007-01-19)

class A049310()

class A051959()

g(*k*)

EXAMPLES:

```
sage: sloane.A051959.g(2)
15
sage: sloane.A051959.g(1)
0
```

class A055790()

class A061084()

class A064553()

class A079922(*offset=1*)

function returns solutions to the Dancing School problem with n girls and $n+3$ boys.

The value is $per(B)$, the permanent of the (0,1)-matrix B of size $n \times n+3$ with $b(i,j) = 1$ if and only if $i \leq j \leq i+n$.

REFERENCES:

- Jaap Spies, Nieuw Archief voor Wiskunde, 5/7 nr 4, December 2006

INPUT:

- n - positive integer

OUTPUT:

- integer - function value

EXAMPLES:

```
sage: a = sloane.A079922; a
Solutions to the Dancing School problem with n girls and n+3 boys
sage: a.offset
1
sage: a(1)
4
sage: a(8)
2227
sage: a.list(8)
[4, 13, 36, 90, 212, 478, 1044, 2227]
```

Compare: Searching Sloane's online database... Solution to the Dancing School Problem with n girls and n+3 boys: f(n,3). [4, 13, 36, 90, 212, 478, 1044, 2227]

```
sage: a(-1)
...
ValueError: input n (=-1) must be a positive integer
```

AUTHORS:

- Jaap Spies (2007-01-14)

class A079923(*offset=1*)

function returns solutions to the Dancing School problem with n girls and $n + 4$ boys.

The value is $per(B)$, the permanent of the (0,1)-matrix B of size $n \times n + 3$ with $b(i, j) = 1$ if and only if $i \leq j \leq i + n$.

REFERENCES:

- Jaap Spies, Nieuw Archief voor Wiskunde, 5/7 nr 4, December 2006

INPUT:

- `n` - positive integer

OUTPUT:

- `integer` - function value

EXAMPLES:

```
sage: a = sloane.A079923; a
Solutions to the Dancing School problem with n girls and n+4 boys
sage: a.offset
1
sage: a(1)
5
sage: a(8)
15458
sage: a.list(8)
[5, 21, 76, 246, 738, 2108, 5794, 15458]
```

Compare: Searching Sloane's online database... Solution to the Dancing School Problem with n girls and n+4 boys: f(n,4). [5, 21, 76, 246, 738, 2108, 5794, 15458]

```
sage: a(0)
...
ValueError: input n (=0) must be a positive integer
```

AUTHORS:

•Jaap Spies (2007-01-17)

class A082411()

class A083103()

class A083104()

class A083105()

class A083216()

class A090010()

class A090012()

class A090013()

class A090014()

class A090015()

class A090016()

class A111774()

is_number_of_the_third_kind(n)

This function returns True if and only if n is a number of the third kind.

A number is of the third kind if it can be written as a sum of at least three consecutive positive integers. Odd primes can only be written as a sum of two consecutive integers. Powers of 2 do not have a representation as a sum of k consecutive integers (other than the trivial $n = n$ for $k = 1$).

See: http://www.jaapspies.nl/mathfiles/problem2005-2C.pdf

INPUT:

•n - positive integer

OUTPUT:

•True - if n is not prime and not a power of 2 False -

EXAMPLES:

```
sage: a = sloane.A111774
sage: a.is_number_of_the_third_kind(6)
True
sage: a.is_number_of_the_third_kind(100)
True
sage: a.is_number_of_the_third_kind(16)
False
sage: a.is_number_of_the_third_kind(97)
False
```

AUTHORS:

•Jaap Spies (2006-12-09)

list(n)

EXAMPLES:

```
sage: sloane.A111774.list(12)
[6, 9, 10, 12, 14, 15, 18, 20, 21, 22, 24, 25]
```

class A111775()

class A111776()

class A111787()

class **ExponentialNumbers**(*a*)

class **ExtremesOfPermanentsSequence**(*offset=1*)

gen(*a0, a1, d*)
EXAMPLES:

```
sage: it = sloane.A000153.gen(0,1,2)
sage: [it.next() for i in range(5)]
[0, 1, 2, 7, 32]
```

list(*n*)
EXAMPLES:

```
sage: sloane.A000153.list(8)
[0, 1, 2, 7, 32, 181, 1214, 9403]
```

class **ExtremesOfPermanentsSequence2**(*offset=1*)

gen(*a0, a1, d*)
EXAMPLES:

```
sage: from sage.combinat.sloane_functions import ExtremesOfPermanentsSequence2
sage: e = ExtremesOfPermanentsSequence2()
sage: it = e.gen(6,43,6)
sage: [it.next() for i in range(5)]
[6, 43, 307, 2542, 23799]
```

class **RecurrenceSequence**(*offset=1*)

list(*n*)
EXAMPLES:

```
sage: sloane.A001110.list(8)
[0, 1, 36, 1225, 41616, 1413721, 48024900, 1631432881]
```

class **RecurrenceSequence2**(*offset=1*)

list(*n*)
EXAMPLES:

```
sage: sloane.A001906.list(10)
[0, 1, 3, 8, 21, 55, 144, 377, 987, 2584]
```

class **Sloane**()
A collection of Sloane generating functions.

This class inspects sage.combinat.sloane_functions, accumulating all the SloaneSequence classes starting with 'A'. These are listed for tab completion, but not instantiated until requested.

EXAMPLES: Ensure we have lots of entries:

```
sage: len(sloane.trait_names()) > 100
True
```

And ensure none are being incorrectly returned:

```
sage: [ None for n in sloane.trait_names() if not n.startswith('A') ]
[]
```

Ensure we can access dynamic constructions and cache correctly:

```
sage: s = sloane.A000587
sage: s is sloane.A000587
True
```

And that we can access other functions in parent classes:

```
sage: sloane.__class__
<class 'sage.combinat.sloane_functions.Sloane'>
```

AUTHORS:

- Nick Alexander

trait_names()

List Sloane generating functions for tab-completion. The member classes are inspected from module sage.combinat.sloane_functions.

They must be sub classes of SloaneSequence and must start with 'A'. These restrictions are only to prevent typos, incorrect inspecting, etc.

EXAMPLES:

```
sage: type(sloane.trait_names())
<type 'list'>
```

class SloaneSequence(*offset=1*)

Base class for a Sloane integer sequence.

EXAMPLES:

We create a dummy sequence:

list(*n*)

Return n terms of the sequence: sequence[offset], sequence[offset+1], ... , sequence[offset+n-1]. EXAMPLES:

```
sage: sloane.A000012.list(4)
[1, 1, 1, 1]
```

perm_mh(*m, h*)

This functions calculates $f(g,h)$ from Sloane's sequences A079908-A079928

INPUT:

- m - positive integer
- h - non negative integer

OUTPUT: permanent of the m x (m+h) matrix, etc.

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import perm_mh
sage: perm_mh(3,3)
36
sage: perm_mh(3,4)
76
```

AUTHORS:

- Jaap Spies (2006)

recur_gen2 (*a0, a1, a2, a3*)

homogenous general second-order linear recurrence generator with fixed coefficients

a(0) = a0, a(1) = a1, a(n) = a2*a(n-1) + a3*a(n-2)

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import recur_gen2
sage: it = recur_gen2(1,1,1,1)
sage: [it.next() for i in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

recur_gen2b (*a0, a1, a2, a3, b*)

inhomogenous second-order linear recurrence generator with fixed coefficients and $b = f(n)$

$a(0) = a0$, $a(1) = a1$, $a(n) = a2 * a(n-1) + a3 * a(n-2) + f(n)$.

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import recur_gen2b
sage: it = recur_gen2b(1,1,1,1, lambda n: 0)
sage: [it.next() for i in range(10)]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

recur_gen3 (*a0, a1, a2, a3, a4, a5*)

homogenous general third-order linear recurrence generator with fixed coefficients

a(0) = a0, a(1) = a1, a(2) = a2, a(n) = a3*a(n-1) + a4*a(n-2) + a5*a(n-3)

EXAMPLES:

```
sage: from sage.combinat.sloane_functions import recur_gen3
sage: it = recur_gen3(1,1,1,1,1,1)
sage: [it.next() for i in range(10)]
[1, 1, 1, 3, 5, 9, 17, 31, 57, 105]
```

17.3 Compute Bell and Uppuluri-Carpenter numbers.

AUTHORS:

- Nick Alexander

clear_mpz_globals()

expnums()

Compute the first n exponential numbers around aa, starting with the zero-th.

INPUT:

- `n` - C machine int
- `aa` - C machine int

OUTPUT: A list of length n.

ALGORITHM: We use the same integer addition algorithm as GAP. This is an extension of Bell's triangle to the general case of exponential numbers. The recursion performs $O(n^2)$ additions, but the running time is dominated by the cost of the last integer addition, because the growth of the integer results of partial computations is exponential in n. The algorithm stores $O(n)$ integers, but each is exponential in n.

EXAMPLES:

```
sage: expnums(10, 1)
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]

sage: expnums(10, -1)
[1, -1, 0, 1, 1, -2, -9, -9, 50, 267]

sage: expnums(1, 1)
[1]
sage: expnums(0, 1)
[]
sage: expnums(-1, 0)
[]
```

AUTHORS:

- Nick Alexander

expnums2()

A vanilla python (but compiled via pyrex) implementation of expnums.

We Compute the first n exponential numbers around aa, starting with the zero-th.

EXAMPLES:

```
sage: from sage.combinat.expnums import expnums2
sage: expnums2(10, 1)
[1, 1, 2, 5, 15, 52, 203, 877, 4140, 21147]
```

gmp_randrange()

init_mpz_globals()

17.4 Alternating Sign Matrices

AlternatingSignMatrices(n)

Returns the combinatorial class of alternating sign matrices of size n.

EXAMPLES:

```
sage: a2 = AlternatingSignMatrices(2); a2
Alternating sign matrices of size 2
sage: for a in a2: print a, "-\n"
[0 1]
[1 0]
-
[1 0]
[0 1]
-
```

class **AlternatingSignMatrices_n**(n)

cardinality()

TESTS:

```
sage: [ AlternatingSignMatrices(n).cardinality() for n in range(0, 11)]
[1, 1, 2, 7, 42, 429, 7436, 218348, 10850216, 911835460, 129534272700]
```

```
sage: asms = [ AlternatingSignMatrices(n) for n in range(6) ]
sage: all( [ asm.cardinality() == len(asm.list()) for asm in asms] )
True
```

ContreTableaux(*n*)

Returns the combinatorial class of contre tableaux of size n.

EXAMPLES:

```
sage: ct4 = ContreTableaux(4); ct4
Contre tableaux of size 4
sage: ct4.cardinality()
42
sage: ct4.first()
[[1, 2, 3, 4], [1, 2, 3], [1, 2], [1]]
sage: ct4.last()
[[1, 2, 3, 4], [2, 3, 4], [3, 4], [4]]
sage: ct4.random_element()
[[1, 2, 3, 4], [1, 2, 3], [1, 3], [3]]
```

class **ContreTableaux_n**(*n*)

cardinality()

TESTS:

```
sage: [ ContreTableaux(n).cardinality() for n in range(0, 11)]
[1, 1, 2, 7, 42, 429, 7436, 218348, 10850216, 911835460, 129534272700]
```

TruncatedStaircases(*n, last_column*)

Returns the combinatorial class of truncated staircases of size n with last column last_column.

EXAMPLES:

```
sage: t4 = TruncatedStaircases(4, [2,3]); t4
Truncated staircases of size 4 with last column [2, 3]
sage: t4.cardinality()
4
sage: t4.first()
[[4, 3, 2, 1], [3, 2, 1], [3, 2]]
sage: t4.list()
[[[4, 3, 2, 1], [3, 2, 1], [3, 2]], [[4, 3, 2, 1], [4, 2, 1], [3, 2]], [[4, 3, 2, 1], [4, 3, 1],
```

class **TruncatedStaircases_nlastcolumn**(*n, last_column*)

from_contre_tableau(*comps*)

Returns an alternating sign matrix from a contretableaux.

TESTS:

```
sage: import sage.combinat.alternating_sign_matrix as asm
sage: asm.from_contre_tableau([[1, 2, 3], [1, 2], [1]])
[0 0 1]
[0 1 0]
[1 0 0]
sage: asm.from_contre_tableau([[1, 2, 3], [2, 3], [3]])
[1 0 0]
[0 1 0]
[0 0 1]
```

17.5 Cartesian Products

CartesianProduct(**iters*)

Returns the combinatorial class of the cartesian product of *iters.

EXAMPLES:

```
sage: cp = CartesianProduct([1,2], [3,4]); cp
Cartesian product of [1, 2], [3, 4]
sage: cp.list()
[[1, 3], [1, 4], [2, 3], [2, 4]]
```

Note that if you have a generator-type object that is returned by a function, then you should use IterableFunctionCall class defined in sage.combinat.misc.

```
sage: def a(n): yield 1*n; yield 2*n
sage: def b(): yield 'a'; yield 'b'
sage: CartesianProduct(a(3), b()).list()
[[3, 'a'], [3, 'b']]
sage: from sage.combinat.misc import IterableFunctionCall
sage: CartesianProduct(IterableFunctionCall(a, 3), IterableFunctionCall(b)).list()
[[3, 'a'], [3, 'b'], [6, 'a'], [6, 'b']]
```

See the documentation for IterableFunctionCall for more information.

class **CartesianProduct_iters**(**iters*)

cardinality()

Returns the number of elements in the cartesian product of everything in *iters.

EXAMPLES:

```
sage: CartesianProduct(range(2), range(3)).cardinality()
6
sage: CartesianProduct(range(2), xrange(3)).cardinality()
6
sage: CartesianProduct(range(2), xrange(3), xrange(4)).cardinality()
24
```

list()

Returns

EXAMPLES:

```
sage: CartesianProduct(range(3), range(3)).list()
[[0, 0], [0, 1], [0, 2], [1, 0], [1, 1], [1, 2], [2, 0], [2, 1], [2, 2]]
sage: CartesianProduct('dog', 'cat').list()
[['d', 'c'],
 ['d', 'a'],
 ['d', 't'],
 ['o', 'c'],
 ['o', 'a'],
 ['o', 't'],
 ['g', 'c'],
 ['g', 'a'],
 ['g', 't']]
```

random_element()

Returns a random element from the cartesian product of *iters.

EXAMPLES:

```
sage: CartesianProduct('dog', 'cat').random_element()
['d', 'a']
```

17.6 Combinations

Combinations(*mset, k=None*)

Returns the combinatorial class of combinations of mset. If k is specified, then it returns the combinatorial class of combinations of mset of size k.

The combinatorial classes correctly handle the cases where mset has duplicate elements.

EXAMPLES:

```
sage: C = Combinations(range(4)); C
Combinations of [0, 1, 2, 3]
sage: C.list()
[[],
 [0],
 [1],
 [2],
 [3],
 [0, 1],
 [0, 2],
 [0, 3],
 [1, 2],
 [1, 3],
 [2, 3],
 [0, 1, 2],
 [0, 1, 3],
 [0, 2, 3],
 [1, 2, 3],
 [0, 1, 2, 3]]
 sage: C.cardinality()
 16
```

```
sage: C2 = Combinations(range(4),2); C2
Combinations of [0, 1, 2, 3] of length 2
sage: C2.list()
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
sage: C2.cardinality()
6
```

```
sage: Combinations([1,2,2,3]).list()
[[],
 [1],
 [2],
 [3],
 [1, 2],
 [1, 3],
 [2, 2],
 [2, 3],
 [1, 2, 2],
 [1, 2, 3],
 [2, 2, 3],
 [1, 2, 2, 3]]
```

class **Combinations_mset**(*mset*)

cardinality()
TESTS:

```
sage: Combinations([1,2,3]).cardinality()
8
sage: Combinations(['a','a','b']).cardinality()
6
```

class **Combinations_msetk**(*mset, k*)

cardinality()
Returns the size of combinations(mset,k). IMPLEMENTATION: Wraps GAP's NrCombinations.
EXAMPLES:

```
sage: mset = [1,1,2,3,4,4,5]
sage: Combinations(mset,2).cardinality()
12
```

class **Combinations_set**(*mset*)

rank(*x*)
EXAMPLES:

```
sage: c = Combinations([1,2,3])
sage: range(c.cardinality()) == map(c.rank, c)
True
```

unrank(*r*)
EXAMPLES:

```
sage: c = Combinations([1,2,3])
sage: c.list() == map(c.unrank, range(c.cardinality()))
True
```

class **Combinations_setk**(*mset, k*)

list()
EXAMPLES:

```
sage: Combinations([1,2,3,4,5],3).list()
[[1, 2, 3],
 [1, 2, 4],
 [1, 2, 5],
 [1, 3, 4],
 [1, 3, 5],
 [1, 4, 5],
 [2, 3, 4],
 [2, 3, 5],
 [2, 4, 5],
 [3, 4, 5]]
```

rank(*x*)
EXAMPLES:

```
sage: c = Combinations([1,2,3], 2)
sage: range(c.cardinality()) == map(c.rank, c.list())
True
```

unrank(*r*)

EXAMPLES:

```
sage: c = Combinations([1,2,3], 2)
sage: c.list() == map(c.unrank, range(c.cardinality()))
True
```

17.7 Signed Compositions

SignedCompositions(*n*)

Returns the combinatorial class of signed compositions of n.

EXAMPLES:

```
sage: SC3 = SignedCompositions(3); SC3
Signed compositions of 3
sage: SC3.cardinality()
18
sage: len(SC3.list())
18
sage: SC3.first()
[1, 1, 1]
sage: SC3.last()
[-3]
sage: SC3.random_element()
[1, -1, 1]
sage: SC3.list()
[[1, 1, 1],
 [1, 1, -1],
 [1, -1, 1],
 [1, -1, -1],
 [-1, 1, 1],
 [-1, 1, -1],
 [-1, -1, 1],
 [-1, -1, -1],
 [1, 2],
 [1, -2],
 [-1, 2],
 [-1, -2],
 [2, 1],
 [2, -1],
 [-2, 1],
 [-2, -1],
 [3],
 [-3]]
```

class **SignedCompositions_n**(*n*)

cardinality()

TESTS:

```
sage: SC4 = SignedCompositions(4)
sage: SC4.cardinality() == len(SC4.list())
True
sage: SignedCompositions(3).cardinality()
18
```

17.8 Compositions

A composition c of a nonnegative integer n is a list of positive integers with total sum n.

EXAMPLES: There are 8 compositions of 4.

```
sage: Compositions(4).cardinality()
8
```

Here is the list of them:

```
sage: Compositions(4).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
```

You can use the .first() method to get the 'first' composition of a number.

```
sage: Compositions(4).first()
[1, 1, 1, 1]
```

You can also calculate the 'next' composition given the current one.

```
sage: Compositions(4).next([1,1,2])
[1, 2, 1]
```

The following examples shows how to test whether or not an object is a composition.

```
sage: [3,4] in Compositions()
True
sage: [3,4] in Compositions(7)
True
sage: [3,4] in Compositions(5)
False
```

Similarly, one can check whether or not an object is a composition which satisfies further constraints.

```
sage: [4,2] in Compositions(6, inner=[2,2], min_part=2)
True
sage: [4,2] in Compositions(6, inner=[2,2], min_part=2)
True
sage: [4,2] in Compositions(6, inner=[2,2], min_part=3)
False
```

Note that the given constraints should compatible.

```
sage: [4,1] in Compositions(5, inner=[2,1], min_part=1)
True
```

The options length, min_length, and max_length can be used to set length constraints on the compositions. For example, the compositions of 4 of length equal to, at least, and at most 2 are given by:

```
sage: Compositions(4, length=2).list()
[[1, 3], [2, 2], [3, 1]]
sage: Compositions(4, min_length=2).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1]]
```

```
sage: Compositions(4, max_length=2).list()
[[1, 3], [2, 2], [3, 1], [4]]
```

Setting both min_length and max_length to the same value is equivalent to setting length to this value.

```
sage: Compositions(4, min_length=2, max_length=2).list()
[[1, 3], [2, 2], [3, 1]]
```

The options inner and outer can be used to set part-by-part containment constraints. The list of compositions of 4 bounded above by [3,1,2] is given by:

```
sage: Compositions(4, outer=[3,1,2]).list()
[[1, 1, 2], [2, 1, 1], [3, 1]]
```

Outer sets max_length to the length of its argument. Moreover, the parts of outer may be infinite to clear the constraint on specific parts. This is the list of compositions of 4 of length at most 3 such that the first and third parts are at most 1:

```
sage: Compositions(4, outer=[1,oo,1]).list()
[[1, 2, 1], [1, 3]]
```

This is the list of compositions of 4 bounded below by [1,1,1].

```
sage: Compositions(4, inner=[1,1,1]).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1]]
```

The options min_slope and max_slope can be used to set constraints on the slope, that is the difference p[i+1]-p[i] of two consecutive parts. The following is the list of weakly increasing compositions of 4.

```
sage: Compositions(4, min_slope=0).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 3], [2, 2], [4]]
```

The following is the list of compositions of 4 such that two consecutive parts differ by at most one unit:

```
sage: Compositions(4, min_slope=-1, max_slope=1).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 2], [4]]
```

The constraints can be combinat together in all reasonable ways. This is the list of compositions of 5 of length between 2 and 4 such that the difference between consecutive parts is between -2 and 1.

```
sage: Compositions(5, max_slope=1, min_slope=-2, min_length=2, max_length=4).list()
[[1, 1, 1, 2],
 [1, 1, 2, 1],
 [1, 2, 1, 1],
 [1, 2, 2],
 [2, 1, 1, 1],
 [2, 1, 2],
 [2, 2, 1],
 [2, 3],
 [3, 1, 1],
 [3, 2]]
```

We can do the same thing with an outer constraint:

```
sage: Compositions(5, max_slope=1, min_slope=-2, min_length=2, max_length=4, outer=[2,5,2]).list()
[[1, 2, 2], [2, 1, 2], [2, 2, 1], [2, 3]]
```

However, providing incoherent constraints may yield strange results. It is up to the user to ensure that the inner and outer compositions themselves satisfy the parts and slope constraints.

AUTHORS:

- Mike Hansen
- MuPAD-Combinat developers (for algorithms and design inspiration)

Composition(*co=None, descents=None, code=None*)

Returns a composition object.

EXAMPLES: The standard way to create a composition is by specifying it as a list.

```
sage: Composition([3,1,2])
[3, 1, 2]
```

You can create a composition from a list of its descents.

```
sage: Composition([1, 1, 3, 4, 3]).descents()
[0, 1, 4, 8, 11]
sage: Composition(descents=[1,0,4,8,11])
[1, 1, 3, 4, 3]
```

You can also create a composition from its code.

```
sage: Composition([4,1,2,3,5]).to_code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: Composition(code=_)
[4, 1, 2, 3, 5]
sage: Composition([3,1,2,3,5]).to_code()
[1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: Composition(code=_)
[3, 1, 2, 3, 5]
```

class **Composition_class**(*l*)

complement()

Returns the complement of the composition co. The complement is the reverse of co's conjugate composition.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).conjugate()
[1, 1, 3, 3, 1, 3]
sage: Composition([1, 1, 3, 1, 2, 1, 3]).complement()
[3, 1, 3, 3, 1, 1]
```

conjugate()

Returns the conjugate of the composition comp.

Algorithm from mupad-combinat.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).conjugate()
[1, 1, 3, 3, 1, 3]
```

descents(*final_descent=False*)

Returns the list of descents of the composition co.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).descents()
[0, 1, 4, 5, 7, 8, 11]
```

is_finer(*co2*)

Returns True if the composition self is finer than the composition co2; otherwise, it returns False.

EXAMPLES:

```
sage: Composition([4,1,2]).is_finer([3,1,3])
False
sage: Composition([3,1,3]).is_finer([4,1,2])
False
sage: Composition([1,2,2,1,1,2]).is_finer([5,1,3])
True
sage: Composition([2,2,2]).is_finer([4,2])
True
```

major_index()

Returns the major index of the composition co. The major index is defined as the sum of the descents.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).major_index()
31
```

peaks()

Returns a list of the peaks of the composition self. The peaks of a composition are the descents which do not immediately follow another descent.

EXAMPLES:

```
sage: Composition([1, 1, 3, 1, 2, 1, 3]).peaks()
[4, 7]
```

refinement(*co2*)

Returns the refinement composition of self and co2.

EXAMPLES:

```
sage: Composition([1,2,2,1,1,2]).refinement([5,1,3])
[3, 1, 2]
```

to_code()

Returns the code of the composition self. The code of a composition is a list of length self.size() of 1s and 0s such that there is a 1 wherever a new part starts.

EXAMPLES:

```
sage: Composition([4,1,2,3,5]).to_code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
```

to_skew_partition(*overlap=1*)

Returns the skew partition obtained from the composition co. The parameter overlap indicates the number of boxes that are covered by boxes of the previous line.

EXAMPLES:

```
sage: Composition([3,4,1]).to_skew_partition()
[[6, 6, 3], [5, 2]]
sage: Composition([3,4,1]).to_skew_partition(overlap=0)
[[8, 7, 3], [7, 3]]
```

Compositions(*n=None, **kwargs*)

Returns the combinatorial class of compositions.

EXAMPLES: If n is not specified, it returns the combinatorial class of all (non-negative) integer compositions.

```
sage: Compositions()
Compositions of non-negative integers
sage: [] in Compositions()
True
sage: [2,3,1] in Compositions()
True
sage: [-2,3,1] in Compositions()
False
```

If n is specified, it returns the class of compositions of n.

```
sage: Compositions(3)
Compositions of 3
sage: Compositions(3).list()
[[1, 1, 1], [1, 2], [2, 1], [3]]
sage: Compositions(3).cardinality()
4
```

In addition, the following constraints can be put on the compositions: length, min_part, max_part, min_length, max_length, min_slope, max_slope, inner, and outer. For example,

```
sage: Compositions(3, length=2).list()
[[1, 2], [2, 1]]
sage: Compositions(4, max_slope=0).list()
[[1, 1, 1, 1], [2, 1, 1], [2, 2], [3, 1], [4]]
```

class **Compositions_all**()

object_class()

class **Compositions_constraints**(*n, **kwargs*)

cardinality()

EXAMPLES:

```
sage: Compositions(4, length=2).cardinality()
3
sage: Compositions(4, min_length=2).cardinality()
7
sage: Compositions(4, max_length=2).cardinality()
4
sage: Compositions(4, max_part=2).cardinality()
5
sage: Compositions(4, min_part=2).cardinality()
2
sage: Compositions(4, outer=[3,1,2]).cardinality()
3
```

list()

Returns a list of all the compositions of n.

EXAMPLES:

```
sage: Compositions(4, length=2).list()
[[1, 3], [2, 2], [3, 1]]
sage: Compositions(4, min_length=2).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1]]
sage: Compositions(4, max_length=2).list()
[[1, 3], [2, 2], [3, 1], [4]]
sage: Compositions(4, max_part=2).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 2]]
sage: Compositions(4, min_part=2).list()
[[2, 2], [4]]
sage: Compositions(4, outer=[3,1,2]).list()
[[1, 1, 2], [2, 1, 1], [3, 1]]
sage: Compositions(4, outer=[1,'inf',1]).list()
[[1, 2, 1], [1, 3]]
sage: Compositions(4, inner=[1,1,1]).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1]]
sage: Compositions(4, min_slope=0).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 3], [2, 2], [4]]
sage: Compositions(4, min_slope=-1, max_slope=1).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 2], [4]]
sage: Compositions(5, max_slope=1, min_slope=-2, min_length=2, max_length=4).list()
[[1, 1, 1, 2],
 [1, 1, 2, 1],
 [1, 2, 1, 1],
 [1, 2, 2],
 [2, 1, 1, 1],
 [2, 1, 2],
 [2, 2, 1],
 [2, 3],
 [3, 1, 1],
 [3, 2]]
sage: Compositions(5, max_slope=1, min_slope=-2, min_length=2, max_length=4, outer=[2,5,2]).
[[1, 2, 2], [2, 1, 2], [2, 2, 1], [2, 3]]
```

object_class()

class **Compositions_n**(*n*)

cardinality()

TESTS:

```
sage: Compositions(3).cardinality()
4
sage: Compositions(0).cardinality()
1
```

list()

TESTS:

```
sage: Compositions(4).list()
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
sage: Compositions(0).list()
[[]]
```

from_code(*code*)

Return the composition from its code.The code of a composition is a list of length self.size() of 1s and 0s such that there is a 1 wherever a new part starts.

EXAMPLES:

```
sage: import sage.combinat.composition as composition
sage: Composition([4,1,2,3,5]).to_code()
[1, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: composition.from_code(_)
[4, 1, 2, 3, 5]
sage: Composition([3,1,2,3,5]).to_code()
[1, 0, 0, 1, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0]
sage: composition.from_code(_)
[3, 1, 2, 3, 5]
```

from_descents (*descents, nps=None*)

Returns a composition from the list of descents.

EXAMPLES:

```
sage: Composition([1, 1, 3, 4, 3]).descents()
[0, 1, 4, 8, 11]
sage: sage.combinat.composition.from_descents([1,0,4,8],12)
[1, 1, 3, 4, 3]
sage: sage.combinat.composition.from_descents([1,0,4,8,11])
[1, 1, 3, 4, 3]
```

17.9 Exact Cover Problem via Dancing Links

AllExactCovers (*M*)

Utilizes A. Ajanki's DLXMatrix class to solve the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])  #no exact covers
sage: for cover in AllExactCovers(M):
...       print cover
sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]]) #two exact covers
sage: for cover in AllExactCovers(M):
...       print cover
[(1, 1, 0), (0, 0, 1)]
[(1, 0, 1), (0, 1, 0)]
```

class DLXMatrix (*ones, initialsolution=None*)

next ()

Search for the first solution we can find, and return it.

Knuth describes the Dancing Links algorithm recursively, though actually implementing it as a recursive algorithm is permissible only for highly restricted problems. (for example, the original author implemented this for Sudoku, and it works beautifully there)

What follows is an iterative description of DLX:

```
stack <- [(NULL)]
level <- 0
while level >= 0:
    cur <- stack[level]
    if cur = NULL:
        if R[h] = h:
            level <- level - 1
```

```
                yield solution
            else:
                cover(best_column)
                stack[level] = best_column
        else if D[cur] != C[cur]:
            if cur != C[cur]:
                delete solution[level]
                for j in L[cur], L[L[cur]], ... , while j != cur:
                    uncover(C[j])
            cur <- D[cur]
            solution[level] <- cur
            stack[level] <- cur
            for j in R[cur], R[R[cur]], ... , while j != cur:
                cover(C[j])
            level <- level + 1
            stack[level] <- (NULL)
        else:
            if C[cur] != cur:
                delete solution[level]
                for j in L[cur], L[L[cur]], ... , while j != cur:
                    uncover(C[j])
            uncover(cur)
            level <- level - 1
```

TESTS:

```
sage: from sage.combinat.dlx import *
sage: M = DLXMatrix([[1,[1,2]],[2,[2,3]],[3,[1,3]]])
sage: while 1:
...     try:
...         C = M.next()
...     except StopIteration:
...         print "StopIteration"
...         break
...     print C
StopIteration
sage: M = DLXMatrix([[1,[1,2]],[2,[2,3]],[3,[3]]])
sage: for C in M:
...       print C
[1, 3]
sage: M = DLXMatrix([[1,[1]],[2,[2,3]],[3,[2]],[4,[3]]])
sage: for C in M:
...       print C
[1, 2]
[1, 3, 4]
```

OneExactCover(*M*)

Utilizes A. Ajanki's DLXMatrix class to solve the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])  #no exact covers
sage: print OneExactCover(M)
None
sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]]) #two exact covers
sage: print OneExactCover(M)
[(1, 1, 0), (0, 0, 1)]
```

17.10 Dancing links C++ wrapper

AllExactCovers(*M*)

Solves the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES: No exact covers:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])
sage: print [cover for cover in AllExactCovers(M)]
[]
```

Two exact covers:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]])
sage: print [cover for cover in AllExactCovers(M)]
[[(1, 1, 0), (0, 0, 1)], [(1, 0, 1), (0, 1, 0)]]
```

DLXCPP(*rows*)

Solves the Exact Cover problem by using the Dancing Links algorithm described by Knuth.

Consider a matrix M with entries of 0 and 1, and compute a subset of the rows of this matrix which sum to the vector of all 1's.

The dancing links algorithm works particularly well for sparse matrices, so the input is a list of lists of the form:

```
[
 [i_11,i_12,...,i_1r]
 ...
 [i_m1,i_m2,...,i_ms]
]
```

where M[j][i_jk] = 1.

The first example below corresponds to the matrix:

```
1110
1010
0100
0001
```

which is exactly covered by:

```
1110
0001
```

and

```
1010
0100
0001
```

If soln is a solution given by DLXCPP(rows) then

[rows[soln[0]], rows[soln[1]], ... rows[soln[len(soln)-1]]]

is an exact cover.

Solutions are given as a list

EXAMPLES:

```
sage: rows = [[0,1,2]]
sage: rows+= [[0,2]]
sage: rows+= [[1]]
sage: rows+= [[3]]
sage: print [ x for x in DLXCPP(rows) ]
[[3, 0], [3, 1, 2]]
```

OneExactCover(*M*)

Solves the exact cover problem on the matrix M (treated as a dense binary matrix).

EXAMPLES:

```
sage: M = Matrix([[1,1,0],[1,0,1],[0,1,1]])  #no exact covers
sage: print OneExactCover(M)
None
sage: M = Matrix([[1,1,0],[1,0,1],[0,0,1],[0,1,0]]) #two exact covers
sage: print OneExactCover(M)
[(1, 1, 0), (0, 0, 1)]
```

17.11 Dyck Words

AUTHORS:

- Mike Hansen
- Dan Drake (2008-05-30): DyckWordBacktracker support

DyckWord(*dw=None, noncrossing_partition=None*)

Returns a Dyck word object

EXAMPLES:

```
sage: dw = DyckWord([1, 0, 1, 0]); dw
[1, 0, 1, 0]
sage: print dw
()()
sage: print dw.height()
1
sage: dw.to_noncrossing_partition()
[[1], [2]]
```

```
sage: DyckWord('()()')
[1, 0, 1, 0]
```

```
sage: DyckWord(noncrossing_partition=[[1],[2]])
[1, 0, 1, 0]
```

class DyckWordBacktracker(*k1, k2*)

DyckWordBacktracker: this class is an iterator for all Dyck words with n opening parentheses and n - endht closing parentheses using the backtracker class. It is used by the DyckWords_size class.

This is not really meant to be called directly, partially because it fails in a couple corner cases: DWB(0) yields [0], not the empty word, and DWB(k, k+1) yields something (it shouldn't yield anything). This could be fixed with a sanity check in _rec(), but then we'd be doing the sanity check *every time* we generate new objects; instead, we do *one* sanity check in DyckWords and assume here that the sanity check has already been made.

AUTHOR:

- Dan Drake (2008-05-30)

class DyckWord_class(l)

a_statistic()

Returns the a-statistic for the Dyck word.

One can view a balanced Dyck word as a lattice path from $(0,0)$ to (n,n) in the first quadrant by letting '1's represent steps in the direction $(1,0)$ and '0's represent steps in the direction $(0,1)$. The resulting path will remain weakly above the diagonal $y = x$.

The a-statistic, or area statistic, is the number of complete squares in the integer lattice which are below the path and above the line $y = x$. The 'half-squares' directly above the line $y = x$ do not contribute to this statistic.

EXAMPLES:

```
sage: dw = DyckWord([1,0,1,0])
sage: dw.a_statistic() # 2 half-squares, 0 complete squares
0

sage: dw = DyckWord([1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,0])
sage: dw.a_statistic()
19

sage: DyckWord([1,1,1,1,0,0,0,0]).a_statistic()
6
sage: DyckWord([1,1,1,0,1,0,0,0]).a_statistic()
5
sage: DyckWord([1,1,1,0,0,1,0,0]).a_statistic()
4
sage: DyckWord([1,1,1,0,0,0,1,0]).a_statistic()
3
sage: DyckWord([1,0,1,1,0,1,0,0]).a_statistic()
2
sage: DyckWord([1,1,0,1,1,0,0,0]).a_statistic()
4
sage: DyckWord([1,1,0,0,1,1,0,0]).a_statistic()
2
sage: DyckWord([1,0,1,1,1,0,0,0]).a_statistic()
3
sage: DyckWord([1,0,1,1,0,0,1,0]).a_statistic()
1
sage: DyckWord([1,0,1,0,1,1,0,0]).a_statistic()
1
sage: DyckWord([1,1,0,0,1,0,1,0]).a_statistic()
1
sage: DyckWord([1,1,0,1,0,0,1,0]).a_statistic()
2
sage: DyckWord([1,1,0,1,0,1,0,0]).a_statistic()
3
sage: DyckWord([1,0,1,0,1,0,1,0]).a_statistic()
0
```

associated_parenthesis(pos)

EXAMPLES:

```
sage: DyckWord([1, 0]).associated_parenthesis(0)
1
```

```
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(0)
1
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(1)
0
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(2)
3
sage: DyckWord([1, 0, 1, 0]).associated_parenthesis(3)
2
sage: DyckWord([1, 1, 0]).associated_parenthesis(1)
2
sage: DyckWord([1, 1]).associated_parenthesis(0)
```

b_statistic()

Returns the b-statistic for the Dyck word.

One can view a balanced Dyck word as a lattice path from $(0, 0)$ to (n, n) in the first quadrant by letting '1's represent steps in the direction $(0, 1)$ and '0's represent steps in the direction $(1, 0)$. The resulting path will remain weakly above the diagonal $y = x$.

We describe the b-statistic of such a path in terms of what is known as the "bounce path".

We can think of our bounce path as describing the trail of a billiard ball shot West from (n, n), which "bounces" down whenever it encounters a vertical step and "bounces" left when it encounters the line y = x. The bouncing ball will strike the diagonal at places $(0, 0), (j_1, j_1), (j_2, j_2), ... , (j_r-1, j_r-1), (j_r, j_r) = (n, n)$. We define the b-statistic to be the sum $\sum_{i=1}^{r-1} j_i$.

(0, 0), (j_1, j_1), (j_2, j_2), ... , (j_r-1, j_r-1), (j_r, j_r) = (n, n).

We define the b-statistic to be the sum $\sum_{i=1}^{r-1} n - j_i$.

EXAMPLES:

```
sage: dw = DyckWord([1,1,1,0,1,1,1,0,0,0,1,1,0,0,1,0,0,0])
sage: dw.b_statistic()
7

sage: DyckWord([1,1,1,1,0,0,0,0]).b_statistic()
0
sage: DyckWord([1,1,1,0,1,0,0,0]).b_statistic()
1
sage: DyckWord([1,1,1,0,0,1,0,0]).b_statistic()
2
sage: DyckWord([1,1,1,0,0,0,1,0]).b_statistic()
3
sage: DyckWord([1,0,1,1,0,1,0,0]).b_statistic()
3
sage: DyckWord([1,1,0,1,1,0,0,0]).b_statistic()
1
sage: DyckWord([1,1,0,0,1,1,0,0]).b_statistic()
2
sage: DyckWord([1,0,1,1,1,0,0,0]).b_statistic()
1
sage: DyckWord([1,0,1,1,0,0,1,0]).b_statistic()
4
sage: DyckWord([1,0,1,0,1,1,0,0]).b_statistic()
3
sage: DyckWord([1,1,0,0,1,0,1,0]).b_statistic()
5
sage: DyckWord([1,1,0,1,0,0,1,0]).b_statistic()
4
sage: DyckWord([1,1,0,1,0,1,0,0]).b_statistic()
2
```

```
sage: DyckWord([1,0,1,0,1,0,1,0]).b_statistic()
6
```

REFERENCES:

- [1] The q,t-Catalan Numbers and the Space of Diagonal Harmonics: With an Appendix on the Combinatorics of Macdonald Polynomials - James Haglund, University of Pennsylvania, Philadelphia - AMS, 2008, 167 pp.

height()

Returns the height of the Dyck word.

We view the Dyck word as a Dyck path from $(0,0)$ to $(n,0)$ in the first quadrant by letting '1's represent steps in the direction $(1,1)$ and '0's represent steps in the direction $(1,-1)$.

The height is the maximum y-coordinate reached.

EXAMPLES:

```
sage: DyckWord([]).height()
0
sage: DyckWord([1,0]).height()
1
sage: DyckWord([1, 1, 0, 0]).height()
2
sage: DyckWord([1, 1, 0, 1, 0]).height()
2
sage: DyckWord([1, 1, 0, 0, 1, 0]).height()
2
sage: DyckWord([1, 0, 1, 0]).height()
1
sage: DyckWord([1, 1, 0, 0, 1, 1, 1, 0, 0, 0]).height()
3
```

peaks()

Returns a list of the positions of the peaks of a Dyck word. A peak is 1 followed by a 0. Note that this does not agree with the definition given by Haglund in: The q,t-Catalan Numbers and the Space of Diagonal Harmonics: With an Appendix on the Combinatorics of Macdonald Polynomials - James Haglund, University of Pennsylvania, Philadelphia - AMS, 2008, 167 pp.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).peaks()
[0, 2]
sage: DyckWord([1, 1, 0, 0]).peaks()
[1]
sage: DyckWord([1,1,0,1,0,1,0,0]).peaks() # Haglund's def gives 2
[1, 3, 5]
```

size()

Returns the size of the Dyck word, which is the number of opening parentheses in the Dyck word.

EXAMPLES:

```
sage: DyckWord([1, 0, 1, 0]).size()
2
sage: DyckWord([1, 0, 1, 1, 0]).size()
3
```

to_noncrossing_partition()

Bijection of Biane from Dyck words to non crossing partitions Thanks to Mathieu Dutour for describing the bijection.

EXAMPLES:

```
sage: DyckWord([1, 0]).to_noncrossing_partition()
[[1]]
sage: DyckWord([1, 1, 0, 0]).to_noncrossing_partition()
[[1, 2]]
sage: DyckWord([1, 1, 1, 0, 0, 0]).to_noncrossing_partition()
[[1, 2, 3]]
sage: DyckWord([1, 0, 1, 0, 1, 0]).to_noncrossing_partition()
[[1], [2], [3]]
sage: DyckWord([1, 1, 0, 1, 0, 0]).to_noncrossing_partition()
[[2], [1, 3]]
```

to_ordered_tree()

TESTS:

```
sage: DyckWord([1, 1, 0, 0]).to_ordered_tree()
...
NotImplementedError: TODO
```

to_tableau()

EXAMPLES:

```
sage: DyckWord([]).to_tableau()
[]
sage: DyckWord([1, 0]).to_tableau()
[[2], [1]]
sage: DyckWord([1, 1, 0, 0]).to_tableau()
[[3, 4], [1, 2]]
sage: DyckWord([1, 0, 1, 0]).to_tableau()
[[2, 4], [1, 3]]
sage: DyckWord([1]).to_tableau()
[[1]]
sage: DyckWord([1, 0, 1]).to_tableau()
[[2], [1, 3]]
```

to_triangulation()

TESTS:

```
sage: DyckWord([1, 1, 0, 0]).to_triangulation()
...
NotImplementedError: TODO
```

DyckWords(*k1=None, k2=None*)

Returns the combinatorial class of Dyck words. A Dyck word is a sequence $(w_1, ..., w_n)$ consisting of '1's and '0's, with the property that for any i with $1 \leq i \leq n$, the sequence $(w_1, ..., w_i)$ contains at least as many 1 s as 0 .

A Dyck word is balanced if the total number of '1's is equal to the total number of '0's. The number of balanced Dyck words of length $2k$ is given by the Catalan number C_k.

EXAMPLES: If neither k1 nor k2 are specified, then DyckWords returns the combinatorial class of all balanced Dyck words.

```
sage: DW = DyckWords(); DW
Dyck words
sage: [] in DW
True
sage: [1, 0, 1, 0] in DW
True
sage: [1, 1, 0] in DW
False
```

If just k1 is specified, then it returns the combinatorial class of balanced Dyck words with k1 opening parentheses and k1 closing parentheses.

```
sage: DW2 = DyckWords(2); DW2
Dyck words with 2 opening parentheses and 2 closing parentheses
sage: DW2.first()
[1, 0, 1, 0]
sage: DW2.last()
[1, 1, 0, 0]
sage: DW2.cardinality()
2
sage: DyckWords(100).cardinality() == catalan_number(100)
True
```

If k2 is specified in addition to k1, then it returns the combinatorial class of Dyck words with k1 opening parentheses and k2 closing parentheses.

```
sage: DW32 = DyckWords(3,2); DW32
Dyck words with 3 opening parentheses and 2 closing parentheses
sage: DW32.list()
[[1, 0, 1, 0, 1],
 [1, 0, 1, 1, 0],
 [1, 1, 0, 0, 1],
 [1, 1, 0, 1, 0],
 [1, 1, 1, 0, 0]]
```

class DyckWords_all()

class DyckWords_size(*k1, k2=None*)

cardinality()

Returns the number of Dyck words of size n, i.e. the n-th Catalan number.

EXAMPLES:

```
sage: DyckWords(4).cardinality()
14
sage: ns = range(9)
sage: dws = [DyckWords(n) for n in ns]
sage: all([ dw.cardinality() == len(dw.list()) for dw in dws])
True
```

list()

Returns a list of all the Dyck words with k1 opening and k2 closing parentheses.

EXAMPLES:

```
sage: DyckWords(0).list()
[[]]
sage: DyckWords(1).list()
[[1, 0]]
sage: DyckWords(2).list()
[[1, 0, 1, 0], [1, 1, 0, 0]]
```

from_noncrossing_partition(*ncp*)

TESTS:

```
sage: DyckWord(noncrossing_partition=[[1,2]]) # indirect doctest
[1, 1, 0, 0]
sage: DyckWord(noncrossing_partition=[[1],[2]])
[1, 0, 1, 0]
```

```
sage: dws = DyckWords(5).list()
sage: ncps = map( lambda x: x.to_noncrossing_partition(), dws)
sage: dws2 = map( lambda x: DyckWord(noncrossing_partition=x), ncps)
sage: dws == dws2
True
```

from_ordered_tree(*tree*)

TESTS:

```
sage: sage.combinat.dyck_word.from_ordered_tree(1)
...
NotImplementedError: TODO
```

is_a(*obj, k1=None, k2=None*)

If k1 is specified, then the object must have exactly k1 open symbols. If k2 is also specified, then obj must have exactly k2 close symbols.

EXAMPLES:

```
sage: from sage.combinat.dyck_word import is_a
sage: is_a([1,1,0,0])
True
sage: is_a([1,0,1,0])
True
sage: is_a([1,1,0,0],2)
True
sage: is_a([1,1,0,0],3)
False
```

is_a_prefix(*obj, k1=None, k2=None*)

If k1 is specified, then the object must have exactly k1 open symbols. If k2 is also specified, then obj must have exactly k2 close symbols.

EXAMPLES:

```
sage: from sage.combinat.dyck_word import is_a_prefix
sage: is_a_prefix([1,1,0])
True
sage: is_a_prefix([0,1,0])
False
sage: is_a_prefix([1,1,0],2,1)
True
sage: is_a_prefix([1,1,0],1,1)
False
```

replace_parens(*x*)

EXAMPLES:

```
sage: from sage.combinat.dyck_word import replace_parens
sage: replace_parens('(')
1
sage: replace_parens(')')
0
sage: replace_parens(1)
...
ValueError
```

replace_symbols(*x*)
EXAMPLES:

```
sage: from sage.combinat.dyck_word import replace_symbols
sage: replace_symbols(1)
'('
sage: replace_symbols(0)
')'
sage: replace_symbols(3)
...
ValueError
```

17.12 Finite combinatorial classes

class **FiniteCombinatorialClass**(*l*)

INPUT: • l a list or iterable

Returns l, wrapped as a combinatorial class

EXAMPLES:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.list()
[1, 2, 3]
sage: F.cardinality()
3
sage: F.random_element()
1
sage: F.first()
1
sage: F.last()
3
```

cardinality()
EXAMPLES:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.cardinality()
3
```

keys()
EXAMPLES: sage: F = FiniteCombinatorialClass([1,2,3]) sage: F.keys() [0, 1, 2]

list()
TESTS:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.list()
[1, 2, 3]
```

object_class(*x*)
EXAMPLES:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.object_class(1)
1
```

class **FiniteCombinatorialClass_l**(*l*)

INPUT: • l a list or iterable

Returns l, wrapped as a combinatorial class

EXAMPLES:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.list()
[1, 2, 3]
sage: F.cardinality()
3
sage: F.random_element()
1
sage: F.first()
1
sage: F.last()
3
```

cardinality()

EXAMPLES:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.cardinality()
3
```

keys()

EXAMPLES: sage: F = FiniteCombinatorialClass([1,2,3]) sage: F.keys() [0, 1, 2]

list()

TESTS:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.list()
[1, 2, 3]
```

object_class(*x*)

EXAMPLES:

```
sage: F = FiniteCombinatorialClass([1,2,3])
sage: F.object_class(1)
1
```

17.13 Tools for generating lists of integers in lexicographic order.

IMPORTANT NOTE (2009/02): The internal functions in this file will be deprecated soon. Please only use them through IntegerListsLex.

class IntegerListsLex(*n, length=None, min_length=0, max_length=+Infinity, floor=None, ceiling=None, min_part=0, max_part=+Infinity, min_slope=-Infinity, max_slope=+Infinity, name=None, element_constructor=None*)

A combinatorial class C for integer lists satisfying certain sum, length, upper/lower bound and regularity constraints. The purpose of this tool is mostly to provide a Constant Amortized Time iterator through those lists, in lexicographic order.

INPUT:

- `n` - a non negative integer
- `min_length` - a non negative integer
- `max_length` - an integer or ∞

- `length` - an integer; overrides min_length and max_length if specified
- `floor` - a function f (or list); defaults to `lambda i: 0`
- `ceiling` - a function f (or list); defaults to `lambda i: infinity`
- `min_slope` - an integer or $-\infty$; defaults to $-\infty$
- `max_slope` - an integer or $+\infty$; defaults to $+\infty$

An *integer list* is a list l of nonnegative integers, its *parts*. The *length* of l is the number of its parts; the *sum* of l is the sum of its parts.

Note: Two valid integer lists are considered equivalent if they only differ by trailing zeroes. In this case, only the list with the least number of trailing zeroes will be produced.

The constraints on the lists are as follow:

- Sum: $sum(l) == n$
- Length: $min\_length \leq len(l) \leq max\_length$
- Lower and upper bounds: $floor(i) \leq l[i] \leq ceiling(i)$, for $i = 0, \ldots, len(l)$
- Regularity condition: $minSlope \leq l[i+1] - l[i] \leq maxSlope$, for $i = 0, \ldots, len(l) - 1$

This is a generic low level tool. The interface has been designed with efficiency in mind. It is subject to incompatible changes in the future. More user friendly interfaces are provided by high level tools like Partitions or Compositions.

EXAMPLES:

We create the combinatorial class of lists of length 3 and sum 2:

```
sage: C = IntegerListsLex(2, length=3)
sage: C
Integer lists of sum 2 satisfying certain constraints
sage: C.cardinality()
6
sage: [p for p in C]
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]

sage: [2, 0, 0] in C
True
sage: [2, 0, 1] in C
False
sage: "a" in C
False
sage: ["a"] in C
False

sage: C.first()
[2, 0, 0]
```

One can specify lower and upper bound on each part:

```
sage: list(IntegerListsLex(5, length = 3, floor = [1,2,0], ceiling = [3,2,3]))
[[3, 2, 0], [2, 2, 1], [1, 2, 2]]
```

Using the slope condition, one can generate integer partitions (but see `sage.combinat.partition.Partitions`):

```
sage: list(IntegerListsLex(4, max_slope=0))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

This is the list of all partitions of 7 with parts at least 2:

```
sage: list(IntegerListsLex(7, max_slope = 0, min_part = 2))
[[7], [5, 2], [4, 3], [3, 2, 2]]
```

This is the list of all partitions of 5 and length at most 3 which are bounded below by [2,1,1]:

```
sage: list(IntegerListsLex(5, max_slope = 0, max_length = 3, floor = [2,1,1]))
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1]]
```

Note that [5] is considered valid, because the lower bound constraint only apply to existing positions in the list. To obtain instead the partitions containing [2,1,1], one need to use min_length:

```
sage: list(IntegerListsLex(5, max_slope = 0, min_length = 3, max_length = 3, floor = [2,1,1]))
[[3, 1, 1], [2, 2, 1]]
```

This is the list of all partitions of 5 which are contained in [3,2,2]:

```
sage: list(IntegerListsLex(5, max_slope = 0, max_length = 3, ceiling = [3,2,2]))
[[3, 2], [3, 1, 1], [2, 2, 1]]
```

This is the list of all compositions of 4 (but see Compositions):

```
sage: list(IntegerListsLex(4, min_part = 1))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

This is the list of all integer vectors of sum 4 and length 3:

```
sage: list(IntegerListsLex(4, length = 3))
[[4, 0, 0], [3, 1, 0], [3, 0, 1], [2, 2, 0], [2, 1, 1], [2, 0, 2], [1, 3, 0], [1, 2, 1], [1, 1,
```

There are all the lists of sum 4 and length 4 such that l[i] <= i:

```
sage: list(IntegerListsLex(4, length=4, ceiling=lambda i: i))
[[0, 1, 2, 1], [0, 1, 1, 2], [0, 1, 0, 3], [0, 0, 2, 2], [0, 0, 1, 3]]
```

This is the list of all monomials of degree 4 which divide the monomial $x^3y^1z^2$ (a monomial being identified with its exponent vector):

```
sage: R.<x,y,z> = QQ[]
sage: m = [3,1,2]
sage: def term(exponents):
...       return x^exponents[0] * y^exponents[1] * z^exponents[2]
...
sage: list( IntegerListsLex(4, length = len(m), ceiling = m, element_constructor = term) )
[x^3*y, x^3*z, x^2*y*z, x^2*z^2, x*y*z^2]
```

Note the use of the element_constructor feature.

In general, the complexity of the iteration algorithm is constant time amortized for each integer list produced. There is one degenerate case though where the algorithm may run forever without producing anything. If max_length is $+\infty$ and max_slope > 0, testing whether there exists a valid integer list of sum n may be only semi-decidable. In the following example, the algorithm will enter an infinite loop, because it needs to decide whether $ceiling(i)$ is nonzero for some i:

```
sage: list( IntegerListsLex(1, ceiling = lambda i: 0) ) # todo: not implemented
```

Note: Caveat: counting is done by brute force generation. In some special cases, it would be possible to do better by counting techniques for integral point in polytopes.

Note: Caveat: with the current implementation, the constraints should satisfy the following conditions:

- The upper and lower bounds themselves should satisfy the slope constraints.
- The maximal and minimal slopes values should not be equal.
- The maximal and minimal part values should not be equal.

Those conditions are not checked by the algorithm, and the result may be completely incorrect if they are not satisfied:

In the following example, the slope condition is not satisfied by the upper bound on the parts, and [3,3] is erroneously included in the result:

```
sage: list(IntegerListsLex(6, max_part=3,max_slope=-1))
[[3, 3], [3, 2, 1]]
```

With some work, this could be fixed withoug affecting the overall complexity and efficiency. Also, the generation algorithm could be extended to deal with non-constant slope constraints and with negative parts, as well as to accept a range parameter instead of a single integer for the sum n of the lists (the later was readilly implemented in MuPAD-Combinat). Encouragements, suggestions, and help are welcome.

TODO: integrate all remaining tests from http://mupad-combinat.svn.sourceforge.net/viewvc/mupad-combinat/trunk/MuPAD-Combinat/lib/COMBINAT/TEST/MachineIntegerListsLex.tst

TESTS:

```
sage: g = lambda x: lambda i: x
sage: list(IntegerListsLex(0, floor = g(1), min_slope = 0))
[[]]
sage: list(IntegerListsLex(0, floor = g(1), min_slope = 0, max_slope = 0))
[[]]
sage: list(IntegerListsLex(0, max_length=0, floor = g(1), min_slope = 0, max_slope = 0))
[[]]
sage: list(IntegerListsLex(0, max_length=0, floor = g(0), min_slope = 0, max_slope = 0))
[[]]
sage: list(IntegerListsLex(0, min_part = 1, min_slope = 0))
[[]]
sage: list(IntegerListsLex(1, min_part = 1, min_slope = 0))
[[1]]
sage: list(IntegerListsLex(0, min_length = 1, min_part = 1, min_slope = 0))
[]
sage: list(IntegerListsLex(0, min_length = 1, min_slope = 0))
[[0]]
sage: list(IntegerListsLex(3, max_length=2, ))
[[3], [2, 1], [1, 2], [0, 3]]
sage: partitions = {"min_part": 1, "max_slope": 0}
sage: partitions_min_2 = {"floor": g(2), "max_slope": 0}
sage: compositions = {"min_part": 1}
sage: integer_vectors = lambda l: {"length": l}
sage: lower_monomials = lambda c: {"length": c, "floor": lambda i: c[i]}
sage: upper_monomials = lambda c: {"length": c, "ceiling": lambda i: c[i]}
sage: constraints = { "min_part":1, "min_slope": -1, "max_slope": 0}
sage: list(IntegerListsLex(6, **partitions))
[[6],
 [5, 1],
 [4, 2],
 [4, 1, 1],
 [3, 3],
```

```
 [3, 2, 1],
 [3, 1, 1, 1],
 [2, 2, 2],
 [2, 2, 1, 1],
 [2, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1]]
sage: list(IntegerListsLex(6, **constraints))
[[6],
 [3, 3],
 [3, 2, 1],
 [2, 2, 2],
 [2, 2, 1, 1],
 [2, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1]]
sage: list(IntegerListsLex(1, **partitions_min_2))
[]
sage: list(IntegerListsLex(2, **partitions_min_2))
[[2]]
sage: list(IntegerListsLex(3, **partitions_min_2))
[[3]]
sage: list(IntegerListsLex(4, **partitions_min_2))
[[4], [2, 2]]
sage: list(IntegerListsLex(5, **partitions_min_2))
[[5], [3, 2]]
sage: list(IntegerListsLex(6, **partitions_min_2))
[[6], [4, 2], [3, 3], [2, 2, 2]]
sage: list(IntegerListsLex(7, **partitions_min_2))
[[7], [5, 2], [4, 3], [3, 2, 2]]
sage: list(IntegerListsLex(9, **partitions_min_2))
[[9], [7, 2], [6, 3], [5, 4], [5, 2, 2], [4, 3, 2], [3, 3, 3], [3, 2, 2, 2]]
sage: list(IntegerListsLex(10, **partitions_min_2))
[[10],
 [8, 2],
 [7, 3],
 [6, 4],
 [6, 2, 2],
 [5, 5],
 [5, 3, 2],
 [4, 4, 2],
 [4, 3, 3],
 [4, 2, 2, 2],
 [3, 3, 2, 2],
 [2, 2, 2, 2, 2]]
sage: list(IntegerListsLex(4, **compositions))
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

build_args()

Returns a list of arguments that can be passed into the prexisting first,next,is_a, ... functions in this module. n is currently not included in this list.

EXAMPLES:

```
sage: C = IntegerListsLex(2, length=3)
sage: C.build_args()
[3,
 3,
 <function <lambda> at 0x...>,
 <function <lambda> at 0x...>,
 -Infinity,
```

```
 +Infinity]
```

ceiling(*i*)

Returns the maximum part that can appear in the $i^{th} position in any list produced.

EXAMPLES:

```
sage: C = IntegerListsLex(4, length=2, max_part=3)
sage: C.ceiling(0)
3
sage: C = IntegerListsLex(4, length=2, ceiling=[3,2])
sage: C.ceiling(0)
3
sage: C.ceiling(1)
2
```

count()

Default brute force implementation of count by iteration through all the objects.

Note that this skips the call to _element_constructor, unlike the default implementation from CombinatorialClass

TODO: do the iteration in place to save on copying time

first()

Returns the lexicographically maximal element in this combinatorial class.

EXAMPLES:

```
sage: C = IntegerListsLex(2, length=3)
sage: C.first()
[2, 0, 0]
```

floor(*i*)

Returns the minimum part that can appear in the i^{th} position in any list produced.

EXAMPLES:

```
sage: C = IntegerListsLex(4, length=2, min_part=1)
sage: C.floor(0)
1
sage: C = IntegerListsLex(4, length=2, floor=[1,2])
sage: C.floor(0)
1
sage: C.floor(1)
2
```

comp2ceil(*c, min_slope, max_slope*)

Given a composition, returns the lowest regular function N->N below it.

EXAMPLES:

```
sage: from sage.combinat.integer_list import comp2ceil
sage: f = comp2ceil([2, 1, 1],-1,0)
sage: [f(i) for i in range(10)]
[2, 1, 1, 1, 2, 3, 4, 5, 6, 7]
```

comp2floor(*f, min_slope, max_slope*)

Given a composition, returns the lowest regular function N->N above it.

EXAMPLES:

```
sage: from sage.combinat.integer_list import comp2floor
sage: f = comp2floor([2, 1, 1],-1,0)
```

```
sage: [f(i) for i in range(10)]
[2, 1, 1, 1, 2, 3, 4, 5, 6, 7]
```

first(*n, min_length, max_length, floor, ceiling, min_slope, max_slope*)

Returns the lexicographically smallest valid composition of n satisfying the conditions.

> **Warning:** INTERNAL FUNCTION! DO NOT USE DIRECTLY!

Preconditions:

- minslope < maxslope
- floor and ceiling need to satisfy the slope constraints, e.g. be obtained fromcomp2floor or comp2ceil
- floor must be below ceiling to ensure the existence a valid composition

TESTS:

```
sage: import sage.combinat.integer_list as integer_list
sage: f = lambda l: lambda i: l[i-1]
sage: f([0,1,2,3,4,5])(1)
0
sage: integer_list.first(12, 4, 4, f([0,0,0,0]), f([4,4,4,4]), -1, 1)
[4, 3, 3, 2]
sage: integer_list.first(36, 9, 9, f([3,3,3,2,1,1,0,0,0]), f([7,6,5,5,5,5,5,4,4]), -1, 1)
[7, 6, 5, 5, 4, 3, 3, 2, 1]
sage: integer_list.first(25, 9, 9, f([3,3,3,2,1,1,0,0,0]), f([7,6,5,5,5,5,5,4,4]), -2, 1)
[7, 6, 5, 4, 2, 1, 0, 0, 0]
sage: integer_list.first(36, 9, 9, f([3,3,3,2,1,4,2,0,0]), f([7,6,5,5,5,5,5,4,4]), -2, 1)
[7, 6, 5, 5, 5, 4, 3, 1, 0]
```

is_a(*comp, min_length, max_length, floor, ceiling, min_slope, max_slope*)

Returns True if comp meets the constraints imposed by the arguments.

> **Warning:** INTERNAL FUNCTION! DO NOT USE DIRECTLY!

EXAMPLES:

```
sage: from sage.combinat.integer_list import is_a
sage: IV = IntegerVectors(2,3,min_slope=0)
sage: params = IV._parameters()
sage: all([is_a(iv, *params) for iv in IV])
True
```

iterator(*n, min_length, max_length, floor, ceiling, min_slope, max_slope*)

> **Warning:** INTERNAL FUNCTION! DO NOT USE DIRECTLY!

EXAMPLES:

```
sage: from sage.combinat.integer_list import iterator
sage: IV = IntegerVectors(2,3,min_slope=0)
sage: params = IV._parameters()
sage: list(iterator(2,*params))
[[0, 1, 1], [0, 0, 2]]
```

list(*n, min_length, max_length, floor, ceiling, min_slope, max_slope*)

Warning: THIS FUNCTION IS DEPRECATED!

Please use IntegersListsLex(...) instead

EXAMPLES:

```
sage: import sage.combinat.integer_list as integer_list
sage: g = lambda x: lambda i: x
sage: integer_list.list(0,0,infinity,g(1),g(infinity),0,infinity)
[[]]
sage: integer_list.list(0,0,infinity,g(1),g(infinity),0,0)
[[]]
sage: integer_list.list(0, 0, 0, g(1), g(infinity), 0, 0)
[[]]
sage: integer_list.list(0, 0, 0, g(0), g(infinity), 0, 0)
[[]]
sage: integer_list.list(0, 0, infinity, g(1), g(infinity), 0, infinity)
[[]]
sage: integer_list.list(1, 0, infinity, g(1), g(infinity), 0, infinity)
[[1]]
sage: integer_list.list(0, 1, infinity, g(1), g(infinity), 0, infinity)
[]
sage: integer_list.list(0, 1, infinity, g(0), g(infinity), 0, infinity)
[[0]]
sage: integer_list.list(3, 0, 2, g(0), g(infinity), -infinity, infinity)
[[3], [2, 1], [1, 2], [0, 3]]
sage: partitions = (0, infinity, g(0), g(infinity), -infinity, 0)
sage: partitions_min_2 = (0, infinity, g(2), g(infinity), -infinity, 0)
sage: compositions = (0, infinity, g(1), g(infinity), -infinity, infinity)
sage: integer_vectors = lambda l: (l, l, g(0), g(infinity), -infinity, infinity)
sage: lower_monomials = lambda c: (len(c), len(c), g(0), lambda i: c[i], -infinity, infinity)
sage: upper_monomials = lambda c: (len(c), len(c), g(0), lambda i: c[i], -infinity, infinity)
sage: constraints = (0, infinity, g(1), g(infinity), -1, 0)
sage: integer_list.list(6, *partitions)
[[6],
 [5, 1],
 [4, 2],
 [4, 1, 1],
 [3, 3],
 [3, 2, 1],
 [3, 1, 1, 1],
 [2, 2, 2],
 [2, 2, 1, 1],
 [2, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1]]
sage: integer_list.list(6, *constraints)
[[6],
 [3, 3],
 [3, 2, 1],
 [2, 2, 2],
 [2, 2, 1, 1],
 [2, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1]]
sage: integer_list.list(1, *partitions_min_2)
[]
sage: integer_list.list(2, *partitions_min_2)
[[2]]
```

```
sage: integer_list.list(3, *partitions_min_2)
[[3]]
sage: integer_list.list(4, *partitions_min_2)
[[4], [2, 2]]
sage: integer_list.list(5, *partitions_min_2)
[[5], [3, 2]]
sage: integer_list.list(6, *partitions_min_2)
[[6], [4, 2], [3, 3], [2, 2, 2]]
sage: integer_list.list(7, *partitions_min_2)
[[7], [5, 2], [4, 3], [3, 2, 2]]
sage: integer_list.list(9, *partitions_min_2)
[[9], [7, 2], [6, 3], [5, 4], [5, 2, 2], [4, 3, 2], [3, 3, 3], [3, 2, 2, 2]]
sage: integer_list.list(10, *partitions_min_2)
[[10],
 [8, 2],
 [7, 3],
 [6, 4],
 [6, 2, 2],
 [5, 5],
 [5, 3, 2],
 [4, 4, 2],
 [4, 3, 3],
 [4, 2, 2, 2],
 [3, 3, 2, 2],
 [2, 2, 2, 2, 2]]
sage: integer_list.list(4, *compositions)
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 3], [1, 2, 1], [1, 1, 2], [1, 1, 1, 1]]
```

lower_regular (*comp, min_slope, max_slope*)

Returns the uppest regular composition below comp

TESTS:

```
sage: import sage.combinat.integer_list as integer_list
sage: integer_list.lower_regular([4,2,6], -1, 1)
[3, 2, 3]
sage: integer_list.lower_regular([4,2,6], -1, infinity)
[3, 2, 6]
sage: integer_list.lower_regular([1,4,2], -1, 1)
[1, 2, 2]
sage: integer_list.lower_regular([4,2,6,3,7], -2, 1)
[4, 2, 3, 3, 4]
sage: integer_list.lower_regular([4,2,infinity,3,7], -2, 1)
[4, 2, 3, 3, 4]
sage: integer_list.lower_regular([1, infinity, 2], -1, 1)
[1, 2, 2]
sage: integer_list.lower_regular([infinity, 4, 2], -1, 1)
[4, 3, 2]
```

next (*comp, min_length, max_length, floor, ceiling, min_slope, max_slope*)

Returns the next integer list after comp that satisfies the constraints.

> **Warning:** INTERNAL FUNCTION! DO NOT USE DIRECTLY!

EXAMPLES:

```
sage: from sage.combinat.integer_list import next
sage: IV = IntegerVectors(2,3,min_slope=0)
```

```
sage: params = IV._parameters()
sage: next([0,1,1], *params)
[0, 0, 2]
```

rightmost_pivot(*comp, min_length, max_length, floor, ceiling, min_slope, max_slope*)

TESTS:

```
sage: import sage.combinat.integer_list as integer_list
sage: f = lambda l: lambda i: l[i-1]
sage: integer_list.rightmost_pivot([7,6,5,5,4,3,3,2,1], 9, 9, f([3,3,3,2,1,1,0,0,0]), f([7,6,5,5
[7, 2]
sage: integer_list.rightmost_pivot([7,6,5,5,4,3,3,2,1], 9, 9,f([3,3,3,2,1,1,0,0,0]), f([7,6,5,5,
[7, 1]
sage: integer_list.rightmost_pivot([7,6,5,5,4,3,3,2,1], 9, 9,f([3,3,3,2,1,1,0,0,0]), f([7,6,5,5,
[8, 1]
sage: integer_list.rightmost_pivot([7,6,5,5,4,3,3,2,1], 9, 9,f([3,3,3,2,1,1,0,0,0]), f([7,6,5,5,
[8, 1]
sage: integer_list.rightmost_pivot([7,6,5,5,5,5,5,4,4], 9, 9,f([3,3,3,2,1,1,0,0,0]), f([7,6,5,5,
sage: integer_list.rightmost_pivot([3,3,3,2,1,1,0,0,0], 9, 9,f([3,3,3,2,1,1,0,0,0]), f([7,6,5,5,
sage: g = lambda x: lambda i: x
sage: integer_list.rightmost_pivot([1],1,1,g(0),g(2),-10, 10)
sage: integer_list.rightmost_pivot([1,2],2,2,g(0),g(2),-10, 10)
sage: integer_list.rightmost_pivot([1,2],2,2,g(1),g(2), -10, 10)
sage: integer_list.rightmost_pivot([1,2],2,3,g(1),g(2), -10, 10)
[2, 1]
sage: integer_list.rightmost_pivot([2,2],2,3,g(2),g(2),-10, 10)
sage: integer_list.rightmost_pivot([2,3],2,3,g(2),g(2),-10,+10)
sage: integer_list.rightmost_pivot([3,2],2,3,g(2),g(2),-10,+10)
sage: integer_list.rightmost_pivot([3,3],2,3,g(2),g(2),-10,+10)
[1, 2]
sage: integer_list.rightmost_pivot([6],1,3,g(0),g(6),-1,0)
[1, 0]
sage: integer_list.rightmost_pivot([6],1,3,g(0),g(6),-2,0)
[1, 0]
sage: integer_list.rightmost_pivot([7,9,8,7],1,5,g(0),g(10),-1,10)
[2, 6]
sage: integer_list.rightmost_pivot([7,9,8,7],1,5,g(5),g(10),-10,10)
[3, 5]
sage: integer_list.rightmost_pivot([7,9,8,7],1,5,g(5),g(10),-1,10)
[2, 6]
sage: integer_list.rightmost_pivot([7,9,8,7],1,5,g(4),g(10),-2,10)
[3, 7]
sage: integer_list.rightmost_pivot([9,8,7],1,4,g(4),g(10),-2,0)
[1, 4]
sage: integer_list.rightmost_pivot([1,3],1,5,lambda i: i,g(10),-10,10)
sage: integer_list.rightmost_pivot([1,4],1,5,lambda i: i,g(10),-10,10)
sage: integer_list.rightmost_pivot([2,4],1,5,lambda i: i,g(10),-10,10)
[1, 1]
```

upper_bound(*min_length, max_length, floor, ceiling, min_slope, max_slope*)

Compute a coarse upper bound on the size of a vector satisfying the constraints.

TESTS:

```
sage: import sage.combinat.integer_list as integer_list
sage: f = lambda x: lambda i: x
sage: integer_list.upper_bound(0,4,f(0), f(1),-infinity,infinity)
4
```

```
sage: integer_list.upper_bound(0, infinity, f(0), f(1), -infinity, infinity)
+Infinity
sage: integer_list.upper_bound(0, infinity, f(0), f(1), -infinity, -1)
1
sage: integer_list.upper_bound(0, infinity, f(0), f(5), -infinity, -1)
15
sage: integer_list.upper_bound(0, infinity, f(0), f(5), -infinity, -2)
9
```

upper_regular (*comp, min_slope, max_slope*)

Returns the uppest regular composition above comp.

TESTS:

```
sage: import sage.combinat.integer_list as integer_list
sage: integer_list.upper_regular([4,2,6],-1,1)
[4, 5, 6]
sage: integer_list.upper_regular([4,2,6],-2, 1)
[4, 5, 6]
sage: integer_list.upper_regular([4,2,6,3,7],-2, 1)
[4, 5, 6, 6, 7]
sage: integer_list.upper_regular([4,2,6,1], -2, 1)
[4, 5, 6, 4]
```

17.14 Integer vectors

IntegerVectors (*n=None, k=None, **kwargs*)

Returns the combinatorial class of integer vectors.

EXAMPLES: If n is not specified, it returns the class of all integer vectors.

```
sage: IntegerVectors()
Integer vectors
sage: [] in IntegerVectors()
True
sage: [1,2,1] in IntegerVectors()
True
sage: [1, 0, 0] in IntegerVectors()
True
```

If n is specified, then it returns the class of all integer vectors which sum to n.

```
sage: IV3 = IntegerVectors(3); IV3
Integer vectors that sum to 3
```

Note that trailing zeros are ignored so that [3, 0] does not show up in the following list (since [3] does)

```
sage: IntegerVectors(3, max_length=2).list()
[[3], [2, 1], [1, 2], [0, 3]]
```

If n and k are both specified, then it returns the class of integer vectors that sum to n and are of length k.

```
sage: IV53 = IntegerVectors(5,3); IV53
Integer vectors of length 3 that sum to 5
sage: IV53.cardinality()
21
```

```
sage: IV53.first()
[5, 0, 0]
sage: IV53.last()
[0, 0, 5]
sage: IV53.random_element()
[4, 0, 1]
```

class **IntegerVectors_all**()

cardinality()
EXAMPLES:

```
sage: IntegerVectors().cardinality()
+Infinity
```

list()
EXAMPLES:

```
sage: IntegerVectors().list()
...
NotImplementedError: infinite list
```

class **IntegerVectors_nconstraints**(*n, constraints*)

cardinality()
EXAMPLES:

```
sage: IntegerVectors(3, max_length=2).cardinality()
4
sage: IntegerVectors(3).cardinality()
+Infinity
```

list()
EXAMPLES:

```
sage: IntegerVectors(3, max_length=2).list()
[[3], [2, 1], [1, 2], [0, 3]]
sage: IntegerVectors(3).list()
...
NotImplementedError: infinite list
```

class **IntegerVectors_nk**(*n, k*)

list()
EXAMPLE:

```
sage: IV = IntegerVectors(2,3)
sage: IV.list()
[[2, 0, 0], [1, 1, 0], [1, 0, 1], [0, 2, 0], [0, 1, 1], [0, 0, 2]]
sage: IntegerVectors(3, 0).list()
[]
sage: IntegerVectors(3, 1).list()
[[3]]
sage: IntegerVectors(0, 1).list()
[[0]]
sage: IntegerVectors(0, 2).list()
[[0, 0]]
```

```
sage: IntegerVectors(2, 2).list()
[[2, 0], [1, 1], [0, 2]]
```

class IntegerVectors_nkconstraints(*n, k, constraints*)

cardinality()
EXAMPLES:

```
sage: IntegerVectors(3,3, min_part=1).cardinality()
1
sage: IntegerVectors(5,3, min_part=1).cardinality()
6
sage: IntegerVectors(13, 4, min_part=2, max_part=4).cardinality()
16
```

first()
EXAMPLES:

```
sage: IntegerVectors(2,3,min_slope=0).first()
[0, 1, 1]
```

next(*x*)
EXAMPLES:

```
sage: IntegerVectors(2,3,min_slope=0).last()
[0, 0, 2]
```

class IntegerVectors_nnondescents(*n, comp*)
The combinatorial class of integer vectors v graded by two parameters:

- n: the sum of the parts of v
- comp: the non descents composition of v

In other words: the length of v equals c[1]+...+c[k], and v is decreasing in the consecutive blocs of length c[1], ..., c[k]

Those are the integer vectors of sum n which are lexicographically maximal (for the natural left->right reading) in their orbit by the young subgroup S_c_1 x x S_c_k. In particular, they form a set of orbit representative of integer vectors w.r.t. this young subgroup.

constant_func(*i*)
Returns the constant function i.

EXAMPLES:

```
sage: f = sage.combinat.integer_vector.constant_func(3)
sage: f(-1)
3
sage: f('asf')
3
```

list2func(*l, default=None*)
Given a list l, return a function that takes in a value i and return l[i-1]. If default is not None, then the function will return the default value for out of range i's.

EXAMPLES:

```
sage: f = sage.combinat.integer_vector.list2func([1,2,3])
sage: f(1)
1
```

```
sage: f(2)
2
sage: f(3)
3
sage: f(4)
...
IndexError: list index out of range

sage: f = sage.combinat.integer_vector.list2func([1,2,3], 0)
sage: f(3)
3
sage: f(4)
0
```

17.15 Weighted Integer Vectors

WeightedIntegerVectors(*n, weight*)

Returns the combinatorial class of integer vectors of n weighted by weight.

EXAMPLES:

```
sage: WeightedIntegerVectors(8, [1,1,2])
Integer vectors of 8 weighted by [1, 1, 2]
sage: WeightedIntegerVectors(8, [1,1,2]).first()
[0, 0, 4]
sage: WeightedIntegerVectors(8, [1,1,2]).last()
[8, 0, 0]
sage: WeightedIntegerVectors(8, [1,1,2]).cardinality()
25
sage: WeightedIntegerVectors(8, [1,1,2]).random_element()
[1, 1, 3]
```

class **WeightedIntegerVectors_nweight**(*n, weight*)

list()

TESTS:

```
sage: WeightedIntegerVectors(7, [2,2]).list()
[]
sage: WeightedIntegerVectors(3, [2,1,1]).list()
[[1, 0, 1], [1, 1, 0], [0, 0, 3], [0, 1, 2], [0, 2, 1], [0, 3, 0]]

sage: ivw = [ WeightedIntegerVectors(k, [1,1,1]) for k in range(11) ]
sage: iv  = [ IntegerVectors(k, 3) for k in range(11) ]
sage: all( [ sorted(iv[k].list()) == sorted(ivw[k].list()) for k in range(11) ] )
True

sage: ivw = [ WeightedIntegerVectors(k, [2,3,7]) for k in range(11) ]
sage: all( [ i.cardinality() == len(i.list()) for i in ivw] )
True
```

17.16 Restricted growth arrays

class RestrictedGrowthArrays(*n*)

cardinality()
EXAMPLES:

```
sage: from sage.combinat.restricted_growth import RestrictedGrowthArrays
sage: R = RestrictedGrowthArrays(6)
sage: R.cardinality()
203
```

17.17 Yamanouchi Words

A right (respectively left) Yamanouchi word on a completely ordered alphabet, for instance [1,2,...,n], is a word math such that any right (respectively left) factor of math contains more entries math than math. For example, the word [2, 3, 2, 2, 1, 3, 1, 2, 1, 1] is a right Yamanouchi one.

The evaluation of a word math encodes the number of occurrences of each letter of math. In the case of Yamanouchi words, the evaluation is a partition. For example, the word [2, 3, 2, 2, 1, 3, 1, 2, 1, 1] has evaluation [4, 4, 2].

Yamanouchi words can be useful in the computation of Littlewood-Richardson coefficients $c^{\nu}_{\lambda,\mu}$. According to the Littlewood-Richardson rule, $c^{\nu}_{\lambda,\mu}$ is the number of skew tableaux of shape ν/λ and evaluation μ, whose row readings are Yamanouchi words.

17.18 Paths in Directed Acyclic Graphs

GraphPaths(*g, source=None, target=None*)
Returns the combinatorial class of paths in the directed acyclic graph g.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
```

If source and target are not given, then the returned class contains all paths (including trivial paths containing only one vertex).

```
sage: p = GraphPaths(G); p
Paths in Multi-digraph on 5 vertices
sage: p.cardinality()
37
sage: p.random_element()
[1, 2, 3, 4, 5]
```

If the source is specified, then the returned class contains all of the paths starting at the vertex source (including the trivial path).

```
sage: p = GraphPaths(G, source=3); p
Paths in Multi-digraph on 5 vertices starting at 3
sage: p.list()
[[3], [3, 4], [3, 4, 5], [3, 4, 5]]
```

If the target is specified, then the returned class contains all of the paths ending at the vertex target (including the trivial path).

```
sage: p = GraphPaths(G, target=3); p
Paths in Multi-digraph on 5 vertices ending at 3
sage: p.cardinality()
5
sage: p.list()
[[3], [1, 3], [2, 3], [1, 2, 3], [1, 2, 3]]
```

If both the target and source are specified, then the returned class contains all of the paths from source to target.

```
sage: p = GraphPaths(G, source=1, target=3); p
Paths in Multi-digraph on 5 vertices starting at 1 and ending at 3
sage: p.cardinality()
3
sage: p.list()
[[1, 2, 3], [1, 2, 3], [1, 3]]
```

Note that G must be a directed acyclic graph.

```
sage: G = DiGraph({1:[2,2,3,5], 2:[3,4], 3:[4], 4:[2,5,7], 5:[6]}, multiedges=True)
sage: GraphPaths(G)
...
TypeError: g must be a directed acyclic graph
```

class GraphPaths_all(*g*)

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G)
sage: p.cardinality()
37
```

list()

Returns a list of the paths of self.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: len(GraphPaths(G).list())
37
```

class GraphPaths_common()

incoming_edges(*v*)

Returns a list of v's incoming edges.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G)
sage: p.incoming_edges(2)
[(1, 2, None), (1, 2, None)]
```

incoming_paths(*v*)

Returns a list of paths that end at v.

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: gp.incoming_paths(2)
[[2], [1, 2], [1, 2]]
```

outgoing_edges(*v*)
Returns a list of v's outgoing edges.
EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G)
sage: p.outgoing_edges(2)
[(2, 3, None), (2, 4, None)]
```

outgoing_paths(*v*)
Returns a list of the paths that start at v.
EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: gp.outgoing_paths(3)
[[3], [3, 4], [3, 4, 5], [3, 4, 5]]
sage: gp.outgoing_paths(2)
[[2],
 [2, 3],
 [2, 3, 4],
 [2, 3, 4, 5],
 [2, 3, 4, 5],
 [2, 4],
 [2, 4, 5],
 [2, 4, 5]]
```

paths()
Returns a list of all the paths of self.
EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: len(gp.paths())
37
```

paths_from_source_to_target(*source, target*)
Returns a list of paths from source to target.
EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: gp = GraphPaths(G)
sage: gp.paths_from_source_to_target(2,4)
[[2, 3, 4], [2, 4]]
```

class **GraphPaths_s**(*g, source*)

list()
EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G, 4)
sage: p.list()
[[4], [4, 5], [4, 5]]
```

class **GraphPaths_st**(*g, source, target*)
EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: GraphPaths(G,1,2).cardinality()
2
sage: GraphPaths(G,1,3).cardinality()
3
sage: GraphPaths(G,1,4).cardinality()
5
sage: GraphPaths(G,1,5).cardinality()
10
sage: GraphPaths(G,2,3).cardinality()
1
sage: GraphPaths(G,2,4).cardinality()
2
sage: GraphPaths(G,2,5).cardinality()
4
sage: GraphPaths(G,3,4).cardinality()
1
sage: GraphPaths(G,3,5).cardinality()
2
sage: GraphPaths(G,4,5).cardinality()
2
```

list()

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G,1,2)
sage: p.list()
[[1, 2], [1, 2]]
```

class GraphPaths_t(*g*, *target*)

list()

EXAMPLES:

```
sage: G = DiGraph({1:[2,2,3], 2:[3,4], 3:[4], 4:[5,5]}, multiedges=True)
sage: p = GraphPaths(G, target=4)
sage: p.list()
[[4],
 [2, 4],
 [1, 2, 4],
 [1, 2, 4],
 [3, 4],
 [1, 3, 4],
 [2, 3, 4],
 [1, 2, 3, 4],
 [1, 2, 3, 4]]
```

17.19 Latin Squares

A *latin square* of order n is an $n \times n$ array such that each symbol $s \in \{0, 1, \ldots, n-1\}$ appears precisely once in each row, and precisely once in each column. A *partial latin square* of order n is an $n \times n$ array such that each symbol $s \in \{0, 1, \ldots, n-1\}$ appears at most once in each row, and at most once in each column. Empty cells are denoted by -1. A latin square L is a *completion* of a partial latin square P if $P \subseteq L$. If P completes to just L then P *has unique completion*.

A *latin bitrade* (T_1, T_2) is a pair of partial latin squares such that:

1. $\{(i, j) \mid (i, j, k) \in T_1 \text{ for some symbol } k\} = \{(i, j) \mid (i, j, k') \in T_2 \text{ for some symbol } k'\}$;
2. for each $(i, j, k) \in T_1$ and $(i, j, k') \in T_2$, $k \neq k'$;
3. the symbols appearing in row i of T_1 are the same as those of row i of T_2; the symbols appearing in column j of T_1 are the same as those of column j of T_2.

Intuitively speaking, a bitrade gives the difference between two latin squares, so if (T_1, T_2) is a bitrade for the pair of latin squares (L_1, L_2), then $L1 = (L2 \setminus T_1) \cup T_2$ and $L2 = (L1 \setminus T_2) \cup T_1$.

This file contains

1. LatinSquare class definition;
2. some named latin squares (back circulant, forward circulant, abelian 2-group);
3. functions is_partial_latin_square and is_latin_square to test if a LatinSquare object satisfies the definition of a latin square or partial latin square, respectively;
4. tests for completion and unique completion (these use the C++ implementation of Knuth's dancing links algorithm to solve the problem as a instance of $0 - 1$ matrix exact cover);
5. functions for calculating the τ_i representation of a bitrade and the genus of the associated hypermap embedding;
6. Markov chain of Jacobson and Matthews (1996) for generating latin squares uniformly at random (provides a generator interface);
7. a few examples of τ_i representations of bitrades constructed from the action of a group on itself by right multiplication, functions for converting to a pair of LatinSquare objects.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(5)
sage: B
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
sage: B.is_latin_square()
True
sage: B[0, 1] = 0
sage: B.is_latin_square()
False

sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0  2 -1  1]
[-1  0  1  3]
[ 3 -1  0  2]
[ 1  3  2 -1]
sage: T2
[ 1  0 -1  2]
[-1  3  0  1]
[ 0 -1  2  3]
```

```
[ 3  2  1 -1]
sage: T1.nr_filled_cells()
12
sage: genus(T1, T2)
1
```

To do:

1. Latin squares with symbols from a ring instead of the integers $\{0, 1, \ldots, n-1\}$.
2. Isotopism testing of latin squares and bitrades via graph isomorphism (nauty?).
3. Combinatorial constructions for bitrades.

AUTHORS:

- Carlo Hamalainen (2008-03-23): initial version

TESTS:

```
sage: L = elementary_abelian_2group(3)
sage: L == loads(dumps(L))
True
```

class LatinSquare(**args*)

actual_row_col_sym_sizes()

Bitrades sometimes end up in partial latin squares with unused rows, columns, or symbols. This function works out the actual number of used rows, columns, and symbols.

> **Warning:** We assume that the unused rows/columns occur in the lower right of self, and that the used symbols are in the range {0, 1, ..., m} (no holes in that list).

EXAMPLE:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(3)
sage: B[0,2] = B[1,2] = B[2,2] = -1
sage: B[0,0] = B[2,1] = -1
sage: B
[-1  1 -1]
[ 1  2 -1]
[ 2 -1 -1]
sage: B.actual_row_col_sym_sizes()
(3, 2, 2)
```

apply_isotopism(*row_perm, col_perm, sym_perm*)

An isotopism is a permutation of the rows, columns, and symbols of a partial latin square self. Use isotopism() to convert a tuple (indexed from 0) to a Permutation object.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(5)
sage: B
[0 1 2 3 4]
[1 2 3 4 0]
```

```
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
sage: alpha = isotopism((0,1,2,3,4))
sage: beta  = isotopism((1,0,2,3,4))
sage: gamma = isotopism((2,1,0,3,4))
sage: B.apply_isotopism(alpha, beta, gamma)
[3 4 2 0 1]
[0 2 3 1 4]
[1 3 0 4 2]
[4 0 1 2 3]
[2 1 4 3 0]
```

clear_cells()

Mark every cell in self as being empty.

EXAMPLES:

```
sage: A = LatinSquare(matrix(ZZ, [[0, 1], [2, 3]]))
sage: A.clear_cells()
sage: A
[-1 -1]
[-1 -1]
```

column(*x*)

Returns column x of the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(3).column(0)
(0, 1, 2)
```

contained_in(*Q*)

Returns True if self is a subset of Q?

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: P = elementary_abelian_2group(2)
sage: P[0, 0] = -1
sage: P.contained_in(elementary_abelian_2group(2))
True
sage: back_circulant(4).contained_in(elementary_abelian_2group(2))
False
```

copy()

To copy a latin square we must copy the underlying matrix.

EXAMPLES:

```
sage: A = LatinSquare(matrix(ZZ, [[0, 1], [2, 3]]))
sage: B = A.copy()
sage: A
[0 1]
[2 3]
```

disjoint_mate_dlxcpp_rows_and_map(*allow_subtrade*)

Internal function for find_disjoint_mates.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(4)
```

```
sage: B.disjoint_mate_dlxcpp_rows_and_map(allow_subtrade = True)
([[0, 16, 32], [1, 17, 32], [2, 18, 32], [3, 19, 32], [4, 16, 33], [5, 17, 33], [6, 18, 33],
```

dlxcpp_has_unique_completion()

Check if the partial latin square self of order n can be embedded in precisely one latin square of order n.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(2).dlxcpp_has_unique_completion()
True
sage: P = LatinSquare(2)
sage: P.dlxcpp_has_unique_completion()
False
sage: P[0, 0] = 0
sage: P.dlxcpp_has_unique_completion()
True
```

dumps()

Since the latin square class doesn't hold any other private variables we just call dumps on self.square:

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(2) == loads(dumps(back_circulant(2)))
True
```

filled_cells_map()

Number the filled cells of self with integers from {1, 2, 3, ...}

INPUT:

- `self` - Partial latin square self (empty cells have negative values)

OUTPUT: A dictionary cells_map where cells_map[(i,j)] = m means that (i,j) is the m-th filled cell in P, while cells_map[m] = (i,j).

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1.filled_cells_map()
{1: (0, 0), 2: (0, 1), 3: (0, 3), 4: (1, 1), 5: (1, 2), 6: (1, 3), 7: (2, 0), 8: (2, 2), 9:
```

find_disjoint_mates(*nr_to_find=None, allow_subtrade=False*)

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(4)
sage: g = B.find_disjoint_mates(allow_subtrade = True)
sage: B1 = g.next()
sage: B0, B1 = bitrade(B, B1)
sage: assert is_bitrade(B0, B1)
sage: print B0, "\n,\n", B1
[-1  1  2 -1]
[-1  2 -1  0]
[-1 -1 -1 -1]
[-1  0  1  2]
,
[-1  2  1 -1]
[-1  0 -1  2]
[-1 -1 -1 -1]
[-1  1  2  0]
```

gcs()

A greedy critical set of a latin square self is found by successively removing elements in a row-wise (bottom-up) manner, checking for unique completion at each step.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: A = elementary_abelian_2group(3)
sage: G = A.gcs()
sage: A
[0 1 2 3 4 5 6 7]
[1 0 3 2 5 4 7 6]
[2 3 0 1 6 7 4 5]
[3 2 1 0 7 6 5 4]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
sage: G
[ 0  1  2  3  4  5  6 -1]
[ 1  0  3  2  5  4 -1 -1]
[ 2  3  0  1  6 -1  4 -1]
[ 3  2  1  0 -1 -1 -1 -1]
[ 4  5  6 -1  0  1  2 -1]
[ 5  4 -1 -1  1  0 -1 -1]
[ 6 -1  4 -1  2 -1  0 -1]
[-1 -1 -1 -1 -1 -1 -1 -1]
```

is_completable()

Returns True if the partial latin square can be completed to a latin square.

EXAMPLES:

The following partial latin square has no completion because there is nowhere that we can place the symbol 0 in the third row:

```
sage: B = LatinSquare(3)

sage: B[0, 0] = 0
sage: B[1, 1] = 0
sage: B[2, 2] = 1

sage: B
[ 0 -1 -1]
[-1  0 -1]
[-1 -1  1]

sage: B.is_completable()
False

sage: B[2, 2] = 0
sage: B.is_completable()
True
```

is_empty_column(*c*)

Checks if column c of the partial latin square self is empty.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(4)
sage: L.is_empty_column(0)
False
```

```
sage: L[0,0] = L[1,0] = L[2,0] = L[3,0] = -1
sage: L.is_empty_column(0)
True
```

is_empty_row(*r*)

Checks if row r of the partial latin square self is empty.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(4)
sage: L.is_empty_row(0)
False
sage: L[0,0] = L[0,1] = L[0,2] = L[0,3] = -1
sage: L.is_empty_row(0)
True
```

is_latin_square()

self is a latin square if it is an n by n matrix, and each symbol in [0, 1, ..., n-1] appears exactly once in each row, and exactly once in each column.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: elementary_abelian_2group(4).is_latin_square()
True

sage: forward_circulant(7).is_latin_square()
True
```

is_partial_latin_square()

self is a partial latin square if it is an n by n matrix, and each symbol in [0, 1, ..., n-1] appears at most once in each row, and at most once in each column.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: LatinSquare(4).is_partial_latin_square()
True
sage: back_circulant(3).gcs().is_partial_latin_square()
True
sage: back_circulant(6).is_partial_latin_square()
True
```

is_uniquely_completable()

Returns True if the partial latin square self has exactly one completion to a latin square. This is just a wrapper for the current best-known algorithm, Dancing Links by Knuth. See dancing_links.spyx

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(4).gcs().is_uniquely_completable()
True

sage: G = elementary_abelian_2group(3).gcs()
sage: G.is_uniquely_completable()
True

sage: G[0, 0] = -1
sage: G.is_uniquely_completable()
False
```

latex()

Returns LaTeX code for the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: print back_circulant(3).latex()
\begin{array}{|c|c|c|}\hline 0 & 1 & 2\\\hline 1 & 2 & 0\\\hline 2 & 0 & 1\\\hline\end{array
```

list()

Convert the latin square into a list, in a row-wise manner.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(3).list()
[0, 1, 2, 1, 2, 0, 2, 0, 1]
```

ncols()

Number of columns in the latin square.

EXAMPLES:

```
sage: LatinSquare(3).ncols()
3
```

nr_distinct_symbols()

Returns the number of distinct symbols in the partial latin square self.

EXAMPLE:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(5).nr_distinct_symbols()
5
sage: L = LatinSquare(10)
sage: L.nr_distinct_symbols()
0
sage: L[0, 0] = 0
sage: L[0, 1] = 1
sage: L.nr_distinct_symbols()
2
```

nr_filled_cells()

Returns the number of filled cells (i.e. cells with a positive value) in the partial latin square self.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: LatinSquare(matrix([[0, -1], [-1, 0]])).nr_filled_cells()
2
```

nrows()

Number of rows in the latin square.

EXAMPLES:

```
sage: LatinSquare(3).nrows()
3
```

permissable_values(*r*, *c*)

Find all values that do not appear in row r and column c of the latin square self. If self[r, c] is filled then we return the empty list.

INPUT:

- `self` - LatinSquare
- `r` - int; row of the latin square
- `c` - int; column of the latin square

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(5)
sage: L[0, 0] = -1
sage: L.permissable_values(0, 0)
[0]
```

random_empty_cell()

Find an empty cell of self, uniformly at random.

INPUT:

- `self` - LatinSquare

OUTPUT:

- `[r, c]` - cell such that self[r, c] is empty, or returns None if self is a (full) latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: P = back_circulant(2)
sage: P[1,1] = -1
sage: P.random_empty_cell()
[1, 1]
```

row(*x*)

Returns row x of the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(3).row(0)
(0, 1, 2)
```

set_immutable()

A latin square is immutable if the underlying matrix is immutable.

EXAMPLES:

```
sage: L = LatinSquare(matrix(ZZ, [[0, 1], [2, 3]]))
sage: L.set_immutable()
sage: {L : 0}   # this would fail without set_immutable()
{[0 1]
[2 3]: 0}
```

top_left_empty_cell()

Returns the least [r, c] such that self[r, c] is an empty cell. If all cells are filled then we return None.

INPUT:

- `self` - LatinSquare

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(5)
sage: B[3, 4] = -1
sage: B.top_left_empty_cell()
[3, 4]
```

vals_in_col(*c*)

Returns a dictionary with key e if and only if column c of self has the symbol e.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(3)
```

```
sage: B[0, 0] = -1
sage: back_circulant(3).vals_in_col(0)
{0: True, 1: True, 2: True}
```

vals_in_row(*r*)

Returns a dictionary with key e if and only if row r of self has the symbol e.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B = back_circulant(3)
sage: B[0, 0] = -1
sage: back_circulant(3).vals_in_row(0)
{0: True, 1: True, 2: True}
```

LatinSquare_generator(*L_start, check_assertions=False*)

Generator for a sequence of uniformly distributed latin squares, given L_start as the initial latin square. This code implements the Markov chain algorithm of Jacobson and Matthews (1996), see below for the BibTex entry. This generator will never throw the StopIteration exception, so it provides an infinite sequence of latin squares.

EXAMPLES:

Use the back circulant latin square of order 4 as the initial square and print the next two latin squares given by the Markov chain:

```
sage: from sage.combinat.matrices.latin import *
sage: g = LatinSquare_generator(back_circulant(4))
sage: g.next().is_latin_square()
True
```

REFERENCE:

```
@article{MR1410617,
    AUTHOR = {Jacobson, Mark T. and Matthews, Peter},
     TITLE = {Generating uniformly distributed random {L}atin squares},
   JOURNAL = {J. Combin. Des.},
  FJOURNAL = {Journal of Combinatorial Designs},
    VOLUME = {4},
      YEAR = {1996},
    NUMBER = {6},
     PAGES = {405--437},
      ISSN = {1063-8539},
   MRCLASS = {05B15 (60J10)},
  MRNUMBER = {MR1410617 (98b:05021)},
MRREVIEWER = {Lars D{\o}vling Andersen},
}
```

alternating_group_bitrade_generators(*m*)

Construct generators a, b, c for the alternating group on 3m+1 points, such that a*b*c = 1.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: a, b, c, G = alternating_group_bitrade_generators(1)
sage: (a, b, c, G)
((1,2,3), (1,4,2), (2,4,3), Permutation Group with generators [(1,2,3), (1,4,2)])
sage: a*b*c
()
```

```
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0  2 -1  1]
[-1  0  1  3]
[ 3 -1  0  2]
[ 1  3  2 -1]
sage: T2
[ 1  0 -1  2]
[-1  3  0  1]
[ 0 -1  2  3]
[ 3  2  1 -1]
```

back_circulant(*n*)

The back-circulant latin square of order n is the Cayley table for (Z_n, +), the integers under addition modulo n.

INPUT:

- `n` - int; order of the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: back_circulant(5)
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
```

beta1(*rce, T1, T2*)

Find the unique (x, c, e) in T2 such that (r, c, e) is in T1.

INPUT:

- `rce` - tuple (or list) (r, c, e) in T1
- `T1, T2` - latin bitrade

OUTPUT: (x, c, e) in T2.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: beta1([0, 0, 0], T1, T2)
(1, 0, 0)
```

beta2(*rce, T1, T2*)

Find the unique (r, x, e) in T2 such that (r, c, e) is in T1.

INPUT:

- `rce` - tuple (or list) (r, c, e) in T1
- `T1, T2` - latin bitrade

OUTPUT:

- (r, x, e) in T2.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: beta2([0, 0, 0], T1, T2)
(0, 1, 0)
```

beta3 (*rce, T1, T2*)

Find the unique (r, c, x) in T2 such that (r, c, e) is in T1.

INPUT:

- `rce` - tuple (or list) (r, c, e) in T1
- `T1, T2` - latin bitrade

OUTPUT:

- (r, c, x) in T2.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: beta3([0, 0, 0], T1, T2)
(0, 0, 4)
```

bitrade (*T1, T2*)

Form the bitrade (Q1, Q2) from (T1, T2) by setting empty the cells (r, c) such that T1[r, c] == T2[r, c].

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: B1 = back_circulant(5)
sage: alpha = isotopism((0,1,2,3,4))
sage: beta  = isotopism((1,0,2,3,4))
sage: gamma = isotopism((2,1,0,3,4))
sage: B2 = B1.apply_isotopism(alpha, beta, gamma)
sage: T1, T2 = bitrade(B1, B2)
sage: T1
[ 0  1 -1  3  4]
[ 1 -1 -1  4  0]
[ 2 -1  4  0  1]
[ 3  4  0  1  2]
[ 4  0  1  2  3]
```

```
sage: T2
[ 3  4 -1  0  1]
[ 0 -1 -1  1  4]
[ 1 -1  0  4  2]
[ 4  0  1  2  3]
[ 2  1  4  3  0]
```

bitrade_from_group(*a, b, c, G*)

Given group elements a, b, c in G such that abc = 1 and the subgroups a, b, c intersect (pairwise) only in the identity, construct a bitrade (T1, T2) where rows, columns, and symbols correspond to cosets of a, b, and c, respectively.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: a, b, c, G = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0  2 -1  1]
[-1  0  1  3]
[ 3 -1  0  2]
[ 1  3  2 -1]
sage: T2
[ 1  0 -1  2]
[-1  3  0  1]
[ 0 -1  2  3]
[ 3  2  1 -1]
```

cells_map_as_square(*cells_map, n*)

Returns a LatinSquare with cells numbered from 1, 2, ... to given the dictionary cells_map.

Note: The value n should be the maximum of the number of rows and columns of the original partial latin square

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0  2 -1  1]
[-1  0  1  3]
[ 3 -1  0  2]
[ 1  3  2 -1]
```

There are 12 filled cells in T:

```
sage: cells_map_as_square(T1.filled_cells_map(), max(T1.nrows(), T1.ncols()))
[ 1  2 -1  3]
[-1  4  5  6]
[ 7 -1  8  9]
[10 11 12 -1]
```

check_bitrade_generators(*a, b, c*)

Three group elements a, b, c will generate a bitrade if a*b*c = 1 and the subgroups a, b, c intersect (pairwise) in just the identity.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: a, b, c, G = p3_group_bitrade_generators(3)
sage: check_bitrade_generators(a, b, c)
True
sage: check_bitrade_generators(a, b, gap('()'))
False
```

coin()

Simulates a fair coin (returns True or False) using ZZ.random_element(2).

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: x = coin()
sage: x == 0 or x == 1
True
```

column_containing_sym(*L, r, x*)

Given an improper latin square L with L[r, c1] = L[r, c2] = x, return c1 or c2 with equal probability. This is an internal function and should only be used in LatinSquare_generator().

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = matrix([(1, 0, 2, 3), (0, 2, 3, 0), (2, 3, 0, 1), (3, 0, 1, 2)])
sage: L
[1 0 2 3]
[0 2 3 0]
[2 3 0 1]
[3 0 1 2]
sage: c = column_containing_sym(L, 1, 0)
sage: c == 0 or c == 3
True
```

direct_product(*L1, L2, L3, L4*)

The 'direct product' of four latin squares L1, L2, L3, L4 of order n is the latin square of order 2n consisting of

```
-----------
| L1 | L2 |
-----------
| L3 | L4 |
-----------
```

where the subsquares L2 and L3 have entries offset by n.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: direct_product(back_circulant(4), back_circulant(4), elementary_abelian_2group(2), element
[0 1 2 3 4 5 6 7]
[1 2 3 0 5 6 7 4]
[2 3 0 1 6 7 4 5]
[3 0 1 2 7 4 5 6]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
```

dlxcpp_find_completions(*P, nr_to_find=None*)

Returns a list of all latin squares L of the same order as P such that P is contained in L. The optional parameter nr_to_find limits the number of latin squares that are found.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: dlxcpp_find_completions(LatinSquare(2))
[[0 1]
[1 0], [1 0]
[0 1]]


sage: dlxcpp_find_completions(LatinSquare(2), 1)
[[0 1]
[1 0]]
```

dlxcpp_rows_and_map(*P*)

Internal function for dlxcpp_find_completions. Given a partial latin square P we construct a list of rows of a 0-1 matrix M such that an exact cover of M corresponds to a completion of P to a latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: dlxcpp_rows_and_map(LatinSquare(2))
([[0, 4, 8], [1, 5, 8], [2, 4, 9], [3, 5, 9], [0, 6, 10], [1, 7, 10], [2, 6, 11], [3, 7, 11]], {
```

elementary_abelian_2group(*s*)

Returns the latin square based on the Cayley table for the elementary abelian 2-group of order 2s.

INPUT:

- s - int; order of the latin square will be 2s.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: elementary_abelian_2group(3)
[0 1 2 3 4 5 6 7]
[1 0 3 2 5 4 7 6]
[2 3 0 1 6 7 4 5]
[3 2 1 0 7 6 5 4]
[4 5 6 7 0 1 2 3]
[5 4 7 6 1 0 3 2]
[6 7 4 5 2 3 0 1]
[7 6 5 4 3 2 1 0]
```

forward_circulant(*n*)

The forward-circulant latin square of order n is the Cayley table for the operation r + c = (n-c+r) mod n.

INPUT:

- n - int; order of the latin square.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: forward_circulant(5)
[0 4 3 2 1]
[1 0 4 3 2]
[2 1 0 4 3]
```

```
[3 2 1 0 4]
[4 3 2 1 0]
```

genus(*T1, T2*)

Returns the genus of hypermap embedding associated with the bitrade (T1, T2). Informally, we compute the [tau_1, tau_2, tau_3] permutation representation of the bitrade. Each cycle of tau_1, tau_2, and tau_3 gives a rotation scheme for a black, white, and star vertex (respectively). The genus then comes from Euler's formula. For more details see Carlo Hamalainen: Partitioning 3-homogeneous latin bitrades. To appear in Geometriae Dedicata, available at http://arxiv.org/abs/0710.0938

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = alternating_group_bitrade_generators(1)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: genus(T1, T2)
1
sage: (a, b, c, G) = pq_group_bitrade_generators(3, 7)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: genus(T1, T2)
3
```

group_to_LatinSquare(*G*)

Construct a latin square on the symbols [0, 1, ..., n-1] for a group with an n by n Cayley table.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: group_to_LatinSquare(DihedralGroup(2))
[0 1 2 3]
[1 0 3 2]
[2 3 0 1]
[3 2 1 0]


sage: G = gap.Group(PermutationGroupElement((1,2,3)))
sage: group_to_LatinSquare(G)
[0 1 2]
[1 2 0]
[2 0 1]
```

is_bitrade(*T1, T2*)

Combinatorially, a pair (T1, T2) of partial latin squares is a bitrade if they are disjoint, have the same shape, and have row and column balance. For definitions of each of these terms see the relevant function in this file.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
```

is_disjoint(*T1, T2*)

The partial latin squares T1 and T2 are disjoint if T1[r, c] != T2[r, c] or T1[r, c] == T2[r, c] == -1 for each cell [r, c].

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import is_disjoint, back_circulant, isotopism
sage: is_disjoint(back_circulant(2), back_circulant(2))
False
```

```
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_disjoint(T1, T2)
True
```

is_primary_bitrade(*a, b, c, G*)

A bitrade generated from elements a, b, c is primary if a, b, c = G.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = p3_group_bitrade_generators(5)
sage: is_primary_bitrade(a, b, c, G)
True
```

is_row_and_col_balanced(*T1, T2*)

Partial latin squares T1 and T2 are balanced if the symbols appearing in row r of T1 are the same as the symbols appearing in row r of T2, for each r, and if the same condition holds on columns.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = matrix([[0,1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1]])
sage: T2 = matrix([[0,1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1]])
sage: is_row_and_col_balanced(T1, T2)
True
sage: T2 = matrix([[0,3,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1], [-1,-1,-1,-1]])
sage: is_row_and_col_balanced(T1, T2)
False
```

is_same_shape(*T1, T2*)

Two partial latin squares T1, T2 have the same shape if T1[r, c] = 0 if and only if T2[r, c] = 0.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: is_same_shape(elementary_abelian_2group(2), back_circulant(4))
True
sage: is_same_shape(LatinSquare(5), LatinSquare(5))
True
sage: is_same_shape(forward_circulant(5), LatinSquare(5))
False
```

isotopism(*p*)

Returns a Permutation object that represents an isotopism (for rows, columns or symbols of a partial latin square). Since matrices in Sage are indexed from 0, this function translates +1 to agree with the Permutation class. We also handle PermutationGroupElements.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: isotopism(5) # identity on 5 points
[1, 2, 3, 4, 5]

sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: g = G.gen(0)
sage: isotopism(g)
[2, 3, 1, 5, 4]

sage: isotopism([0,3,2,1]) # 0 goes to 0, 1 goes to 3, etc.
[1, 4, 3, 2]

sage: isotopism( (0,1,2) ) # single cycle, presented as a tuple
[2, 3, 1]

sage: x = isotopism( ((0,1,2), (3,4)) ) # tuple of cycles
sage: x
[2, 3, 1, 5, 4]
sage: x.to_cycles()
[(1, 2, 3), (4, 5)]
```

next_conjugate(*L*)

Permutes L[r, c] = e to the conjugate L[c, e] = r.

We assume that L is an n by n matrix and has values in the range 0, 1, ..., n-1.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = back_circulant(6)
sage: L
[0 1 2 3 4 5]
[1 2 3 4 5 0]
[2 3 4 5 0 1]
[3 4 5 0 1 2]
[4 5 0 1 2 3]
[5 0 1 2 3 4]
sage: next_conjugate(L)
[0 1 2 3 4 5]
[5 0 1 2 3 4]
[4 5 0 1 2 3]
[3 4 5 0 1 2]
[2 3 4 5 0 1]
[1 2 3 4 5 0]
sage: L == next_conjugate(next_conjugate(next_conjugate(L)))
True
```

p3_group_bitrade_generators(*p*)

Generators for a group of order p3 where p is a prime.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: p3_group_bitrade_generators(3)
((2,6,7)(3,8,9), (1,2,3)(4,7,8)(5,6,9), (1,9,2)(3,7,4)(5,8,6), Permutation Group with generators
```

pq_group_bitrade_generators(*p, q*)

Generators for a group of order pq where p and q are primes such that (q % p) == 1.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: pq_group_bitrade_generators(3,7)
((2,3,5)(4,7,6), (1,2,3,4,5,6,7), (1,4,2)(3,5,6), Permutation Group with generators [(2,3,5)(4,7
```

row_containing_sym(*L, c, x*)

Given an improper latin square L with L[r1, c] = L[r2, c] = x, return r1 or r2 with equal probability. This is an internal function and should only be used in LatinSquare_generator().

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: L = matrix([(0, 1, 0, 3), (3, 0, 2, 1), (1, 0, 3, 2), (2, 3, 1, 0)])
sage: L
[0 1 0 3]
[3 0 2 1]
[1 0 3 2]
[2 3 1 0]
sage: c = row_containing_sym(L, 1, 0)
sage: c == 1 or c == 2
True
```

tau1(*T1, T2, cells_map*)

The definition of τ_1 is

$$\tau_1 : T1 \to T1$$
$$\tau_1 = \beta_2^{-1}\beta_3$$

where the composition is left to right and $\beta_i : T2 \to T1$ changes just the i^{th} coordinate of a triple.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: t1 = tau1(T1, T2, cells_map)
sage: t1
[2, 3, 4, 5, 1, 7, 8, 9, 10, 6, 12, 13, 14, 15, 11, 17, 18, 19, 20, 16, 22, 23, 24, 25, 21]
sage: t1.to_cycles()
[(1, 2, 3, 4, 5), (6, 7, 8, 9, 10), (11, 12, 13, 14, 15), (16, 17, 18, 19, 20), (21, 22, 23, 24,
```

tau123(*T1, T2*)

Compute the tau_i representation for a bitrade (T1, T2). See the functions tau1, tau2, and tau3 for the mathematical definitions.

OUTPUT:

- (cells_map, t1, t2, t3)

where cells_map is a map to/from the filled cells of T1, and t1, t2, t3 are the tau1, tau2, tau3 permutations.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: (a, b, c, G) = pq_group_bitrade_generators(3, 7)
sage: (T1, T2) = bitrade_from_group(a, b, c, G)
sage: T1
[ 0  1  2 -1 -1 -1 -1]
[ 2  4  3 -1 -1 -1 -1]
[ 4  0  5 -1 -1 -1 -1]
[ 3  5  1 -1 -1 -1 -1]
[ 6  3  0 -1 -1 -1 -1]
[ 1  6  4 -1 -1 -1 -1]
[ 5  2  6 -1 -1 -1 -1]
sage: T2
[ 2  0  1 -1 -1 -1 -1]
[ 3  2  4 -1 -1 -1 -1]
[ 5  4  0 -1 -1 -1 -1]
[ 1  3  5 -1 -1 -1 -1]
[ 0  6  3 -1 -1 -1 -1]
[ 4  1  6 -1 -1 -1 -1]
[ 6  5  2 -1 -1 -1 -1]
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: cells_map
{1: (0, 0), 2: (0, 1), 3: (0, 2), 4: (1, 0), 5: (1, 1), 6: (1, 2), 7: (2, 0), 8: (2, 1), 9: (2,
sage: cells_map_as_square(cells_map, max(T1.nrows(), T1.ncols()))
[ 1  2  3 -1 -1 -1 -1]
[ 4  5  6 -1 -1 -1 -1]
[ 7  8  9 -1 -1 -1 -1]
[10 11 12 -1 -1 -1 -1]
[13 14 15 -1 -1 -1 -1]
[16 17 18 -1 -1 -1 -1]
[19 20 21 -1 -1 -1 -1]
sage: t1
[2, 3, 1, 5, 6, 4, 8, 9, 7, 11, 12, 10, 14, 15, 13, 17, 18, 16, 20, 21, 19]
sage: t2
[4, 8, 12, 10, 20, 18, 19, 5, 15, 16, 14, 9, 1, 17, 6, 7, 2, 21, 13, 11, 3]
sage: t3
[15, 16, 20, 3, 7, 14, 18, 1, 11, 6, 19, 2, 21, 10, 8, 12, 13, 5, 9, 4, 17]


sage: t1.to_cycles()
[(1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12), (13, 14, 15), (16, 17, 18), (19, 20, 21)]
sage: t2.to_cycles()
[(1, 4, 10, 16, 7, 19, 13), (2, 8, 5, 20, 11, 14, 17), (3, 12, 9, 15, 6, 18, 21)]
sage: t3.to_cycles()
[(1, 15, 8), (2, 16, 12), (3, 20, 4), (5, 7, 18), (6, 14, 10), (9, 11, 19), (13, 21, 17)]
```

The product t1*t2*t3 is the identity, i.e. it fixes every point:

```
sage: len((t1*t2*t3).fixed_points()) == T1.nr_filled_cells()
True
```

tau2 (*T1, T2, cells_map*)

The definition of τ_2 is

$$\tau_2 : T1 \to T1$$
$$\tau_2 = \beta_3^{-1}\beta_1$$

where the composition is left to right and $\beta_i : T2 \to T1$ changes just the i^{th} coordinate of a triple.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: t2 = tau2(T1, T2, cells_map)
sage: t2
[21, 22, 23, 24, 25, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
sage: t2.to_cycles()
[(1, 21, 16, 11, 6), (2, 22, 17, 12, 7), (3, 23, 18, 13, 8), (4, 24, 19, 14, 9), (5, 25, 20, 15,
```

tau3 (*T1, T2, cells_map*)

The definition of τ_3 is

$$\tau_3 : T1 \to T1$$
$$\tau_3 = \beta_1^{-1}\beta_2$$

where the composition is left to right and $\beta_i : T2 \to T1$ changes just the i^{th} coordinate of a triple.

EXAMPLES:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: is_bitrade(T1, T2)
True
sage: (cells_map, t1, t2, t3) = tau123(T1, T2)
sage: t3 = tau3(T1, T2, cells_map)
sage: t3
[10, 6, 7, 8, 9, 15, 11, 12, 13, 14, 20, 16, 17, 18, 19, 25, 21, 22, 23, 24, 5, 1, 2, 3, 4]
sage: t3.to_cycles()
[(1, 10, 14, 18, 22), (2, 6, 15, 19, 23), (3, 7, 11, 20, 24), (4, 8, 12, 16, 25), (5, 9, 13, 17,
```

tau_to_bitrade (*t1, t2, t3*)

Given permutations t1, t2, t3 that represent a latin bitrade, convert them to an explicit latin bitrade (T1, T2). The result is unique up to isotopism.

EXAMPLE:

```
sage: from sage.combinat.matrices.latin import *
sage: T1 = back_circulant(5)
sage: x = isotopism( (0,1,2,3,4) )
sage: y = isotopism(5) # identity
sage: z = isotopism(5) # identity
sage: T2 = T1.apply_isotopism(x, y, z)
sage: _, t1, t2, t3 = tau123(T1, T2)
sage: U1, U2 = tau_to_bitrade(t1, t2, t3)
sage: assert is_bitrade(U1, U2)
```

```
sage: U1
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
[4 0 1 2 3]
sage: U2
[4 0 1 2 3]
[0 1 2 3 4]
[1 2 3 4 0]
[2 3 4 0 1]
[3 4 0 1 2]
```

17.20 Lyndon words

LyndonWords(*e, k=None*)

Returns the combinatorial class of Lyndon words.

EXAMPLES: If e is an integer, then e specifies the length of the alphabet; k must also be specified in this case.

```
sage: LW = LyndonWords(3,3); LW
Lyndon words from an alphabet of size 3 of length 3
sage: LW.first()
[1, 1, 2]
sage: LW.last()
[2, 3, 3]
sage: LW.random_element()
[1, 1, 2]
sage: LW.cardinality()
8
```

If e is a (weak) composition, then it returns the class of Lyndon words that have evaluation e.

```
sage: LyndonWords([2, 0, 1]).list()
[[1, 1, 3]]
sage: LyndonWords([2, 0, 1, 0, 1]).list()
[[1, 1, 3, 5], [1, 1, 5, 3], [1, 3, 1, 5]]
sage: LyndonWords([2, 1, 1]).list()
[[1, 1, 2, 3], [1, 1, 3, 2], [1, 2, 1, 3]]
```

class **LyndonWords_evaluation**(*e*)

cardinality()

Returns the number of Lyndon words with the evaluation e.

EXAMPLES:

```
sage: LyndonWords([]).cardinality()
0
sage: LyndonWords([2,2]).cardinality()
1
sage: LyndonWords([2,3,2]).cardinality()
30
```

Check to make sure that the count matches up with the number of Lyndon words generated.

```
sage: comps = [[],[2,2],[3,2,7],[4,2]]+Compositions(4).list()
sage: lws = [ LyndonWords(comp) for comp in comps]
sage: all( [ lw.cardinality() == len(lw.list()) for lw in lws] )
True
```

class LyndonWords_nk(*n, k*)

cardinality()
TESTS:

```
sage: [ LyndonWords(3,i).cardinality() for i in range(1, 11) ]
[3, 3, 8, 18, 48, 116, 312, 810, 2184, 5880]
```

StandardBracketedLyndonWords(*n, k*)
Returns the combinatorial class of standard bracketed Lyndon words from [1, ..., n] of length k. These are in one to one correspondence with the Lyndon words and form a basis for the subspace of degree k of the free Lie algebra of rank n.

EXAMPLES:

```
sage: SBLW33 = StandardBracketedLyndonWords(3,3); SBLW33
Standard bracketed Lyndon words from an alphabet of size 3 of length 3
sage: SBLW33.first()
[1, [1, 2]]
sage: SBLW33.last()
[[2, 3], 3]
sage: SBLW33.cardinality()
8
sage: SBLW33.random_element()
[1, [1, 2]]
```

class StandardBracketedLyndonWords_nk(*n, k*)

cardinality()
EXAMPLES:

```
sage: StandardBracketedLyndonWords(3, 3).cardinality()
8
sage: StandardBracketedLyndonWords(3, 4).cardinality()
18
```

standard_bracketing(*lw*)
Returns the standard bracketing of a Lyndon word lw.

EXAMPLES:

```
sage: import sage.combinat.lyndon_word as lyndon_word
sage: map( lyndon_word.standard_bracketing, LyndonWords(3,3) )
[[1, [1, 2]],
 [1, [1, 3]],
 [[1, 2], 2],
 [1, [2, 3]],
 [[1, 3], 2],
 [[1, 3], 3],
 [2, [2, 3]],
 [[2, 3], 3]]
```

17.21 Necklaces

The algorithm used in this file comes from

- Sawada, Joe. "A fast algorithm to generate necklaces with fixed content", Source Theoretical Computer Science archive Volume 301 , Issue 1-3 (May 2003)

Necklaces(*e*)
Returns the combinatorial class of necklaces with evaluation e.

EXAMPLES:

```
sage: Necklaces([2,1,1])
Necklaces with evaluation [2, 1, 1]
sage: Necklaces([2,1,1]).cardinality()
3
sage: Necklaces([2,1,1]).first()
[1, 1, 2, 3]
sage: Necklaces([2,1,1]).last()
[1, 2, 1, 3]
sage: Necklaces([2,1,1]).list()
[[1, 1, 2, 3], [1, 1, 3, 2], [1, 2, 1, 3]]
```

class Necklaces_evaluation(*e*)

cardinality()
Returns the number of integer necklaces with the evaluation e.

EXAMPLES:

```
sage: Necklaces([]).cardinality()
0
sage: Necklaces([2,2]).cardinality()
2
sage: Necklaces([2,3,2]).cardinality()
30
```

Check to make sure that the count matches up with the number of Lyndon words generated.

```
sage: comps = [[],[2,2],[3,2,7],[4,2]]+Compositions(4).list()
sage: ns = [ Necklaces(comp) for comp in comps]
sage: all( [ n.cardinality() == len(n.list()) for n in ns] )
True
```

17.22 Partitions

A partition p of a nonnegative integer n is a non-increasing list of positive integers (the *parts* of the partition) with total sum n.

A partition can be depicted by a diagram made of rows of boxes, where the number of boxes in the i^{th} row starting from the top is the i^{th} part of the partition.

The coordinate system related to a partition applies from the top to the bottom and from left to right. So, the corners of the partition are [[0,4], [1,2], [2,0]].

AUTHORS:

- Mike Hansen (2007): initial version

- Dan Drake (2009-03-28): deprecate RestrictedPartitions and implement Partitions_parts_in

EXAMPLES: There are 5 partitions of the integer 4.

```
sage: Partitions(4).cardinality()
5
sage: Partitions(4).list()
[[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

We can use the method .first() to get the 'first' partition of a number.

```
sage: Partitions(4).first()
[4]
```

Using the method .next(), we can calculate the 'next' partition. When we are at the last partition, None will be returned.

```
sage: Partitions(4).next([4])
[3, 1]
sage: Partitions(4).next([1,1,1,1]) is None
True
```

We can use `iter` to get an object which iterates over the partitions one by one to save memory. Note that when we do something like `for part in Partitions(4)` this iterator is used in the background.

```
sage: g = iter(Partitions(4))


sage: g.next()
[4]
sage: g.next()
[3, 1]
sage: g.next()
[2, 2]
sage: for p in Partitions(4): print p
[4]
[3, 1]
[2, 2]
[2, 1, 1]
[1, 1, 1, 1]
```

We can add constraints to to the type of partitions we want. For example, to get all of the partitions of 4 of length 2, we'd do the following:

```
sage: Partitions(4, length=2).list()
[[3, 1], [2, 2]]
```

Here is the list of partitions of length at least 2 and the list of ones with length at most 2:

```
sage: Partitions(4, min_length=2).list()
[[3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
sage: Partitions(4, max_length=2).list()
[[4], [3, 1], [2, 2]]
```

The options min_part and max_part can be used to set constraints on the sizes of all parts. Using max_part, we can select partitions having only 'small' entries. The following is the list of the partitions of 4 with parts at most 2:

```
sage: Partitions(4, max_part=2).list()
[[2, 2], [2, 1, 1], [1, 1, 1, 1]]
```

The min_part options is complementary to max_part and selects partitions having only 'large' parts. Here is the list of all partitions of 4 with each part at least 2.

```
sage: Partitions(4, min_part=2).list()
[[4], [2, 2]]
```

The options inner and outer can be used to set part-by-part constraints. This is the list of partitions of 4 with [3, 1, 1] as an outer bound:

```
sage: Partitions(4, outer=[3,1,1]).list()
[[3, 1], [2, 1, 1]]
```

Outer sets max_length to the length of its argument. Moreover, the parts of outer may be infinite to clear constraints on specific parts. Here is the list of the partitions of 4 of length at most 3 such that the second and third part are 1 when they exist:

```
sage: Partitions(4, outer=[oo,1,1]).list()
[[4], [3, 1], [2, 1, 1]]
```

Finally, here are the partitions of 4 with [1,1,1] as an inner bound. Note that inner sets min_length to the length of its argument.

```
sage: Partitions(4, inner=[1,1,1]).list()
[[2, 1, 1], [1, 1, 1, 1]]
```

The options min_slope and max_slope can be used to set constraints on the slope, that is on the difference p[i+1]-p[i] of two consecutive parts. Here is the list of the strictly decreasing partitions of 4:

```
sage: Partitions(4, max_slope=-1).list()
[[4], [3, 1]]
```

The constraints can be combined together in all reasonable ways. Here are all the partitions of 11 of length between 2 and 4 such that the difference between two consecutive parts is between -3 and -1:

```
sage: Partitions(11,min_slope=-3,max_slope=-1,min_length=2,max_length=4).list()
[[7, 4], [6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2], [5, 3, 2, 1]]
```

Partition objects can also be created individually with the Partition function.

```
sage: Partition([2,1])
[2, 1]
```

Once we have a partition object, then there are a variety of methods that we can use. For example, we can get the conjugate of a partition. Geometrically, the conjugate of a partition is the reflection of that partition through its main diagonal. Of course, this operation is an involution.

```
sage: Partition([4,1]).conjugate()
[2, 1, 1, 1]
sage: Partition([4,1]).conjugate().conjugate()
[4, 1]
```

We can go back and forth between the exponential notations of a partition. The exponential notation can be padded with extra zeros.

```
sage: Partition([6,4,4,2,1]).to_exp()
[1, 1, 0, 2, 0, 1]
sage: Partition(exp=[1,1,0,2,0,1])
[6, 4, 4, 2, 1]
sage: Partition([6,4,4,2,1]).to_exp(5)
[1, 1, 0, 2, 0, 1]
sage: Partition([6,4,4,2,1]).to_exp(7)
[1, 1, 0, 2, 0, 1, 0]
sage: Partition([6,4,4,2,1]).to_exp(10)
[1, 1, 0, 2, 0, 1, 0, 0, 0, 0]
```

We can get the coordinates of the corners of a partition.

```
sage: Partition([4,3,1]).corners()
[[0, 3], [1, 2], [2, 0]]
```

We can compute the r-core and r-quotient of a partition and build the partition back up from them.

```
sage: Partition([6,3,2,2]).r_core(3)
[2, 1, 1]
sage: Partition([7,7,5,3,3,3,1]).r_quotient(3)
[[2], [1], [2, 2, 2]]
sage: p = Partition([11,5,5,3,2,2,2])
sage: p.r_core(3)
[]
sage: p.r_quotient(3)
[[2, 1], [4], [1, 1, 1]]
sage: Partition(core_and_quotient=([],[[2, 1], [4], [1, 1, 1]]))
[11, 5, 5, 3, 2, 2, 2]
```

OrderedPartitions(*n, k=None*)

Returns the combinatorial class of ordered partitions of n. If k is specified, then only the ordered partitions of length k are returned.

Note: It is recommended that you use Compositions instead as OrderedPartitions wraps GAP. See also ordered_partitions.

EXAMPLES:

```
sage: OrderedPartitions(3)
Ordered partitions of 3
sage: OrderedPartitions(3).list()
[[3], [2, 1], [1, 2], [1, 1, 1]]
sage: OrderedPartitions(3,2)
Ordered partitions of 3 of length 2
sage: OrderedPartitions(3,2).list()
[[2, 1], [1, 2]]
```

class OrderedPartitions_nk(*n, k=None*)

cardinality()

EXAMPLES:

```
sage: OrderedPartitions(3).cardinality()
4
sage: OrderedPartitions(3,2).cardinality()
2
```

list()

EXAMPLES:

```
sage: OrderedPartitions(3).list()
[[3], [2, 1], [1, 2], [1, 1, 1]]
sage: OrderedPartitions(3,2).list()
[[2, 1], [1, 2]]
```

Partition(*l=None, exp=None, core_and_quotient=None*)

Returns a partition object.

Note that Sage uses the English convention for partitions and tableaux.

EXAMPLES:

```
sage: Partition(exp=[2,1,1])
[3, 2, 1, 1]
sage: Partition(core_and_quotient=([2,1], [[2,1],[3],[1,1,1]]))
[11, 5, 5, 3, 2, 2, 2]
sage: Partition([3,2,1])
[3, 2, 1]
sage: [2,1] in Partitions()
True
sage: [2,1,0] in Partitions()
False
sage: Partition([2,1,0])
[2, 1]
sage: Partition([1,2,3])
...
ValueError: [1, 2, 3] is not a valid partition
```

PartitionTuples(*n, k*)

Returns the combinatorial class of k-tuples of partitions of n. These are are ordered list of k partitions whose sizes add up to

TODO: reimplement in term of species.ProductSpecies and Partitions

These describe the classes and the characters of wreath products of groups with k conjugacy classes with the symmetric group S_n.

EXAMPLES:

```
sage: PartitionTuples(4,2)
2-tuples of partitions of 4
sage: PartitionTuples(3,2).list()
[[[3], []],
 [[2, 1], []],
 [[1, 1, 1], []],
 [[2], [1]],
 [[1, 1], [1]],
 [[1], [2]],
 [[1], [1, 1]],
 [[], [3]],
 [[], [2, 1]],
 [[], [1, 1, 1]]]
```

class **PartitionTuples_nk**(*n, k*)

cardinality()

Returns the number of k-tuples of partitions which together form a partition of n.

Wraps GAP's NrPartitionTuples.

EXAMPLES:

```
sage: PartitionTuples(3,2).cardinality()
10
sage: PartitionTuples(8,2).cardinality()
185
```

Now we compare that with the result of the following GAP computation:

```
gap> S8:=Group((1,2,3,4,5,6,7,8),(1,2));
Group([ (1,2,3,4,5,6,7,8), (1,2) ])
gap> C2:=Group((1,2));
Group([ (1,2) ])
gap> W:=WreathProduct(C2,S8);
<permutation group of size 10321920 with 10 generators>
gap> Size(W);
10321920     ## = 2^8*Factorial(8), which is good:-)
gap> Size(ConjugacyClasses(W));
185
```

object_class()

class **Partition_class**(*l*)

add_box(*i, j=None*)

Returns a partition corresponding to self with a box added in row i. i and j are 0-based row and column indices. This does not change p.

Note that if you have coordinates in a list, you can call this function with python's * notation (see the examples below).

EXAMPLES:

```
sage: Partition([3, 2, 1, 1]).add_box(0)
[4, 2, 1, 1]
sage: cell = [4, 0]; Partition([3, 2, 1, 1]).add_box(*cell)
[3, 2, 1, 1, 1]
```

add_horizontal_border_strip(*k*)

Returns a list of all the partitions that can be obtained by adding a horizontal border strip of length k to self.

EXAMPLES:

```
sage: Partition([]).add_horizontal_border_strip(0)
[[]]
sage: Partition([]).add_horizontal_border_strip(2)
[[2]]
sage: Partition([2,2]).add_horizontal_border_strip(2)
[[2, 2, 2], [3, 2, 1], [4, 2]]
sage: Partition([3,2,2]).add_horizontal_border_strip(2)
[[3, 2, 2, 2], [3, 3, 2, 1], [4, 2, 2, 1], [4, 3, 2], [5, 2, 2]]
```

TODO: reimplement like remove_horizontal_border_strip using IntegerListsLex

add_vertical_border_strip(*k*)

Returns a list of all the partitions that can be obtained by adding a vertical border strip of length k to self.

EXAMPLES:

```
sage: Partition([]).add_vertical_border_strip(0)
[[]]
sage: Partition([]).add_vertical_border_strip(2)
[[1, 1]]
sage: Partition([2,2]).add_vertical_border_strip(2)
[[3, 3], [3, 2, 1], [2, 2, 1, 1]]
sage: Partition([3,2,2]).add_vertical_border_strip(2)
[[4, 3, 2], [4, 2, 2, 1], [3, 3, 3], [3, 3, 2, 1], [3, 2, 2, 1, 1]]
```

arm(*i, j*)

Returns the arm of cell (i,j) in partition p. The arm of cell (i,j) is the number of boxes that appear to the right of cell (i,j). Note that i and j are 0-based indices. If your coordinates are in the form [i,j], use Python's *-operator.

EXAMPLES:

```
sage: p = Partition([2,2,1])
sage: p.arm(0, 0)
1
sage: p.arm(0, 1)
0
sage: p.arm(2, 0)
0
sage: Partition([3,3]).arm(0, 0)
2
sage: Partition([3,3]).arm(*[0,0])
2
```

arm_lengths(*flat=False*)

Returns a tableau of shape p where each box is filled its arm. The optional boolean parameter flat provides the option of returning a flat list.

EXAMPLES:

```
sage: Partition([2,2,1]).arm_lengths()
[[1, 0], [1, 0], [0]]
sage: Partition([2,2,1]).arm_lengths(flat=True)
[1, 0, 1, 0, 0]
sage: Partition([3,3]).arm_lengths()
[[2, 1, 0], [2, 1, 0]]
sage: Partition([3,3]).arm_lengths(flat=True)
[2, 1, 0, 2, 1, 0]
```

arms_legs_coeff(*i, j*)

This is a statistic on a cell c=[i,j] in the diagram of partition p given by

$$[(1 - q^{arm(c)} * t^{leg(c)+1})]/[(1 - q^{arm(c)+1} * t^{leg(c)})]$$

EXAMPLES:

```
sage: Partition([3,2,1]).arms_legs_coeff(1,1)
(-t + 1)/(-q + 1)
sage: Partition([3,2,1]).arms_legs_coeff(0,0)
(-q^2*t^3 + 1)/(-q^3*t^2 + 1)
sage: Partition([3,2,1]).arms_legs_coeff(*[0,0])
(-q^2*t^3 + 1)/(-q^3*t^2 + 1)
```

associated()

An alias for partition.conjugate(p).

EXAMPLES:

```
sage: Partition([4,1,1]).associated()
[3, 1, 1, 1]
sage: Partition([4,1,1]).conjugate()
[3, 1, 1, 1]
sage: Partition([5,4,2,1,1,1]).associated().associated()
[5, 4, 2, 1, 1, 1]
```

atom()

Returns a list of the standard tableaux of size self.size() whose atom is equal to self.

EXAMPLES:

```
sage: Partition([2,1]).atom()
[[[1, 2], [3]]]
sage: Partition([3,2,1]).atom()
[[[1, 2, 3, 6], [4, 5]], [[1, 2, 3], [4, 5], [6]]]
```

aut()

Returns a factor for the number of permutations with cycle type self. self.aut() returns $1^{j_1} j_1! \cdots n^{j_n} j_n!$ where j_k is the number of parts in self equal to k.

The number of permutations having p as a cycle type is given by

$$\frac{n!}{1^{j_1} j_1! \cdots n^{j_n} j_n!}.$$

EXAMPLES:

```
sage: Partition([2,1]).aut()
2
```

boxes()

Return the coordinates of the boxes of self. Coordinates are given as (row-index, column-index) and are 0 based.

EXAMPLES:

```
sage: Partition([2,2]).boxes()
[(0, 0), (0, 1), (1, 0), (1, 1)]
sage: Partition([3,2]).boxes()
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1)]
```

centralizer_size(*t=0, q=0*)

Returns the size of the centralizer of any permutation of cycle type p. If m_i is the multiplicity of i as a part of p, this is given by $\prod_i (i^{m[i]}) * (m[i]!)$. Including the optional parameters t and q gives the q-t analog which is the former product times $\prod_{i=1}^{length(p)} (1 - q^{p[i]})/(1 - t^{p[i]})$.

EXAMPLES:

```
sage: Partition([2,2,1]).centralizer_size()
8
sage: Partition([2,2,2]).centralizer_size()
48
```

REFERENCES:

•Kerber, A. 'Algebraic Combinatorics via Finite Group Action', 1.3 p24

character_polynomial()

Returns the character polynomial associated to the partition self. The character polynomial q_μ is defined by

$$q_\mu(x_1, x_2, \ldots, x_k) =\downarrow \sum_{\alpha \vdash k} \frac{\chi^\mu_\alpha}{1^{a_1} 2^{a_2} \cdots k^{a_k} a_1! a_2! \cdots a_k!} \prod_{i=1}^{k} (i x_i - 1)^{a_i}$$

where a_i is the multiplicity of i in α.

It is computed in the following manner.

1.Expand the Schur function s_μ in the power-sum basis.

2.Replace each p_i with $ix_i - 1$

3.Apply the umbral operator $\downarrow$ to the resulting polynomial.

EXAMPLES:

```
sage: Partition([1]).character_polynomial()
x - 1
sage: Partition([1,1]).character_polynomial()
1/2*x0^2 - 3/2*x0 - x1 + 1
sage: Partition([2,1]).character_polynomial()
1/3*x0^3 - 2*x0^2 + 8/3*x0 - x2
```

conjugacy_class_size()

Returns the size of the conjugacy class of the symmetric group indexed by the partition p.

EXAMPLES:

```
sage: Partition([2,2,2]).conjugacy_class_size()
15
sage: Partition([2,2,1]).conjugacy_class_size()
15
sage: Partition([2,1,1]).conjugacy_class_size()
6
```

REFERENCES:

•Kerber, A. 'Algebraic Combinatorics via Finite Group Action' 1.3 p24

conjugate()

conjugate() returns the "conjugate" (also called "associated" in the literature) partition of the partition p which is obtained by transposing the corresponding Ferrers diagram.

EXAMPLES:

```
sage: Partition([2,2]).conjugate()
[2, 2]
sage: Partition([6,3,1]).conjugate()
[3, 2, 2, 1, 1, 1]
sage: print Partition([6,3,1]).ferrers_diagram()
******
***
*
sage: print Partition([6,3,1]).conjugate().ferrers_diagram()
***
**
**
*
*
*
```

contains(x)

Returns True if p2 is a partition whose Ferrers diagram is contained in the Ferrers diagram of self

EXAMPLES:

```
sage: p = Partition([3,2,1])
sage: p.contains([2,1])
True
sage: all(p.contains(mu) for mu in Partitions(3))
True
```

```
sage: all(p.contains(mu) for mu in Partitions(4))
False
```

content(*i, j*)

Returns the content statistic of the given cell, which is simply defined by j - i. This doesn't technically depend on the partition, but is included here because it is often useful in the context of partitions.

EXAMPLES:

```
sage: Partition([2,1]).content(0,1)
1
sage: p = Partition([3,2])
sage: sum([p.content(*c) for c in p.boxes()])
2
```

corners()

Returns a list of the corners of the partitions. These are the positions where we can remove a box. Indices are of the form [i,j] where i is the row-index and j is the column-index, and are 0-based.

EXAMPLES:

```
sage: Partition([3,2,1]).corners()
[[0, 2], [1, 1], [2, 0]]
sage: Partition([3,3,1]).corners()
[[1, 2], [2, 0]]
```

dominate(*rows=None*)

Returns a list of the partitions dominated by n. If n is specified, then it only returns the ones with = rows rows.

EXAMPLES:

```
sage: Partition([3,2,1]).dominate()
[[3, 2, 1], [3, 1, 1, 1], [2, 2, 2], [2, 2, 1, 1], [2, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1]]
sage: Partition([3,2,1]).dominate(rows=3)
[[3, 2, 1], [2, 2, 2]]
```

dominates(*p2*)

Returns True if partition p1 dominates partitions p2. Otherwise, it returns False.

EXAMPLES:

```
sage: p = Partition([3,2])
sage: p.dominates([3,1])
True
sage: p.dominates([2,2])
True
sage: p.dominates([2,1,1])
True
sage: p.dominates([3,3])
False
sage: p.dominates([4])
False
sage: Partition([4]).dominates(p)
False
sage: Partition([]).dominates([1])
False
sage: Partition([]).dominates([])
True
sage: Partition([1]).dominates([])
True
```

down()

Returns a generator for partitions that can be obtained from p by removing a box.

EXAMPLES:

```
sage: [p for p in Partition([2,1,1]).down()]
[[1, 1, 1], [2, 1]]
sage: [p for p in Partition([3,2]).down()]
[[2, 2], [3, 1]]
sage: [p for p in Partition([3,2,1]).down()]
[[2, 2, 1], [3, 1, 1], [3, 2]]
```

down_list()

Returns a list of the partitions that can be obtained from the partition p by removing a box.

EXAMPLES:

```
sage: Partition([2,1,1]).down_list()
[[1, 1, 1], [2, 1]]
sage: Partition([3,2]).down_list()
[[2, 2], [3, 1]]
sage: Partition([3,2,1]).down_list()
[[2, 2, 1], [3, 1, 1], [3, 2]]
```

evaluation()

Returns the evaluation of the partition.

EXAMPLES:

```
sage: Partition([4,3,1,1]).evaluation()
[2, 0, 1, 1]
```

ferrers_diagram()

Return the Ferrers diagram of pi.

INPUT:

- pi - a partition, given as a list of integers.

EXAMPLES:

```
sage: print Partition([5,5,2,1]).ferrers_diagram()
*****
*****
**
*
sage: pi = Partitions(10).list()[11] ## [6,1,1,1,1]
sage: print pi.ferrers_diagram()
******
*
*
*
*
sage: pi = Partitions(10).list()[8] ## [6, 3, 1]
sage: print pi.ferrers_diagram()
******
***
*
```

generalized_pochhammer_symbol(*a*, *alpha*)

Returns the generalized Pochhammer symbol $(a)^{(\alpha)}_{self}$. This is the product over all cells (i,j) in p of $a-(i-1)/\alpha+j-1$.

EXAMPLES:

```
sage: Partition([2,2]).generalized_pochhammer_symbol(2,1)
12
```

hook(*i, j*)

Returns the hook of box (i,j) in the partition p. The hook of box (i,j) is defined to be one more than the sum of number of boxes to the right and the number of boxes below (in the English convention). Note that i and j are 0-based. If your coordinates are in the form [i,j], use Python's *-operator.

EXAMPLES:

```
sage: p = Partition([2,2,1])
sage: p.hook(0, 0)
4
sage: p.hook(0, 1)
2
sage: p.hook(2, 0)
1
sage: Partition([3,3]).hook(0, 0)
4
sage: cell = [0,0]; Partition([3,3]).hook(*cell)
4
```

hook_lengths()

Returns a tableau of shape p with the boxes filled in with the hook lengths.

In each box, put the sum of one plus the number of boxes horizontally to the right and vertically below the box (the hook length).

For example, consider the partition [3,2,1] of 6 with Ferrers Diagram:

```
# # #
# #
#
```

When we fill in the boxes with the hook lengths, we obtain:

```
5 3 1
3 1
1
```

EXAMPLES:

```
sage: Partition([2,2,1]).hook_lengths()
[[4, 2], [3, 1], [1]]
sage: Partition([3,3]).hook_lengths()
[[4, 3, 2], [3, 2, 1]]
sage: Partition([3,2,1]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
sage: Partition([2,2]).hook_lengths()
[[3, 2], [2, 1]]
sage: Partition([5]).hook_lengths()
[[5, 4, 3, 2, 1]]
```

REFERENCES:

- http://mathworld.wolfram.com/HookLengthFormula.html

hook_polynomial(*q, t*)

Returns the two-variable hook polynomial.

EXAMPLES:

```
sage: R.<q,t> = PolynomialRing(QQ)
sage: a = Partition([2,2]).hook_polynomial(q,t)
sage: a == (1 - t)*(1 - q*t)*(1 - t^2)*(1 - q*t^2)
True
sage: a = Partition([3,2,1]).hook_polynomial(q,t)
sage: a == (1 - t)^3*(1 - q*t^2)^2*(1 - q^2*t^3)
True
```

hook_product(*a*)

Returns the Jack hook-product.

EXAMPLES:

```
sage: Partition([3,2,1]).hook_product(x)
(x + 2)^2*(2*x + 3)
sage: Partition([2,2]).hook_product(x)
2*(x + 1)*(x + 2)
```

hooks()

Returns a sorted list of the hook lengths in self.

EXAMPLES:

```
sage: Partition([3,2,1]).hooks()
[5, 3, 3, 1, 1, 1]
```

jacobi_trudi()

EXAMPLES:

```
sage: part = Partition([3,2,1])
sage: jt = part.jacobi_trudi(); jt
[h[3] h[1]    0]
[h[4] h[2]  h[]]
[h[5] h[3] h[1]]
sage: s = SFASchur(QQ)
sage: h = SFAHomogeneous(QQ)
sage: h( s(part) )
h[3, 2, 1] - h[3, 3] - h[4, 1, 1] + h[5, 1]
sage: jt.det()
h[3, 2, 1] - h[3, 3] - h[4, 1, 1] + h[5, 1]
```

k_atom(*k*)

EXAMPLES:

```
sage: p = Partition([3,2,1])
sage: p.k_atom(1)
[]
sage: p.k_atom(3)
[[[1, 1, 1], [2, 2], [3]],
 [[1, 1, 1, 2], [2], [3]],
 [[1, 1, 1, 3], [2, 2]],
 [[1, 1, 1, 2, 3], [2]]]
sage: Partition([3,2,1]).k_atom(4)
[[[1, 1, 1], [2, 2], [3]], [[1, 1, 1, 3], [2, 2]]]
```

TESTS:

```
sage: Partition([1]).k_atom(1)
[[[1]]]
sage: Partition([1]).k_atom(2)
[[[1]]]
sage: Partition([]).k_atom(1)
[[]]
```

k_conjugate(*k*)

Returns the k-conjugate of the partition.

The k-conjugate is the partition that is given by the columns of the k-skew diagram of the partition.

EXAMPLES:

```
sage: p = Partition([4,3,2,2,1,1])
sage: p.k_conjugate(4)
[3, 2, 2, 1, 1, 1, 1, 1, 1]
```

k_skew(*k*)

Returns the k-skew partition.

The k-skew diagram of a k-bounded partition is the skew diagram denoted $\lambda/^k$ satisfying the conditions:

1.row i of $\lambda/^k$ has length λ_i

2.no cell in $\lambda/^k$ has hook-length exceeding k

3.every square above the diagram of $\lambda/^k$ has hook length exceeding k.

REFERENCES:

•Lapointe, L. and Morse, J. 'Order Ideals in Weak Subposets of Young's Lattice and Associated Unimodality Conjectures'

EXAMPLES:

```
sage: p = Partition([4,3,2,2,1,1])
sage: p.k_skew(4)
[[9, 5, 3, 2, 1, 1], [5, 2, 1]]
```

k_split(*k*)

Returns the k-split of self.

EXAMPLES:

```
sage: Partition([4,3,2,1]).k_split(3)
[]
sage: Partition([4,3,2,1]).k_split(4)
[[4], [3, 2], [1]]
sage: Partition([4,3,2,1]).k_split(5)
[[4, 3], [2, 1]]
sage: Partition([4,3,2,1]).k_split(6)
[[4, 3, 2], [1]]
sage: Partition([4,3,2,1]).k_split(7)
[[4, 3, 2, 1]]
sage: Partition([4,3,2,1]).k_split(8)
[[4, 3, 2, 1]]
```

leg(*i, j*)

Returns the leg of box (i,j) in partition p. The leg of box (i,j) is defined to be the number of boxes below it in partition p. Note that i and j are 0-based. If your coordinates are in the form [i,j], use Python's *-operator.

EXAMPLES:

```
sage: p = Partition([2,2,1])
sage: p.leg(0, 0)
2
sage: p.leg(0,1)
1
sage: p.leg(2,0)
0
sage: Partition([3,3]).leg(0, 0)
1
sage: cell = [0,0]; Partition([3,3]).leg(*cell)
1
```

leg_lengths(*flat=False*)

Returns a tableau of shape p with each box filled in with its leg. The optional boolean parameter flat provides the option of returning a flat list.

EXAMPLES:

```
sage: Partition([2,2,1]).leg_lengths()
[[2, 1], [1, 0], [0]]
sage: Partition([2,2,1]).leg_lengths(flat=True)
[2, 1, 1, 0, 0]
sage: Partition([3,3]).leg_lengths()
[[1, 1, 1], [0, 0, 0]]
sage: Partition([3,3]).leg_lengths(flat=True)
[1, 1, 1, 0, 0, 0]
```

length()

Returns the number of parts in self.

EXAMPLES:

```
sage: Partition([3,2]).length()
2
sage: Partition([2,2,1]).length()
3
sage: Partition([]).length()
0
```

lower_hook(*i, j, alpha*)

Returns the lower hook length of the box (i,j) in self. When alpha == 1, this is just the normal hook length. Indices are 0-based.

The lower hook length of a box (i,j) in a partition κ is defined by

$$h^\kappa_*(i,j) = \kappa'_j - i + 1 + \alpha(\kappa_i - j).$$

EXAMPLES:

```
sage: p = Partition([2,1])
sage: p.lower_hook(0,0,1)
3
sage: p.hook(0,0)
3
sage: [ p.lower_hook(i,j,x) for i,j in p.boxes() ]
[x + 2, 1, 1]
```

lower_hook_lengths(*alpha*)

Returns the lower hook lengths of the partition. When alpha == 1, these are just the normal hook lengths.

The lower hook length of a box (i,j) in a partition κ is defined by

$$h^*_\kappa(i,j) = \kappa'_j - i + \alpha(\kappa_i - j + 1).$$

EXAMPLES:

```
sage: Partition([3,2,1]).lower_hook_lengths(x)
[[2*x + 3, x + 2, 1], [x + 2, 1], [1]]
sage: Partition([3,2,1]).lower_hook_lengths(1)
[[5, 3, 1], [3, 1], [1]]
sage: Partition([3,2,1]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
```

next()

Returns the partition that lexicographically follows the partition p. If p is the last partition, then it returns False.

EXAMPLES:

```
sage: Partition([4]).next()
[3, 1]
sage: Partition([1,1,1,1]).next()
False
```

outside_corners()

Returns a list of the positions where we can add a box so that the shape is still a partition. Indices are of the form [i,j] where i is the row-index and j is the column-index, and are 0-based.

EXAMPLES:

```
sage: Partition([2,2,1]).outside_corners()
[[0, 2], [2, 1], [3, 0]]
sage: Partition([2,2]).outside_corners()
[[0, 2], [2, 0]]
sage: Partition([6,3,3,1,1,1]).outside_corners()
[[0, 6], [1, 3], [3, 1], [6, 0]]
sage: Partition([]).outside_corners()
[[0, 0]]
```

parent()

Returns the combinatorial class of partitions of sum(self).

EXAMPLES:

```
sage: Partition([3,2,1]).parent()
Partitions of the integer 6
```

power(k)

partition_power(pi, k) returns the partition corresponding to the k-th power of a permutation with cycle structure pi (thus describes the powermap of symmetric groups).

Wraps GAP's PowerPartition.

EXAMPLES:

```
sage: p = Partition([5,3])
sage: p.power(1)
[5, 3]
sage: p.power(2)
[5, 3]
sage: p.power(3)
[5, 1, 1, 1]
sage: p.power(4)
[5, 3]
sage: Partition([3,2,1]).power(3)
[2, 1, 1, 1, 1]
```

Now let us compare this to the power map on S_8:

```
sage: G = SymmetricGroup(8)
sage: g = G([(1,2,3,4,5),(6,7,8)])
sage: g
(1,2,3,4,5)(6,7,8)
sage: g^2
(1,3,5,2,4)(6,8,7)
sage: g^3
(1,4,2,5,3)
sage: g^4
(1,5,4,3,2)(6,7,8)
```

pp()

EXAMPLES:

```
sage: Partition([5,5,2,1]).pp()
*****
*****
**
*
```

r_core(*length*)

Returns the r-core of the partition p. The construction of the r-core can be visualized by repeatedly removing border strips of size r from p until this is no longer possible. The remaining partition is the r-core.

EXAMPLES:

```
sage: Partition([6,3,2,2]).r_core(3)
[2, 1, 1]
sage: Partition([]).r_core(3)
[]
sage: Partition([8,7,7,4,1,1,1,1,1]).r_core(3)
[2, 1, 1]
```

TESTS:

```
sage: Partition([3,3,3,2,1]).r_core(3)
[]
sage: Partition([10,8,7,7]).r_core(4)
[]
sage: Partition([21,15,15,9,6,6,6,3,3]).r_core(3)
[]
```

r_quotient(*length*)

Returns the r-quotient of the partition p. The r-quotient is a list of r partitions, constructed in the following way. Label each cell in p with its content, modulo r. Let R_i be the set of rows ending in a box labelled i, and C_i be the set of columns ending in a box labelled i. Then the jth component of the r-quotient of p is the partition defined by intersecting R_j with C_j+1.

EXAMPLES:

```
sage: Partition([7,7,5,3,3,3,1]).r_quotient(3)
[[2], [1], [2, 2, 2]]
```

TESTS:

```
sage: Partition([8,7,7,4,1,1,1,1,1]).r_quotient(3)
[[2, 1], [2, 2], [2]]
sage: Partition([10,8,7,7]).r_quotient(4)
[[2], [3], [2], [1]]
sage: Partition([6,3,3]).r_quotient(3)
[[1], [1], [2]]
sage: Partition([3,3,3,2,1]).r_quotient(3)
[[1], [1, 1], [1]]
sage: Partition([6,6,6,3,3,3]).r_quotient(3)
[[2, 1], [2, 1], [2, 1]]
sage: Partition([21,15,15,9,6,6,6,3,3]).r_quotient(3)
[[5, 2, 1], [5, 2, 1], [7, 3, 2]]
sage: Partition([21,15,15,9,6,6,3,3]).r_quotient(3)
[[5, 2], [5, 2, 1], [7, 3, 1]]
sage: Partition([14,12,11,10,10,10,10,9,6,4,3,3,2,1]).r_quotient(5)
[[3, 3], [2, 2, 1], [], [3, 3, 3], [1]]
```

reading_tableau()

EXAMPLES:

```
sage: Partition([3,2,1]).reading_tableau()
[[1, 3, 6], [2, 5], [4]]
```

remove_box(*i, j=None*)

Returns the partition obtained by removing a box at the end of row i.

EXAMPLES:

```
sage: Partition([2,2]).remove_box(1)
[2, 1]
sage: Partition([2,2,1]).remove_box(2)
[2, 2]
sage: #Partition([2,2]).remove_box(0)

sage: Partition([2,2]).remove_box(1,1)
[2, 1]
sage: #Partition([2,2]).remove_box(1,0)
```

remove_horizontal_border_strip(*k*)

Returns the partitions obtained from self by removing an horizontal border strip of length k

EXAMPLES:

```
sage: Partition([5,3,1]).remove_horizontal_border_strip(0).list()
[[5, 3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(1).list()
[[5, 3], [5, 2, 1], [4, 3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(2).list()
[[5, 2], [5, 1, 1], [4, 3], [4, 2, 1], [3, 3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(3).list()
[[5, 1], [4, 2], [4, 1, 1], [3, 3], [3, 2, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(4).list()
[[4, 1], [3, 2], [3, 1, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(5).list()
[[3, 1]]
sage: Partition([5,3,1]).remove_horizontal_border_strip(6).list()
[]
```

The result is returned as a combinatorial class:

```
sage: Partition([5,3,1]).remove_horizontal_border_strip(5)
The subpartitions of [5, 3, 1] obtained by removing an horizontal border strip of length 5
```

TESTS:

```
sage: Partition([3,2,2]).remove_horizontal_border_strip(2).list()
[[3, 2], [2, 2, 1]]
sage: Partition([3,2,2]).remove_horizontal_border_strip(2).first().parent()
Partitions...
sage: Partition([]).remove_horizontal_border_strip(0).list()
[[]]
sage: Partition([]).remove_horizontal_border_strip(6).list()
[]
```

sign()

partition_sign(pi) returns the sign of a permutation with cycle structure given by the partition pi.

This function corresponds to a homomorphism from the symmetric group S_n into the cyclic group of order 2, whose kernel is exactly the alternating group A_n. Partitions of sign 1 are called even partitions while partitions of sign -1 are called odd.

Wraps GAP's SignPartition.

EXAMPLES:

```
sage: Partition([5,3]).sign()
1
sage: Partition([5,2]).sign()
-1
```

Zolotarev's lemma states that the Legendre symbol $\left(\frac{a}{p}\right)$ for an integer $a \pmod p$ (p a prime number), can be computed as sign(p_a), where sign denotes the sign of a permutation and p_a the permutation of the residue classes $\pmod p$ induced by modular multiplication by a, provided p does not divide a.

We verify this in some examples.

```
sage: F = GF(11)
sage: a = F.multiplicative_generator();a
2
sage: plist = [int(a*F(x)) for x in range(1,11)]; plist
[2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
```

This corresponds ot the permutation (1, 2, 4, 8, 5, 10, 9, 7, 3, 6) (acting the set $\{1, 2, ..., 10\}$) and to the partition [10].

```
sage: p = PermutationGroupElement('(1, 2, 4, 8, 5, 10, 9, 7, 3, 6)')
sage: p.sign()
-1
sage: Partition([10]).sign()
-1
sage: kronecker_symbol(11,2)
-1
```

Now replace 2 by 3:

```
sage: plist = [int(F(3*x)) for x in range(1,11)]; plist
[3, 6, 9, 1, 4, 7, 10, 2, 5, 8]
sage: range(1,11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: p = PermutationGroupElement('(3,4,8,7,9)')
sage: p.sign()
1
sage: kronecker_symbol(3,11)
1
sage: Partition([5,1,1,1,1,1]).sign()
1
```

In both cases, Zolotarev holds.

REFERENCES:

- http://en.wikipedia.org/wiki/Zolotarev's_lemma

size()

Returns the size of partition p.

EXAMPLES:

```
sage: Partition([2,2]).size()
4
sage: Partition([3,2,1]).size()
6
```

to_exp(*k=0*)

Return a list of the multiplicities of the parts of a partition. Use the optional parameter k to get a return list of length at least k.

EXAMPLES:

```
sage: Partition([3,2,2,1]).to_exp()
[1, 2, 1]
sage: Partition([3,2,2,1]).to_exp(5)
[1, 2, 1, 0, 0]
```

to_exp_dict()

Returns a dictionary containing the multiplicities of the parts of this partition.

EXAMPLES:

```
sage: p = Partition([4,2,2,1])
sage: d = p.to_exp_dict()
sage: d[4]
1
sage: d[2]
2
sage: d[1]
1
sage: 5 in d
False
```

to_list()

Return self as a list.

EXAMPLES:

```
sage: p = Partition([2,1]).to_list(); p
[2, 1]
sage: type(p)
<type 'list'>
```

TESTS:

```
sage: p = Partition([2,1])
sage: pl = p.to_list()
sage: pl[0] = 0; p
[2, 1]
```

up()

Returns a generator for partitions that can be obtained from pi by adding a box.

EXAMPLES:

```
sage: [p for p in Partition([2,1,1]).up()]
[[3, 1, 1], [2, 2, 1], [2, 1, 1, 1]]
sage: [p for p in Partition([3,2]).up()]
[[4, 2], [3, 3], [3, 2, 1]]
```

up_list()

Returns a list of the partitions that can be formed from the partition p by adding a box.

EXAMPLES:

```
sage: Partition([2,1,1]).up_list()
[[3, 1, 1], [2, 2, 1], [2, 1, 1, 1]]
sage: Partition([3,2]).up_list()
[[4, 2], [3, 3], [3, 2, 1]]
```

upper_hook(*i, j, alpha*)

Returns the upper hook length of the box (i,j) in self. When alpha == 1, this is just the normal hook length. As usual, indices are 0 based.

The upper hook length of a box (i,j) in a partition κ is defined by

$$h_*^\kappa(i,j) = \kappa'_j - i + \alpha(\kappa_i - j + 1).$$

EXAMPLES:

```
sage: p = Partition([2,1])
sage: p.upper_hook(0,0,1)
3
sage: p.hook(0,0)
3
sage: [ p.upper_hook(i,j,x) for i,j in p.boxes() ]
[2*x + 1, x, x]
```

upper_hook_lengths(*alpha*)

Returns the upper hook lengths of the partition. When alpha == 1, these are just the normal hook lengths. The upper hook length of a box (i,j) in a partition κ is defined by

$$h_*^\kappa(i,j) = \kappa'_j - i + 1 + \alpha(\kappa_i - j).$$

EXAMPLES:

```
sage: Partition([3,2,1]).upper_hook_lengths(x)
[[3*x + 2, 2*x + 1, x], [2*x + 1, x], [x]]
sage: Partition([3,2,1]).upper_hook_lengths(1)
[[5, 3, 1], [3, 1], [1]]
sage: Partition([3,2,1]).hook_lengths()
[[5, 3, 1], [3, 1], [1]]
```

weighted_size()

Returns sum([i*p[i] for i in range(len(p))]). It is also the sum of the leg of every cell in b, or the sum of binomial(s[i],2) for s the conjugate partition of p.

EXAMPLES:

```
sage: Partition([2,2]).weighted_size()
2
sage: Partition([3,3,3]).weighted_size()
9
sage: Partition([5,2]).weighted_size()
2
```

Partitions(*n=None, **kwargs*)

Partitions(n, **kwargs) returns the combinatorial class of integer partitions of n, subject to the constraints given by the keywords.

Valid keywords are: `starting`, `ending`, `min_part`, `max_part`, `max_length`, `min_length`, `length`, `max_slope`, `min_slope`, `inner`, `outer`, and `parts_in`. They have the following meanings:

- `starting=p` specifies that the partitions should all be greater than or equal to p in reverse lex order.
- `length=k` specifies that the partitions have exactly k parts.
- `min_length=k` specifies that the partitions have at least k parts.
- `min_part=k` specifies that all parts of the partitions are at least k.
- `outer=p` specifies that the partitions be contained inside the partition p.
- `min_slope=k` specifies that the partitions have slope at least k; the slope is the difference between successive parts.
- `parts_in=S` specifies that the partitions have parts in the set S, which can be any sequence of positive integers.

The `max_*` versions, along with `inner` and `ending`, work analogously.

Right now, the `parts_in`, `starting`, and `ending` keyword arguments are mutually exclusive, both of each other and of other keyword arguments. If you specify, say, `parts_in`, all other keyword arguments will be ignored; `starting` and `ending` work the same way.

EXAMPLES: If no arguments are passed, then the combinatorial class of all integer partitions is returned.

```
sage: Partitions()
Partitions
sage: [2,1] in Partitions()
True
```

If an integer n is passed, then the combinatorial class of integer partitions of n is returned.

```
sage: Partitions(3)
Partitions of the integer 3
sage: Partitions(3).list()
[[3], [2, 1], [1, 1, 1]]
```

If `starting=p` is passed, then the combinatorial class of partitions greater than or equal to p in lexicographic order is returned.

```
sage: Partitions(3, starting=[2,1])
Partitions of the integer 3 starting with [2, 1]
sage: Partitions(3, starting=[2,1]).list()
[[2, 1], [1, 1, 1]]
```

If `ending=p` is passed, then the combinatorial class of partitions at most p in lexicographic order is returned.

```
sage: Partitions(3, ending=[2,1])
Partitions of the integer 3 ending with [2, 1]
sage: Partitions(3, ending=[2,1]).list()
[[3], [2, 1]]
```

Using `max_slope=-1` yields partitions into distinct parts – each part differs from the next by at least 1. Use a different `max_slope` to get parts that differ by, say, 2.

```
sage: Partitions(7, max_slope=-1).list()
[[7], [6, 1], [5, 2], [4, 3], [4, 2, 1]]
sage: Partitions(15, max_slope=-1).cardinality()
27
```

The number of partitions of n into odd parts equals the number of partitions into distinct parts. Let's test that for n from 10 to 20.

```
sage: test = lambda n: Partitions(n, max_slope=-1).cardinality() == Partitions(n, parts_in=[1,3.
sage: all(test(n) for n in [10..20])
True
```

The number of partitions of n into distinct parts that differ by at least 2 equals the number of partitions into parts that equal 1 or 4 modulo 5; this is one of the Rogers-Ramanujan identities.

```
sage: test = lambda n: Partitions(n, max_slope=-2).cardinality() == Partitions(n, parts_in=([1,6
sage: all(test(n) for n in [10..20])
True
```

Here are some more examples illustrating `min_part`, `max_part`, and `length`.

```
sage: Partitions(5,min_part=2)
Partitions of the integer 5 satisfying constraints min_part=2
sage: Partitions(5,min_part=2).list()
[[5], [3, 2]]
```

```
sage: Partitions(3,max_length=2).list()
[[3], [2, 1]]


sage: Partitions(10, min_part=2, length=3).list()
[[6, 2, 2], [5, 3, 2], [4, 4, 2], [4, 3, 3]]
```

Here are some further examples using various constraints:

```
  sage: [x for x in Partitions(4)]
  [[4], [3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
  sage: [x for x in Partitions(4, length=2)]
  [[3, 1], [2, 2]]
  sage: [x for x in Partitions(4, min_length=2)]
  [[3, 1], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
  sage: [x for x in Partitions(4, max_length=2)]
  [[4], [3, 1], [2, 2]]
  sage: [x for x in Partitions(4, min_length=2, max_length=2)]
  [[3, 1], [2, 2]]
  sage: [x for x in Partitions(4, max_part=2)]
  [[2, 2], [2, 1, 1], [1, 1, 1, 1]]
  sage: [x for x in Partitions(4, min_part=2)]
  [[4], [2, 2]]
  sage: [x for x in Partitions(4, outer=[3,1,1])]
  [[3, 1], [2, 1, 1]]
  sage: [x for x in Partitions(4, outer=[infinity, 1, 1])]
  [[4], [3, 1], [2, 1, 1]]
  sage: [x for x in Partitions(4, inner=[1,1,1])]
  [[2, 1, 1], [1, 1, 1, 1]]
  sage: [x for x in Partitions(4, max_slope=-1)]
  [[4], [3, 1]]
  sage: [x for x in Partitions(4, min_slope=-1)]
  [[4], [2, 2], [2, 1, 1], [1, 1, 1, 1]]
  sage: [x for x in Partitions(11, max_slope=-1, min_slope=-3, min_length=2, max_length=4)]
  [[7, 4], [6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2], [5, 3, 2, 1]]
  sage: [x for x in Partitions(11, max_slope=-1, min_slope=-3, min_length=2, max_length=4, outer
  [[6, 5], [6, 4, 1], [6, 3, 2], [5, 4, 2]]
```

```
Note that if you specify min_part=0, then the objects produced
will have parts equal to zero which violates some internal
assumptions that the Partition class makes.

::
```

```
  sage: [x for x in Partitions(4, length=3, min_part=0)]
  doctest:... RuntimeWarning: Currently, setting min_part=0 produces Partition objects which vic
  [[4, 0, 0], [3, 1, 0], [2, 2, 0], [2, 1, 1]]
  sage: [x for x in Partitions(4, min_length=3, min_part=0)]
  [[4, 0, 0], [3, 1, 0], [2, 2, 0], [2, 1, 1], [1, 1, 1, 1]]
```

```
Except for very special cases, counting is done by brute force
iteration through all the partitions. However the iteration
itself has a reasonnable complexity (constant memory, constant
amortized time), which allow for manipulating large partitions::
```

```
  sage: Partitions(1000, max_length=1).list()
  [[1000]]
```

In particular, getting the first element is also constant time::

```
sage: Partitions(30, max_part=29).first()
[29, 1]
```

TESTS:

```
sage: P = Partitions(5, min_part=2)
sage: P == loads(dumps(P))
True

sage: repr( Partitions(5, min_part=2) )
'Partitions of the integer 5 satisfying constraints min_part=2'

sage: P = Partitions(5, min_part=2)
sage: P.first().parent()
Partitions...
sage: [2,1] in P
False
sage: [2,2,1] in P
False
sage: [3,2] in P
True
```

class PartitionsGreatestEQ(*n, k*)

Returns combinatorial class of all (unordered) "restricted" partitions of the integer n having its greatest part equal to the integer k.

EXAMPLES:

```
sage: PartitionsGreatestEQ(10,2)
Partitions of 10 having greatest part equal to 2
sage: PartitionsGreatestEQ(10,2).list()
[[2, 2, 2, 2, 2],
 [2, 2, 2, 2, 1, 1],
 [2, 2, 2, 1, 1, 1, 1],
 [2, 2, 1, 1, 1, 1, 1, 1],
 [2, 1, 1, 1, 1, 1, 1, 1, 1]]

sage: [4,3,2,1] in PartitionsGreatestEQ(10,2)
False
sage: [2,2,2,2,2] in PartitionsGreatestEQ(10,2)
True
sage: [1]*10 in PartitionsGreatestEQ(10,2)
False

sage: PartitionsGreatestEQ(10,2).first().parent()
Partitions...
```

TESTS:

```
sage: p = PartitionsGreatestEQ(10,2)
sage: p == loads(dumps(p))
True
```

class PartitionsGreatestEQ_nk(*n, k*)

class PartitionsGreatestLE(*n, k*)

Returns the combinatorial class of all (unordered) "restricted" partitions of the integer n having parts less than or equal to the integer k.

EXAMPLES:

```
sage: PartitionsGreatestLE(10,2)
Partitions of 10 having parts less than or equal to 2
sage: PartitionsGreatestLE(10,2).list()
[[2, 2, 2, 2, 2],
 [2, 2, 2, 2, 1, 1],
 [2, 2, 2, 1, 1, 1, 1],
 [2, 2, 1, 1, 1, 1, 1, 1],
 [2, 1, 1, 1, 1, 1, 1, 1, 1],
 [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]

sage: [4,3,2,1] in PartitionsGreatestLE(10,2)
False
sage: [2,2,2,2,2] in PartitionsGreatestLE(10,2)
True
sage: PartitionsGreatestLE(10,2).first().parent()
Partitions...
```

TESTS:

```
sage: p = PartitionsGreatestLE(10,2)
sage: p == loads(dumps(p))
True
```

class PartitionsGreatestLE_nk(*n, k*)

PartitionsInBox(*h, w*)

Returns the combinatorial class of partitions that fit in a h by w box.

EXAMPLES:

```
sage: PartitionsInBox(2,2)
Integer partitions which fit in a 2 x 2 box
sage: PartitionsInBox(2,2).list()
[[], [1], [1, 1], [2], [2, 1], [2, 2]]
```

class PartitionsInBox_hw(*h, w*)

list()

Returns a list of all the partitions inside a box of height h and width w.

EXAMPLES:

```
sage: PartitionsInBox(2,2).list()
[[], [1], [1, 1], [2], [2, 1], [2, 2]]
```

class Partitions_all()

cardinality()

Returns the number of integer partitions.

EXAMPLES:

```
sage: Partitions().cardinality()
+Infinity
```

object_class()

class Partitions_constraints(*n, length=None, min_length=0, max_length=+Infinity, floor=None, ceiling=None, min_part=0, max_part=+Infinity, min_slope=-Infinity, max_slope=+Infinity, name=None, element_constructor=None*)

class Partitions_ending(*n, ending_partition*)

first()

EXAMPLES:

```
sage: Partitions(4, ending=[1,1,1,1]).first()
[4]
```

next(*part*)

EXAMPLES:

```
sage: Partitions(4, ending=[1,1,1,1]).next(Partition([4]))
[3, 1]
sage: Partitions(4, ending=[1,1,1,1]).next(Partition([1,1,1,1])) is None
True
```

class Partitions_n(*n*)

cardinality(*algorithm='default'*)

INPUT:

- `algorithm`
  - (default: `default`)
  - `'bober'` - use Jonathan Bober's implementation (*very* fast, but relatively new)
  - `'gap'` - use GAP (VERY *slow*)
  - `'pari'` - use PARI. Speed seems the same as GAP until n is in the thousands, in which case PARI is faster. *But* PARI has a bug, e.g., on 64-bit Linux PARI-2.3.2 outputs numbpart(147007)%1000 as 536, but it should be 533!. So do not use this option.
  - `'default'` - `'bober'` when k is not specified; otherwise use `'gap'`.

Use the function `partitions(n)` to return a generator over all partitions of n.

It is possible to associate with every partition of the integer n a conjugacy class of permutations in the symmetric group on n points and vice versa. Therefore p(n) = NrPartitions(n) is the number of conjugacy classes of the symmetric group on n points.

EXAMPLES:

```
sage: v = Partitions(5).list(); v
[[5], [4, 1], [3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
sage: len(v)
7
sage: Partitions(5).cardinality(algorithm='gap')
7
sage: Partitions(5).cardinality(algorithm='pari')
7
sage: Partitions(5).cardinality(algorithm='bober')
7
```

The input must be a nonnegative integer or a ValueError is raised.

```
sage: Partitions(10).cardinality()
42
sage: Partitions(3).cardinality()
3
```

```
sage: Partitions(10).cardinality()
42
sage: Partitions(3).cardinality(algorithm='pari')
3
sage: Partitions(10).cardinality(algorithm='pari')
42
sage: Partitions(40).cardinality()
37338
sage: Partitions(100).cardinality()
190569292
```

A generating function for p(n) is given by the reciprocal of Euler's function:

$$\sum_{n=0}^{\infty} p(n)x^n = \prod_{k=1}^{\infty}\left(\frac{1}{1-x^k}\right).$$

We use Sage to verify that the first several coefficients do instead agree:

```
sage: q = PowerSeriesRing(QQ, 'q', default_prec=9).gen()
sage: prod([(1-q^k)^(-1) for k in range(1,9)])  ## partial product of
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + O(q^9)
sage: [Partitions(k) .cardinality()for k in range(2,10)]
[2, 3, 5, 7, 11, 15, 22, 30]
```

REFERENCES:

- http://en.wikipedia.org/wiki/Partition_(number_theory)

first()

Returns the lexicographically first partition of a positive integer n. This is the partition [n].

EXAMPLES:

```
sage: Partitions(4).first()
[4]
```

last()

Returns the lexicographically last partition of the positive integer n. This is the all-ones partition.

EXAMPLES:

```
sage: Partitions(4).last()
[1, 1, 1, 1]
```

object_class()

class **Partitions_parts_in**(*n, parts*)

cardinality()

Return the number of partitions with parts in S. Wraps GAP's `NrRestrictedPartitions`.

EXAMPLES:

```
sage: Partitions(15, parts_in=[2,3,7]).cardinality()
5
```

If you can use all parts 1 through n, we'd better get $p(n)$:

```
sage: Partitions(20, parts_in=[1..20]).cardinality() == Partitions(20).cardinality()
True
```

TESTS:

Let's check the consistency of GAP's function and our own algorithm that actually generates the partitions.

```
sage: ps = Partitions(15, parts_in=[1,2,3])
sage: ps.cardinality() == len(ps.list())
True
sage: ps = Partitions(15, parts_in=[])
sage: ps.cardinality() == len(ps.list())
True
sage: ps = Partitions(3000, parts_in=[50,100,500,1000])
sage: ps.cardinality() == len(ps.list())
True
sage: ps = Partitions(10, parts_in=[3,6,9])
sage: ps.cardinality() == len(ps.list())
True
sage: ps = Partitions(0, parts_in=[1,2])
sage: ps.cardinality() == len(ps.list())
True
```

first()

Return the lexicographically first partition of a positive integer n with the specified parts, or None if no such partition exists.

EXAMPLES:

```
sage: Partitions(9, parts_in=[3,4]).first()
[3, 3, 3]
sage: Partitions(6, parts_in=[1..6]).first()
[6]
sage: Partitions(30, parts_in=[4,7,8,10,11]).first()
[11, 11, 8]
```

last()

Returns the lexicographically last partition of the positive integer n with the specified parts, or None if no such partition exists.

EXAMPLES:

```
sage: Partitions(15, parts_in=[2,3]).last()
[3, 2, 2, 2, 2, 2, 2]
sage: Partitions(30, parts_in=[4,7,8,10,11]).last()
[7, 7, 4, 4, 4, 4]
sage: Partitions(10, parts_in=[3,6]).last() is None
True
sage: Partitions(50, parts_in=[11,12,13]).last()
[13, 13, 12, 12]
sage: Partitions(30, parts_in=[4,7,8,10,11]).last()
[7, 7, 4, 4, 4, 4]
```

TESTS:

```
sage: Partitions(6, parts_in=[1..6]).last()
[1, 1, 1, 1, 1, 1]
sage: Partitions(0, parts_in=[]).last()
[]
sage: Partitions(50, parts_in=[11,12]).last() is None
True
```

object_class()

class **Partitions_starting**(*n, starting_partition*)

first()

EXAMPLES:

```
sage: Partitions(3, starting=[2,1]).first()
[2, 1]
```

next(*part*)

EXAMPLES:

```
sage: Partitions(3, starting=[2,1]).next(Partition([2,1]))
[1, 1, 1]
```

RestrictedPartitions(*n, S, k=None*)

This function has been deprecated and will be removed in a future version of Sage; use Partitions() with the `parts_in` keyword.

Original docstring follows.

A restricted partition is, like an ordinary partition, an unordered sum $n = p_1 + p_2 + \ldots + p_k$ of positive integers and is represented by the list $p = [p_1, p_2, \ldots, p_k]$, in nonincreasing order. The difference is that here the p_i must be elements from the set S, while for ordinary partitions they may be elements from $[1..n]$.

Returns the list of all restricted partitions of the positive integer n into sums with k summands with the summands of the partition coming from the set S. If k is not given all restricted partitions for all k are returned.

Wraps GAP's RestrictedPartitions.

EXAMPLES:

```
sage: RestrictedPartitions(5,[3,2,1])
doctest:...: DeprecationWarning: RestrictedPartitions is deprecated; use Partitions with the par
doctest:...: DeprecationWarning: RestrictedPartitions_nsk is deprecated; use Partitions with the
Partitions of 5 restricted to the values [1, 2, 3]
sage: RestrictedPartitions(5,[3,2,1]).list()
[[3, 2], [3, 1, 1], [2, 2, 1], [2, 1, 1, 1], [1, 1, 1, 1, 1]]
sage: RestrictedPartitions(5,[3,2,1],4)
Partitions of 5 restricted to the values [1, 2, 3] of length 4
sage: RestrictedPartitions(5,[3,2,1],4).list()
[[2, 1, 1, 1]]
```

class **RestrictedPartitions_nsk**(*n, S, k=None*)

We are deprecating RestrictedPartitions, so this class should be deprecated too.

cardinality()

Returns the size of RestrictedPartitions(n,S,k). Wraps GAP's NrRestrictedPartitions.

EXAMPLES:

```
sage: RestrictedPartitions(8,[1,3,5,7]).cardinality()
doctest:...: DeprecationWarning: RestrictedPartitions is deprecated; use Partitions with the
6
sage: RestrictedPartitions(8,[1,3,5,7],2).cardinality()
2
```

list()

Returns the list of all restricted partitions of the positive integer n into sums with k summands with the summands of the partition coming from the set S. If k is not given all restricted partitions for all k are returned.

Wraps GAP's RestrictedPartitions.

EXAMPLES:

```
sage: RestrictedPartitions(8,[1,3,5,7]).list()
doctest:...: DeprecationWarning: RestrictedPartitions is deprecated; use Partitions with the
[[7, 1], [5, 3], [5, 1, 1, 1], [3, 3, 1, 1], [3, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1]]
```

```
sage: RestrictedPartitions(8,[1,3,5,7],2).list()
[[7, 1], [5, 3]]
```

cyclic_permutations_of_partition(*partition*)

Returns all combinations of cyclic permutations of each cell of the partition.

AUTHORS:

- Robert L. Miller

EXAMPLES:

```
sage: from sage.combinat.partition import cyclic_permutations_of_partition
sage: cyclic_permutations_of_partition([[1,2,3,4],[5,6,7]])
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

Note that repeated elements are not considered equal:

```
sage: cyclic_permutations_of_partition([[1,2,3],[4,4,4]])
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

cyclic_permutations_of_partition_iterator(*partition*)

Iterates over all combinations of cyclic permutations of each cell of the partition.

AUTHORS:

- Robert L. Miller

EXAMPLES:

```
sage: from sage.combinat.partition import cyclic_permutations_of_partition
sage: list(cyclic_permutations_of_partition_iterator([[1,2,3,4],[5,6,7]]))
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

Note that repeated elements are not considered equal:

```
sage: list(cyclic_permutations_of_partition_iterator([[1,2,3],[4,4,4]]))
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

ferrers_diagram(*pi*)

Return the Ferrers diagram of pi.

INPUT:

- `pi` - a partition, given as a list of integers.

EXAMPLES:

```
sage: print ferrers_diagram([5,5,2,1])
*****
*****
**
*
sage: pi = partitions_list(10)[30] ## [6,1,1,1,1]
sage: print ferrers_diagram(pi)
******
*
*
*
*
sage: pi = partitions_list(10)[33] ## [6, 3, 1]
sage: print ferrers_diagram(pi)
******
***
*
```

from_core_and_quotient(*core, quotient*)

Returns a partition from its r-core and r-quotient.

Algorithm from mupad-combinat.

EXAMPLES:

```
sage: partition.from_core_and_quotient([2,1], [[2,1],[3],[1,1,1]])
[11, 5, 5, 3, 2, 2, 2]
```

from_exp(*a*)

Returns a partition from its list of multiplicities.

EXAMPLES:

```
sage: partition.from_exp([1,2,1])
[3, 2, 2, 1]
```

number_of_ordered_partitions(*n, k=None*)

Returns the size of ordered_partitions(n,k). Wraps GAP's NrOrderedPartitions.

It is possible to associate with every partition of the integer n a conjugacy class of permutations in the symmetric group on n points and vice versa. Therefore p(n) = NrPartitions(n) is the number of conjugacy classes of the symmetric group on n points.

EXAMPLES:

```
sage: number_of_ordered_partitions(10,2)
9
sage: number_of_ordered_partitions(15)
16384
```

number_of_partitions(*n, k=None, algorithm='default'*)

Returns the size of partitions_list(n,k).

INPUT:

- `n` - an integer
- `k` - (default: None); if specified, instead returns the cardinality of the set of all (unordered) partitions of the positive integer n into sums with k summands.
- `algorithm` - (default: 'default')
- `'default'` - If k is not None, then use Gap (very slow). If k is None, use Jonathan Bober's highly optimized implementation (this is the fastest code in the world for this problem).
- `'bober'` - use Jonathan Bober's implementation
- `'gap'` - use GAP (VERY *slow*)
- `'pari'` - use PARI. Speed seems the same as GAP until n is in the thousands, in which case PARI is faster. *But* PARI has a bug, e.g., on 64-bit Linux PARI-2.3.2 outputs numbpart(147007)%1000 as 536 when it should be 533!. So do not use this option.

IMPLEMENTATION: Wraps GAP's NrPartitions or PARI's numbpart function.

Use the function `partitions(n)` to return a generator over all partitions of n.

It is possible to associate with every partition of the integer n a conjugacy class of permutations in the symmetric group on n points and vice versa. Therefore p(n) = NrPartitions(n) is the number of conjugacy classes of the symmetric group on n points.

EXAMPLES:

```
sage: v = list(partitions(5)); v
[(1, 1, 1, 1, 1), (1, 1, 1, 2), (1, 2, 2), (1, 1, 3), (2, 3), (1, 4), (5,)]
sage: len(v)
7
sage: number_of_partitions(5, algorithm='gap')
7
sage: number_of_partitions(5, algorithm='pari')
7
sage: number_of_partitions(5, algorithm='bober')
7
```

The input must be a nonnegative integer or a ValueError is raised.

```
sage: number_of_partitions(-5)
...
ValueError: n (=-5) must be a nonnegative integer
```

```
sage: number_of_partitions(10,2)
5
sage: number_of_partitions(10)
42
sage: number_of_partitions(3)
3
sage: number_of_partitions(10)
42
```

```
sage: number_of_partitions(3, algorithm='pari')
3
sage: number_of_partitions(10, algorithm='pari')
42
sage: number_of_partitions(40)
37338
sage: number_of_partitions(100)
190569292
sage: number_of_partitions(100000)
274935105697756965126775163209863526881734293159800547582031259843021473281149641730550507416607
```

A generating function for p(n) is given by the reciprocal of Euler's function:

$$\sum_{n=0}^{\infty} p(n)x^n = \prod_{k=1}^{\infty} \left(\frac{1}{1-x^k}\right).$$

We use Sage to verify that the first several coefficients do instead agree:

```
sage: q = PowerSeriesRing(QQ, 'q', default_prec=9).gen()
sage: prod([(1-q^k)^(-1) for k in range(1,9)])  ## partial product of
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + O(q^9)
sage: [number_of_partitions(k) for k in range(2,10)]
[2, 3, 5, 7, 11, 15, 22, 30]
```

REFERENCES:

- http://en.wikipedia.org/wiki/Partition_(number_theory)

TESTS:

```
sage: n = 500 + randint(0,500)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1500 + randint(0,1500)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000 + randint(0,1000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 100000000 + randint(0,100000000)
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0
True
sage: n = 1000000000 + randint(0,1000000000)
```

```
sage: number_of_partitions( n - (n % 385) + 369) % 385 == 0       # takes a long time
True
```

Another consistency test for n up to 500:

```
sage: len([n for n in [1..500] if number_of_partitions(n) != number_of_partitions(n,algorithm='p
0
```

number_of_partitions_list(*n, k=None*)

This function will be deprecated in a future version of Sage and eventually removed. Use Partitions(n).cardinality() or Partitions(n, length=k).cardinality() instead.

Original docstring follows.

Returns the size of partitions_list(n,k).

Wraps GAP's NrPartitions.

It is possible to associate with every partition of the integer n a conjugacy class of permutations in the symmetric group on n points and vice versa. Therefore p(n) = NrPartitions(n) is the number of conjugacy classes of the symmetric group on n points.

`number_of_partitions(n)` is also available in PARI, however the speed seems the same until n is in the thousands (in which case PARI is faster).

EXAMPLES:

```
sage: partition.number_of_partitions_list(10,2)
5
sage: partition.number_of_partitions_list(10)
42
```

A generating function for p(n) is given by the reciprocal of Euler's function:

$$\sum_{n=0}^{\infty} p(n)x^n = \prod_{k=1}^{\infty}\left(\frac{1}{1-x^k}\right).$$

Sage verifies that the first several coefficients do instead agree:

```
sage: q = PowerSeriesRing(QQ, 'q', default_prec=9).gen()
sage: prod([(1-q^k)^(-1) for k in range(1,9)])  ## partial product of
1 + q + 2*q^2 + 3*q^3 + 5*q^4 + 7*q^5 + 11*q^6 + 15*q^7 + 22*q^8 + O(q^9)
sage: [partition.number_of_partitions_list(k) for k in range(2,10)]
[2, 3, 5, 7, 11, 15, 22, 30]
```

REFERENCES:

- http://en.wikipedia.org/wiki/Partition_(number_theory)

number_of_partitions_restricted(*n, S, k=None*)

This function will be deprecated in a future version of Sage and eventually removed. Use RestrictedPartitions(n, S, k).cardinality() instead.

Original docstring follows.

Returns the size of partitions_restricted(n,S,k). Wraps GAP's NrRestrictedPartitions.

EXAMPLES:

```
sage: number_of_partitions_restricted(8,[1,3,5,7])
6
sage: number_of_partitions_restricted(8,[1,3,5,7],2)
2
```

number_of_partitions_set(*S, k*)

Returns the size of `partitions_set(S,k)`. Wraps GAP's NrPartitionsSet.

The Stirling number of the second kind is the number of partitions of a set of size n into k blocks.

EXAMPLES:

```
sage: mset = [1,2,3,4]
sage: number_of_partitions_set(mset,2)
7
sage: stirling_number2(4,2)
7
```

REFERENCES

- http://en.wikipedia.org/wiki/Partition_of_a_set

number_of_partitions_tuples(*n, k*)

number_of_partition_tuples(n, k) returns the number of partition_tuples(n,k).

Wraps GAP's NrPartitionTuples.

EXAMPLES:

```
sage: number_of_partitions_tuples(3,2)
10
sage: number_of_partitions_tuples(8,2)
185
```

Now we compare that with the result of the following GAP computation:

```
gap> S8:=Group((1,2,3,4,5,6,7,8),(1,2));
Group([ (1,2,3,4,5,6,7,8), (1,2) ])
gap> C2:=Group((1,2));
Group([ (1,2) ])
gap> W:=WreathProduct(C2,S8);
<permutation group of size 10321920 with 10 generators>
gap> Size(W);
10321920     ## = 2^8*Factorial(8), which is good:-)
gap> Size(ConjugacyClasses(W));
185
```

ordered_partitions(*n, k=None*)

An ordered partition of n is an ordered sum

$$n = p_1 + p_2 + \cdots + p_k$$

of positive integers and is represented by the list $p = [p_1, p_2, \cdots, p_k]$. If k is omitted then all ordered partitions are returned.

`ordered_partitions(n,k)` returns the list of all (ordered) partitions of the positive integer n into sums with k summands.

Do not call `ordered_partitions` with an n much larger than 15, since the list will simply become too large.

Wraps GAP's OrderedPartitions.

The number of ordered partitions T_n of $\{1, 2, ..., n\}$ has the generating function is

$$\sum_n \frac{T_n}{n!} x^n = \frac{1}{2 - e^x}.$$

EXAMPLES:

```
sage: ordered_partitions(10,2)
[[1, 9], [2, 8], [3, 7], [4, 6], [5, 5], [6, 4], [7, 3], [8, 2], [9, 1]]

sage: ordered_partitions(4)
[[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [1, 3], [2, 1, 1], [2, 2], [3, 1], [4]]
```

REFERENCES:

- http://en.wikipedia.org/wiki/Ordered_partition_of_a_set

partition_associated(*pi*)

partition_associated(pi) returns the "associated" (also called "conjugate" in the literature) partition of the partition pi which is obtained by transposing the corresponding Ferrers diagram.

EXAMPLES:

```
sage: partition_associated([2,2])
[2, 2]
sage: partition_associated([6,3,1])
[3, 2, 2, 1, 1, 1]
sage: print ferrers_diagram([6,3,1])
******
***
*
sage: print ferrers_diagram([3,2,2,1,1,1])
***
**
**
*
*
*
```

partition_power(*pi, k*)

partition_power(pi, k) returns the partition corresponding to the k-th power of a permutation with cycle structure pi (thus describes the powermap of symmetric groups).

Wraps GAP's PowerPartition.

EXAMPLES:

```
sage: partition_power([5,3],1)
[5, 3]
sage: partition_power([5,3],2)
[5, 3]
sage: partition_power([5,3],3)
[5, 1, 1, 1]
sage: partition_power([5,3],4)
[5, 3]
```

Now let us compare this to the power map on S_8:

```
sage: G = SymmetricGroup(8)
sage: g = G([(1,2,3,4,5),(6,7,8)])
sage: g
(1,2,3,4,5)(6,7,8)
sage: g^2
(1,3,5,2,4)(6,8,7)
sage: g^3
(1,4,2,5,3)
```

```
sage: g^4
(1,5,4,3,2)(6,7,8)
```

partition_sign(*pi*)

partition_sign(pi) returns the sign of a permutation with cycle structure given by the partition pi.

This function corresponds to a homomorphism from the symmetric group S_n into the cyclic group of order 2, whose kernel is exactly the alternating group A_n. Partitions of sign 1 are called even partitions while partitions of sign -1 are called odd.

Wraps GAP's SignPartition.

EXAMPLES:

```
sage: partition_sign([5,3])
1
sage: partition_sign([5,2])
-1
```

Zolotarev's lemma states that the Legendre symbol $\left(\frac{a}{p}\right)$ for an integer $a \pmod p$ (p a prime number), can be computed as sign(p_a), where sign denotes the sign of a permutation and p_a the permutation of the residue classes $\pmod p$ induced by modular multiplication by a, provided p does not divide a.

We verify this in some examples.

```
sage: F = GF(11)
sage: a = F.multiplicative_generator();a
2
sage: plist = [int(a*F(x)) for x in range(1,11)]; plist
[2, 4, 6, 8, 10, 1, 3, 5, 7, 9]
```

This corresponds ot the permutation (1, 2, 4, 8, 5, 10, 9, 7, 3, 6) (acting the set $\{1, 2, ..., 10\}$) and to the partition [10].

```
sage: p = PermutationGroupElement('(1, 2, 4, 8, 5, 10, 9, 7, 3, 6)')
sage: p.sign()
-1
sage: partition_sign([10])
-1
sage: kronecker_symbol(11,2)
-1
```

Now replace 2 by 3:

```
sage: plist = [int(F(3*x)) for x in range(1,11)]; plist
[3, 6, 9, 1, 4, 7, 10, 2, 5, 8]
sage: range(1,11)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
sage: p = PermutationGroupElement('(3,4,8,7,9)')
sage: p.sign()
1
sage: kronecker_symbol(3,11)
1
sage: partition_sign([5,1,1,1,1,1])
1
```

In both cases, Zolotarev holds.

REFERENCES:

- http://en.wikipedia.org/wiki/Zolotarev's_lemma

partitions(*n*)
Generator of all the partitions of the integer n.

INPUT:

- n - int

To compute the number of partitions of n use number_of_partitions(n).

EXAMPLES:

```
sage: partitions(3)
<generator object at 0x...>
sage: list(partitions(3))
[(1, 1, 1), (1, 2), (3,)]
```

AUTHORS:

- Adapted from David Eppstein, Jan Van lent, George Yoshida; Python Cookbook 2, Recipe 19.16.

partitions_greatest(*n, k*)
Returns the list of all (unordered) "restricted" partitions of the integer n having parts less than or equal to the integer k.

Wraps GAP's PartitionsGreatestLE.

EXAMPLES:

```
sage: partitions_greatest(10,2)
[[1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
 [2, 1, 1, 1, 1, 1, 1, 1, 1],
 [2, 2, 1, 1, 1, 1, 1, 1],
 [2, 2, 2, 1, 1, 1, 1],
 [2, 2, 2, 2, 1, 1],
 [2, 2, 2, 2, 2]]
```

partitions_greatest_eq(*n, k*)
Returns the list of all (unordered) "restricted" partitions of the integer n having at least one part equal to the integer k.

Wraps GAP's PartitionsGreatestEQ.

EXAMPLES:

```
sage: partitions_greatest_eq(10,2)
[[2, 1, 1, 1, 1, 1, 1, 1, 1],
 [2, 2, 1, 1, 1, 1, 1, 1],
 [2, 2, 2, 1, 1, 1, 1],
 [2, 2, 2, 2, 1, 1],
 [2, 2, 2, 2, 2]]
```

partitions_list(*n, k=None*)
This function will be deprecated in a future version of Sage and eventually removed. Use Partitions(n).list() or Partitions(n, length=k).list() instead.

Original docstring follows.

An unordered partition of n is an unordered sum $n = p_1 + p_2 + \ldots + p_k$ of positive integers and is represented by the list $p = [p_1, p_2, \ldots, p_k]$, in nonincreasing order, i.e., $p1 \geq p_2 ... \geq p_k$.

INPUT:

- n, k - positive integer

partitions_list(n,k) returns the list of all (unordered) partitions of the positive integer n into sums with k summands. If k is omitted then all partitions are returned.

Do not call partitions_list with an n much larger than 40, in which case there are 37338 partitions, since the list will simply become too large.

Wraps GAP's Partitions.

EXAMPLES:

```
sage: partitions_list(10,2)
[[5, 5], [6, 4], [7, 3], [8, 2], [9, 1]]
sage: partitions_list(5)
[[1, 1, 1, 1, 1], [2, 1, 1, 1], [2, 2, 1], [3, 1, 1], [3, 2], [4, 1], [5]]
```

partitions_restricted(*n, S, k=None*)

This function will be deprecated in a future version of Sage and eventually removed. Use RestrictedPartitions(n, S, k).list() instead.

Original docstring follows.

A restricted partition is, like an ordinary partition, an unordered sum $n = p_1 + p_2 + \ldots + p_k$ of positive integers and is represented by the list $p = [p_1, p_2, \ldots, p_k]$, in nonincreasing order. The difference is that here the p_i must be elements from the set S, while for ordinary partitions they may be elements from $[1..n]$.

Returns the list of all restricted partitions of the positive integer n into sums with k summands with the summands of the partition coming from the set S. If k is not given all restricted partitions for all k are returned.

Wraps GAP's RestrictedPartitions.

EXAMPLES:

```
sage: partitions_restricted(8,[1,3,5,7])
[[1, 1, 1, 1, 1, 1, 1, 1],
 [3, 1, 1, 1, 1, 1],
 [3, 3, 1, 1],
 [5, 1, 1, 1],
 [5, 3],
 [7, 1]]
sage: partitions_restricted(8,[1,3,5,7],2)
[[5, 3], [7, 1]]
```

partitions_set(*S, k=None, use_file=True*)

An unordered partition of a set S is a set of pairwise disjoint nonempty subsets with union S and is represented by a sorted list of such subsets.

partitions_set returns the list of all unordered partitions of the list S of increasing positive integers into k pairwise disjoint nonempty sets. If k is omitted then all partitions are returned.

The Bell number B_n, named in honor of Eric Temple Bell, is the number of different partitions of a set with n elements.

Warning: Wraps GAP - hence S must be a list of objects that have string representations that can be interpreted by the GAP interpreter. If mset consists of at all complicated Sage objects, this function does *not* do what you expect. See SetPartitions in `combinat/set_partition`.

Warning: This function is inefficient. The runtime is dominated by parsing the output from GAP.

Wraps GAP's PartitionsSet.

EXAMPLES:

```
sage: S = [1,2,3,4]
sage: partitions_set(S,2)
[[[1], [2, 3, 4]],
 [[1, 2], [3, 4]],
 [[1, 2, 3], [4]],
 [[1, 2, 4], [3]],
 [[1, 3], [2, 4]],
 [[1, 3, 4], [2]],
 [[1, 4], [2, 3]]]
```

REFERENCES:

- http://en.wikipedia.org/wiki/Partition_of_a_set

partitions_tuples(*n, k*)

partition_tuples(n, k) returns the list of all k-tuples of partitions which together form a partition of n.

k-tuples of partitions describe the classes and the characters of wreath products of groups with k conjugacy classes with the symmetric group S_n.

Wraps GAP's PartitionTuples.

EXAMPLES:

```
sage: partitions_tuples(3,2)
[[[1, 1, 1], []],
 [[1, 1], [1]],
 [[1], [1, 1]],
 [[], [1, 1, 1]],
 [[2, 1], []],
 [[1], [2]],
 [[2], [1]],
 [[], [2, 1]],
 [[3], []],
 [[], [3]]]
```

17.23 Permutations

The Permutations module. Use Permutation? to get information about the Permutation class, and Permutations? to get information about the combinatorial class of permutations.

AUTHORS:

- Mike Hansen
- Dan Drake (2008-04-07): allow Permutation() to take lists of tuples
- Sebastien Labbe (2009-03-17): added robinson_schensted_inverse

Arrangements(*mset, k*)

An arrangement of mset is an ordered selection without repetitions and is represented by a list that contains only elements from mset, but maybe in a different order.

`Arrangements` returns the combinatorial class of arrangements of the multiset mset that contain k elements.

EXAMPLES:

```
sage: mset = [1,1,2,3,4,4,5]
sage: Arrangements(mset,2).list()
[[1, 1],
 [1, 2],
 [1, 3],
 [1, 4],
 [1, 5],
 [2, 1],
 [2, 3],
 [2, 4],
 [2, 5],
 [3, 1],
 [3, 2],
 [3, 4],
 [3, 5],
 [4, 1],
 [4, 2],
 [4, 3],
 [4, 4],
 [4, 5],
 [5, 1],
 [5, 2],
 [5, 3],
 [5, 4]]
 sage: Arrangements(mset,2).cardinality()
 22
 sage: Arrangements( ["c","a","t"], 2 ).list()
 [['c', 'a'], ['c', 't'], ['a', 'c'], ['a', 't'], ['t', 'c'], ['t', 'a']]
 sage: Arrangements( ["c","a","t"], 3 ).list()
 [['c', 'a', 't'],
  ['c', 't', 'a'],
  ['a', 'c', 't'],
  ['a', 't', 'c'],
  ['t', 'c', 'a'],
  ['t', 'a', 'c']]
```

class **Arrangements_msetk**(*mset, k*)

class **Arrangements_setk**(*s, k*)

CyclicPermutations(*mset*)

Returns the combinatorial class of all cyclic permutations of mset in cycle notation. These are the same as necklaces.

EXAMPLES:

```
sage: CyclicPermutations(range(4)).list()
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
sage: CyclicPermutations([1,1,1]).list()
[[1, 1, 1]]
```

CyclicPermutationsOfPartition(*partition*)

Returns the combinatorial class of all combinations of cyclic permutations of each cell of the partition. This is the same as a Cartesian product of necklaces.

EXAMPLES:

```
sage: CyclicPermutationsOfPartition([[1,2,3,4],[5,6,7]]).list()
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]

sage: CyclicPermutationsOfPartition([[1,2,3,4],[4,4,4]]).list()
[[[1, 2, 3, 4], [4, 4, 4]],
 [[1, 2, 4, 3], [4, 4, 4]],
 [[1, 3, 2, 4], [4, 4, 4]],
 [[1, 3, 4, 2], [4, 4, 4]],
 [[1, 4, 2, 3], [4, 4, 4]],
 [[1, 4, 3, 2], [4, 4, 4]]]

sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list()
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]

sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list(distinct=True)
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

class **CyclicPermutationsOfPartition_partition**(*partition*)

iterator(*distinct=False*)

AUTHORS:

- Robert Miller

EXAMPLES:

```
sage: CyclicPermutationsOfPartition([[1,2,3,4],[5,6,7]]).list() # indirect doctest
[[[1, 2, 3, 4], [5, 6, 7]],
 [[1, 2, 4, 3], [5, 6, 7]],
 [[1, 3, 2, 4], [5, 6, 7]],
 [[1, 3, 4, 2], [5, 6, 7]],
 [[1, 4, 2, 3], [5, 6, 7]],
 [[1, 4, 3, 2], [5, 6, 7]],
 [[1, 2, 3, 4], [5, 7, 6]],
 [[1, 2, 4, 3], [5, 7, 6]],
 [[1, 3, 2, 4], [5, 7, 6]],
 [[1, 3, 4, 2], [5, 7, 6]],
 [[1, 4, 2, 3], [5, 7, 6]],
 [[1, 4, 3, 2], [5, 7, 6]]]
```

```
sage: CyclicPermutationsOfPartition([[1,2,3,4],[4,4,4]]).list()
[[[1, 2, 3, 4], [4, 4, 4]],
 [[1, 2, 4, 3], [4, 4, 4]],
 [[1, 3, 2, 4], [4, 4, 4]],
 [[1, 3, 4, 2], [4, 4, 4]],
 [[1, 4, 2, 3], [4, 4, 4]],
 [[1, 4, 3, 2], [4, 4, 4]]]

sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list()
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]

sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list(distinct=True)
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

list(*distinct=False*)

EXAMPLES:

```
sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list()
[[[1, 2, 3], [4, 4, 4]], [[1, 3, 2], [4, 4, 4]]]
sage: CyclicPermutationsOfPartition([[1,2,3],[4,4,4]]).list(distinct=True)
[[[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]],
 [[1, 2, 3], [4, 4, 4]],
 [[1, 3, 2], [4, 4, 4]]]
```

class **CyclicPermutations_mset**(*mset*)

iterator(*distinct=False*)

EXAMPLES:

```
sage: CyclicPermutations(range(4)).list() # indirect doctest
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
 sage: CyclicPermutations([1,1,1]).list()
 [[1, 1, 1]]
 sage: CyclicPermutations([1,1,1]).list(distinct=True)
 [[1, 1, 1], [1, 1, 1]]
```

list(*distinct=False*)

EXAMPLES:

```
sage: CyclicPermutations(range(4)).list()
[[0, 1, 2, 3],
 [0, 1, 3, 2],
 [0, 2, 1, 3],
 [0, 2, 3, 1],
 [0, 3, 1, 2],
 [0, 3, 2, 1]]
```

class **PatternAvoider**(*n, patterns*)

Permutation(*l*)

Converts l to a permutation.

INPUT:

- an instance of Permutation_class,
- list of integers, viewed as one-line permutation notation,
- string, expressing the permutation in cycle notation,
- list of tuples of integers, the permutation in cycle notation.
- a PermutationGroupElement
- a pair of two tableaux of the same shape, where the second one is standard. This uses the inverse of Robinson Schensted algorithm.

OUTPUT:

- `Permutation_class` object.

EXAMPLES:

```
sage: Permutation([2,1])
[2, 1]
sage: Permutation([2, 1, 4, 5, 3])
[2, 1, 4, 5, 3]
sage: Permutation('(1,2)')
[2, 1]
sage: Permutation('(1,2)(3,4,5)')
[2, 1, 4, 5, 3]
sage: Permutation( ((1,2),(3,4,5)) )
[2, 1, 4, 5, 3]
sage: Permutation( [(1,2),(3,4,5)] )
[2, 1, 4, 5, 3]
sage: Permutation( ((1,2)) )
[2, 1]
sage: Permutation( (1,2) )
[2, 1]
sage: Permutation( ((1,2),) )
[2, 1]
sage: p = Permutation((1, 2, 5)); p
[2, 5, 3, 4, 1]
sage: type(p)
<class 'sage.combinat.permutation.Permutation_class'>
```

Construction from a string in cycle notation

```
sage: p = Permutation( '(4,5)' ); p
[1, 2, 3, 5, 4]
```

The length of the permutation is the maximum integer appearing; add a 1-cycle to increase this:

```
sage: p2 = Permutation( '(4,5)(10)' ); p2
[1, 2, 3, 5, 4, 6, 7, 8, 9, 10]
sage: len(p); len(p2)
5
10
```

We construct a Permutation from a PermutationGroupElement:

```
sage: g = PermutationGroupElement([2,1,3])
sage: Permutation(g)
[2, 1, 3]
```

From a pair of tableaux of the same shape. This uses the inverse of Robinson Schensted algorithm:

```
sage: p = [[1, 4, 7], [2, 5], [3], [6]]
sage: q = [[1, 2, 5], [3, 6], [4], [7]]
sage: P = Tableau(p)
sage: Q = Tableau(q)
sage: Permutation( (p, q) )
[3, 6, 5, 2, 7, 4, 1]
sage: Permutation( [p, q] )
[3, 6, 5, 2, 7, 4, 1]
sage: Permutation( (P, Q) )
[3, 6, 5, 2, 7, 4, 1]
sage: Permutation( [P, Q] )
[3, 6, 5, 2, 7, 4, 1]
```

TESTS:

```
sage: Permutation([()])
[1]
sage: Permutation('()')
[1]
sage: Permutation(())
[1]
```

From a pair of empty tableaux

```
sage: Permutation( ([], []) )
[]
sage: Permutation( [[], []] )
[]
```

PermutationOptions(***kwargs*)

Sets the global options for elements of the permutation class. The defaults are for permutations to be displayed in list notation and the multiplication done from left to right (like in GAP).

display: 'list' - the permutations are displayed in list notation 'cycle' - the permutations are displayed in cycle notation 'singleton' - the permutations are displayed in cycle notation with singleton cycles shown as well.

mult: 'l2r' - the multiplication of permutations is done like composition of functions from left to right. That is, if we think of the permutations p1 and p2 as functions, then (p1*p2)(x) = p2(p1(x)). This is the default in multiplication in GAP. 'r2l' - the multiplication of permutations is done right to left so that (p1*p2)(x) = p1(p2(x))

If no parameters are set, then the function returns a copy of the options dictionary.

Note that these options have no effect on PermutationGroupElements.

EXAMPLES:

```
sage: p213 = Permutation([2,1,3])
sage: p312 = Permutation([3,1,2])
sage: PermutationOptions(mult='l2r', display='list')
sage: po = PermutationOptions()
sage: po['display']
'list'
sage: p213
[2, 1, 3]
sage: PermutationOptions(display='cycle')
sage: p213
(1,2)
sage: PermutationOptions(display='singleton')
```

```
sage: p213
(1,2)(3)
sage: PermutationOptions(display='list')

sage: po['mult']
'l2r'
sage: p213*p312
[1, 3, 2]
sage: PermutationOptions(mult='r2l')
sage: p213*p312
[3, 2, 1]
sage: PermutationOptions(mult='l2r')
```

class **Permutation_class**(*l*)

action(*a*)

Returns the action of the permutation on a list.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: a = range(3)
sage: p.action(a)
[1, 0, 2]
sage: b = [1,2,3,4]
sage: p.action(b)
...
ValueError: len(a) must equal len(self)
```

avoids(*patt*)

Returns True if the permutation avoid the pattern patt and False otherwise.

EXAMPLES:

```
sage: Permutation([6,2,5,4,3,1]).avoids([4,2,3,1])
False
sage: Permutation([6,1,2,5,4,3]).avoids([4,2,3,1])
True
sage: Permutation([6,1,2,5,4,3]).avoids([3,4,1,2])
True
```

bruhat_greater()

Returns the combinatorial class of permutations greater than or equal to p in the Bruhat order.

EXAMPLES:

```
sage: Permutation([4,1,2,3]).bruhat_greater().list()
[[4, 1, 2, 3],
 [4, 1, 3, 2],
 [4, 2, 1, 3],
 [4, 2, 3, 1],
 [4, 3, 1, 2],
 [4, 3, 2, 1]]
```

bruhat_inversions()

Returns the list of inversions of p such that the application of this inversion to p decrements its number of inversions.

Equivalently, it returns the list of pairs (i,j), i < j such that p[i] < p[j] and such that there exists no k between i and j satisfying p[i] < p[k].

EXAMPLES:

```
sage: Permutation([5,2,3,4,1]).bruhat_inversions()
[[0, 1], [0, 2], [0, 3], [1, 4], [2, 4], [3, 4]]
sage: Permutation([6,1,4,5,2,3]).bruhat_inversions()
[[0, 1], [0, 2], [0, 3], [2, 4], [2, 5], [3, 4], [3, 5]]
```

bruhat_inversions_iterator()

Returns the iterator for the inversions of p such that the application of this inversion to p decrements its number of inversions.

EXAMPLES:

```
sage: list(Permutation([5,2,3,4,1]).bruhat_inversions_iterator())
[[0, 1], [0, 2], [0, 3], [1, 4], [2, 4], [3, 4]]
sage: list(Permutation([6,1,4,5,2,3]).bruhat_inversions_iterator())
[[0, 1], [0, 2], [0, 3], [2, 4], [2, 5], [3, 4], [3, 5]]
```

bruhat_lequal(*p2*)

Returns True if self is less than p2 in the Bruhat order.

EXAMPLES:

```
sage: Permutation([2,4,3,1]).bruhat_lequal(Permutation([3,4,2,1]))
True
```

bruhat_pred()

Returns a list of the permutations strictly smaller than p in the Bruhat order such that there is no permutation between one of those and p.

EXAMPLES:

```
sage: Permutation([6,1,4,5,2,3]).bruhat_pred()
[[1, 6, 4, 5, 2, 3],
 [4, 1, 6, 5, 2, 3],
 [5, 1, 4, 6, 2, 3],
 [6, 1, 2, 5, 4, 3],
 [6, 1, 3, 5, 2, 4],
 [6, 1, 4, 2, 5, 3],
 [6, 1, 4, 3, 2, 5]]
```

bruhat_pred_iterator()

An iterator for the permutations strictly smaller than p in the Bruhat order such that there is no permutation between one of those and p.

EXAMPLES:

```
sage: [x for x in Permutation([6,1,4,5,2,3]).bruhat_pred_iterator()]
[[1, 6, 4, 5, 2, 3],
 [4, 1, 6, 5, 2, 3],
 [5, 1, 4, 6, 2, 3],
 [6, 1, 2, 5, 4, 3],
 [6, 1, 3, 5, 2, 4],
 [6, 1, 4, 2, 5, 3],
 [6, 1, 4, 3, 2, 5]]
```

bruhat_smaller()

Returns a the combinatorial class of permutations smaller than or equal to p in the Bruhat order.

EXAMPLES:

```
sage: Permutation([4,1,2,3]).bruhat_smaller().list()
[[1, 2, 3, 4],
 [1, 2, 4, 3],
 [1, 3, 2, 4],
 [1, 4, 2, 3],
```

```
 [2, 1, 3, 4],
 [2, 1, 4, 3],
 [3, 1, 2, 4],
 [4, 1, 2, 3]]
```

bruhat_succ()

Returns a list of the permutations strictly greater than p in the Bruhat order such that there is no permutation between one of those and p.

EXAMPLES:

```
sage: Permutation([6,1,4,5,2,3]).bruhat_succ()
[[6, 4, 1, 5, 2, 3],
 [6, 2, 4, 5, 1, 3],
 [6, 1, 5, 4, 2, 3],
 [6, 1, 4, 5, 3, 2]]
```

bruhat_succ_iterator()

An iterator for the permutations that are strictly greater than p in the Bruhat order such that there is no permutation between one of those and p.

EXAMPLES:

```
sage: [x for x in Permutation([6,1,4,5,2,3]).bruhat_succ_iterator()]
[[6, 4, 1, 5, 2, 3],
 [6, 2, 4, 5, 1, 3],
 [6, 1, 5, 4, 2, 3],
 [6, 1, 4, 5, 3, 2]]
```

complement()

Returns the complement of the permutation which is obtained by replacing each value x in the list with n - x + 1.

EXAMPLES:

```
sage: Permutation([1,2,3]).complement()
[3, 2, 1]
sage: Permutation([1, 3, 2]).complement()
[3, 1, 2]
```

cycle_string(*singletons=False*)

Returns a string of the permutation in cycle notation.

If singletons=True, it includes 1-cycles in the string.

EXAMPLES:

```
sage: Permutation([1,2,3]).cycle_string()
'()'
sage: Permutation([2,1,3]).cycle_string()
'(1,2)'
sage: Permutation([2,3,1]).cycle_string()
'(1,2,3)'
sage: Permutation([2,1,3]).cycle_string(singletons=True)
'(1,2)(3)'
```

cycle_type()

Returns a partition of len(p) corresponding to the cycle type of p. This is a non-increasing sequence of the cycle lengths of p.

EXAMPLES:

```
sage: Permutation([3,1,2,4]).cycle_type()
[3, 1]
```

descent_polynomial()

Returns the descent polynomial of the permutation p.

The descent polynomial of p is the product of all the z[p[i]] where i ranges over the descents of p.

REFERENCES:

•Garsia and Stanton 1984

EXAMPLES:

```
sage: Permutation([2,1,3]).descent_polynomial()
z1
sage: Permutation([4,3,2,1]).descent_polynomial()
z1*z2^2*z3^3
```

descents(*final_descent=False*)

Returns the list of the descents of the permutation p.

A descent of a permutation is an integer i such that p[i] > p[i+1]. With the final_descent option, the last position of a non empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).descents()
[1, 2]
sage: Permutation([1,4,3,2]).descents(final_descent=True)
[1, 2, 3]
```

descents_composition()

Returns the composition corresponding to the descents of the permutation.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).descents_composition()
[2, 2]
```

dict()

Returns a dictionary corresponding to the permutation.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: d = p.dict()
sage: d[1]
2
sage: d[2]
1
sage: d[3]
3
```

fixed_points()

Returns a list of the fixed points of the permutation p.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).fixed_points()
[1, 4]
sage: Permutation([1,2,3,4]).fixed_points()
[1, 2, 3, 4]
```

has_pattern(*patt*)

Returns the boolean answering the question 'Is patt a pattern appearing in permutation p?'

EXAMPLES:

```
sage: Permutation([3,5,1,4,6,2]).has_pattern([1,3,2])
True
```

idescents(*final_descent=False*)

Returns a list of the idescents of self, that is the list of the descents of self's inverse.

With the final_descent option, the last position of a non empty permutation is also considered as a descent.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).idescents()
[1, 2]
sage: Permutation([1,4,3,2]).idescents(final_descent=True)
[1, 2, 3]
```

idescents_signature(*final_descent=False*)

Each position in self is mapped to -1 if it is an idescent and 1 if it is not an idescent.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).idescents()
[1, 2]
sage: Permutation([1,4,3,2]).idescents_signature()
[1, -1, -1, 1]
```

imajor_index(*final_descent=False*)

Returns the inverse major index of the permutation self, which is the major index of the inverse of self.

The major index is the sum of the descents of p. Since our permutation indices are 0-based, we need to add one the number of descents.

EXAMPLES:

```
sage: Permutation([2,1,3]).imajor_index()
1
sage: Permutation([3,4,1,2]).imajor_index()
2
sage: Permutation([4,3,2,1]).imajor_index()
6
```

inverse()

Returns the inverse of a permutation

EXAMPLES:

```
sage: Permutation([3,8,5,10,9,4,6,1,7,2]).inverse()
[8, 10, 1, 6, 3, 7, 9, 2, 5, 4]
sage: Permutation([2, 4, 1, 5, 3]).inverse()
[3, 1, 5, 2, 4]
```

inversions()

Returns a list of the inversions of permutation p.

EXAMPLES:

```
sage: Permutation([3,2,4,1,5]).inversions()
[[0, 1], [0, 3], [1, 3], [2, 3]]
```

is_even()

Returns True if the permutation p is even and false otherwise.

EXAMPLES:

```
sage: Permutation([1,2,3]).is_even()
True
sage: Permutation([2,1,3]).is_even()
False
```

ishift(*i*)

Returns an the i-shift of self. If an i-shift of self can't be performed, then None is returned.

An i-shift can be applied when i is not in between i-1 and i+1. The i-shift moves i to the other side, and leaves the relative positions of i-1 and i+1 in place.

EXAMPLES: Here, 2 is to the left of both 1 and 3. A 2-shift can be applied which moves the 2 to the right and leaves 1 and 3 in their same relative order.

```
sage: Permutation([2,1,3]).ishift(2)
[1, 3, 2]
```

Note that the movement is done in place:

```
sage: Permutation([2,4,1,3]).ishift(2)
[1, 4, 3, 2]
```

Since 2 is between 1 and 3 in [1,2,3], an 2-shift cannot be applied.

```
sage: Permutation([1,2,3]).ishift(2)
[1, 2, 3]
```

iswitch(*i*)

Returns an the i-switch of self. If an i-switch of self can't be performed, then self is returned.

An i-shift can be applied when i is not in between i-1 and i+1. The i-shift moves i to the other side, and switches the relative positions of i-1 and i+1 in place.

EXAMPLES: Here, 2 is to the left of both 1 and 3. A 2-switch can be applied which moves the 2 to the right and switches the relative order between 1 and 3.

```
sage: Permutation([2,1,3]).iswitch(2)
[3, 1, 2]
```

Note that the movement is done in place:

```
sage: Permutation([2,4,1,3]).iswitch(2)
[3, 4, 1, 2]
```

Since 2 is between 1 and 3 in [1,2,3], an 2-switch cannot be applied.

```
sage: Permutation([1,2,3]).iswitch(2)
[1, 2, 3]
```

left_tableau()

Returns the right standard tableau after performing the RSK algorithm on self.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).left_tableau()
[[1, 2], [3], [4]]
```

length()

Returns the length of a permutation p. The length is given by the number of inversions of p.

EXAMPLES:

```
sage: Permutation([5, 1, 3, 2, 4]).length()
5
```

longest_increasing_subsequence()

Returns a list of the longest increasing subsequences of the permutation p.

EXAMPLES:

```
sage: Permutation([2,3,4,1]).longest_increasing_subsequence()
[[2, 3, 4]]
```

major_index(*final_descent=False*)

Returns the major index of the permutation p.

The major index is the sum of the descents of p. Since our permutation indices are 0-based, we need to add one the number of descents.

EXAMPLES:

```
sage: Permutation([2,1,3]).major_index()
1
sage: Permutation([3,4,1,2]).major_index()
2
sage: Permutation([4,3,2,1]).major_index()
6
```

next()

Returns the permutation that follows p in lexicographic order. If p is the last permutation, then next returns false.

EXAMPLES:

```
sage: p = Permutation([1, 3, 2])
sage: p.next()
[2, 1, 3]
sage: p = Permutation([4,3,2,1])
sage: p.next()
False
```

number_of_descents(*final_descent=False*)

Returns the number of descents of the permutation p.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).number_of_descents()
2
sage: Permutation([1,4,3,2]).number_of_descents(final_descent=True)
3
```

number_of_fixed_points()

Returns the number of fixed points of the permutation p.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).number_of_fixed_points()
2
sage: Permutation([1,2,3,4]).number_of_fixed_points()
4
```

number_of_idescents(*final_descent=False*)

Returns the number of descents of the permutation p.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).number_of_idescents()
2
sage: Permutation([1,4,3,2]).number_of_idescents(final_descent=True)
3
```

number_of_inversions()

Returns the number of inversions in the permutation p.

An inversion of a permutation is a pair of elements (p[i],p[j]) with i < j and p[i] > p[j].

REFERENCES:

- http://mathworld.wolfram.com/PermutationInversion.html

EXAMPLES:

```
sage: Permutation([3,2,4,1,5]).number_of_inversions()
4
sage: Permutation([1, 2, 6, 4, 7, 3, 5]).number_of_inversions()
6
```

number_of_peaks()

Returns the number of peaks of the permutation p.

A peak of a permutation is an integer i such that p[i-1] < p[i] and p[i] > p[i+1].

EXAMPLES:

```
sage: Permutation([1,3,2,4,5]).number_of_peaks()
1
sage: Permutation([4,1,3,2,6,5]).number_of_peaks()
2
```

number_of_recoils()

Returns the number of recoils of the permutation p.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).number_of_recoils()
2
```

number_of_saliances()

Returns the number of saliances of the permutation p.

EXAMPLES:

```
sage: Permutation([2,3,1,5,4]).number_of_saliances()
2
sage: Permutation([5,4,3,2,1]).number_of_saliances()
5
```

pattern_positions(*patt*)

Returns the list of positions where the pattern patt appears in p.

EXAMPLES:

```
sage: Permutation([3,5,1,4,6,2]).pattern_positions([1,3,2])
[[0, 1, 3], [2, 3, 5], [2, 4, 5]]
```

peaks()

Returns a list of the peaks of the permutation p.

A peak of a permutation is an integer i such that p[i-1] < p[i] and p[i] > p[i+1].

EXAMPLES:

```
sage: Permutation([1,3,2,4,5]).peaks()
[1]
sage: Permutation([4,1,3,2,6,5]).peaks()
[2, 4]
```

permutohedron_greater(*side='right'*)

Returns a list of permutations greater than or equal to p in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option side='left', then they will be done in the left permutohedron.

EXAMPLES:

```
sage: Permutation([4,2,1,3]).permutohedron_greater()
[[4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 2, 1]]
sage: Permutation([4,2,1,3]).permutohedron_greater(side='left')
[[4, 2, 1, 3], [4, 3, 1, 2], [4, 3, 2, 1]]
```

permutohedron_lequal(*p2, side='right'*)

Returns True if self is less than p2 in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option side='left', then they will be done in the left permutohedron.

EXAMPLES:

```
sage: p = Permutation([3,2,1,4])
sage: p.permutohedron_lequal(Permutation([4,2,1,3]))
False
sage: p.permutohedron_lequal(Permutation([4,2,1,3]), side='left')
True
```

permutohedron_pred(*side='right'*)

Returns a list of the permutations strictly smaller than p in the permutohedron order such that there is no permutation between one of those and p.

By default, the computations are done in the right permutohedron. If you pass the option side='left', then they will be done in the left permutohedron.

EXAMPLES:

```
sage: p = Permutation([4,2,1,3])
sage: p.permutohedron_pred()
[[2, 4, 1, 3], [4, 1, 2, 3]]
sage: p.permutohedron_pred(side='left')
[[4, 1, 2, 3], [3, 2, 1, 4]]
```

permutohedron_smaller(*side='right'*)

Returns a list of permutations smaller than or equal to p in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option side='left', then they will be done in the left permutohedron.

EXAMPLES:

```
sage: Permutation([4,2,1,3]).permutohedron_smaller()
[[1, 2, 3, 4],
 [1, 2, 4, 3],
 [1, 4, 2, 3],
 [2, 1, 3, 4],
 [2, 1, 4, 3],
 [2, 4, 1, 3],
 [4, 1, 2, 3],
 [4, 2, 1, 3]]

sage: Permutation([4,2,1,3]).permutohedron_smaller(side='left')
[[1, 2, 3, 4],
 [1, 3, 2, 4],
 [2, 1, 3, 4],
 [2, 3, 1, 4],
 [3, 1, 2, 4],
 [3, 2, 1, 4],
 [4, 1, 2, 3],
 [4, 2, 1, 3]]
```

permutohedron_succ(*side='right'*)

Returns a list of the permutations strictly greater than p in the permutohedron order such that there is no permutation between one of those and p.

By default, the computations are done in the right permutohedron. If you pass the option side='left', then they will be done in the left permutohedron.

EXAMPLES:

```
sage: p = Permutation([4,2,1,3])
sage: p.permutohedron_succ()
[[4, 2, 3, 1]]
sage: p.permutohedron_succ(side='left')
[[4, 3, 1, 2]]
```

prev()

Returns the permutation that comes directly before p in lexicographic order. If p is the first permutation, then it returns False.

EXAMPLES:

```
sage: p = Permutation([1,2,3])
sage: p.prev()
False
sage: p = Permutation([1,3,2])
sage: p.prev()
[1, 2, 3]
```

rank()

Returns the rank of a permutation in lexicographic ordering.

EXAMPLES:

```
sage: Permutation([1,2,3]).rank()
0
sage: Permutation([1, 2, 4, 6, 3, 5]).rank()
10
sage: perms = Permutations(6).list()
sage: [p.rank() for p in perms ] == range(factorial(6))
True
```

recoils()

Returns the list of the positions of the recoils of the permutation p.

A recoil of a permutation is an integer i such that i+1 is to the left of it.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).recoils()
[2, 3]
```

recoils_composition()

Returns the composition corresponding to recoils of the permutation.

EXAMPLES:

```
sage: Permutation([1,3,2,4]).recoils_composition()
[3]
```

reduced_word()

Returns the reduced word of a permutation.

EXAMPLES:

```
sage: Permutation([3,5,4,6,2,1]).reduced_word()
[2, 1, 4, 3, 2, 4, 3, 5, 4, 5]
```

reduced_word_lexmin()

Returns a lexicographically minimal reduced word of a permutation.

EXAMPLES:

```
sage: Permutation([3,4,2,1]).reduced_word_lexmin()
[1, 2, 1, 3, 2]
```

reduced_words()

Returns a list of the reduced words of the permutation p.

EXAMPLES:

```
sage: Permutation([2,1,3]).reduced_words()
[[1]]
sage: Permutation([3,1,2]).reduced_words()
[[2, 1]]
sage: Permutation([3,2,1]).reduced_words()
[[1, 2, 1], [2, 1, 2]]
sage: Permutation([3,2,4,1]).reduced_words()
[[1, 2, 3, 1], [1, 2, 1, 3], [2, 1, 2, 3]]
```

remove_extra_fixed_points()

Returns the permutation obtained by removing any fixed points at the end of self.

EXAMPLES:

```
sage: Permutation([2,1,3]).remove_extra_fixed_points()
[2, 1]
sage: Permutation([1,2,3,4]).remove_extra_fixed_points()
[1]
```

reverse()

Returns the permutation obtained by reversing the list.

EXAMPLES:

```
sage: Permutation([3,4,1,2]).reverse()
[2, 1, 4, 3]
sage: Permutation([1,2,3,4,5]).reverse()
[5, 4, 3, 2, 1]
```

right_tableau()

Returns the right standard tableau after performing the RSK algorithm on self.

EXAMPLES:

```
sage: Permutation([1,4,3,2]).right_tableau()
[[1, 2], [3], [4]]
```

robinson_schensted()

Returns the pair of standard tableau obtained by running the Robinson-Schensted Algorithm on self.

EXAMPLES:

```
sage: p = Permutation([6,2,3,1,7,5,4])
sage: p.robinson_schensted()
[[[1, 3, 4], [2, 5], [6, 7]], [[1, 3, 5], [2, 6], [4, 7]]]
```

TESTS:

The empty permutation:

```
sage: p = Permutation([])
sage: p.robinson_schensted()
[[], []]
```

runs()

Returns a list of the runs in the permutation p.

REFERENCES:

- http://mathworld.wolfram.com/PermutationRun.html

EXAMPLES:

```
sage: Permutation([1,2,3,4]).runs()
[[1, 2, 3, 4]]
sage: Permutation([4,3,2,1]).runs()
[[4], [3], [2], [1]]
sage: Permutation([2,4,1,3]).runs()
[[2, 4], [1, 3]]
```

saliances()

Returns a list of the saliances of the permutation p.

A saliance of a permutation p is an integer i such that p[i] > p[j] for all j > i.

EXAMPLES:

```
sage: Permutation([2,3,1,5,4]).saliances()
[3, 4]
sage: Permutation([5,4,3,2,1]).saliances()
[0, 1, 2, 3, 4]
```

signature(*p*)

Returns the signature of a permutation.

EXAMPLES:

```
sage: Permutation([4, 2, 3, 1, 5]).signature()
-1
```

to_cycles(*singletons=True*)

Returns the permutation p as a list of disjoint cycles.

EXAMPLES:

```
sage: Permutation([2,1,3,4]).to_cycles()
[(1, 2), (3,), (4,)]
sage: Permutation([2,1,3,4]).to_cycles(singletons=False)
[(1, 2)]
```

to_inversion_vector()

Returns the inversion vector of a permutation p.

If iv is the inversion vector, then iv[i] is the number of elements larger than i that appear to the left of i in the permutation.

EXAMPLES:

```
sage: Permutation([5,9,1,8,2,6,4,7,3]).to_inversion_vector()
[2, 3, 6, 4, 0, 2, 2, 1, 0]
sage: Permutation([8,7,2,1,9,4,6,5,10,3]).to_inversion_vector()
[3, 2, 7, 3, 4, 3, 1, 0, 0, 0]
sage: Permutation([3,2,4,1,5]).to_inversion_vector()
[3, 1, 0, 0, 0]
```

to_lehmer_cocode()

Returns the Lehmer cocode of p.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: p.to_lehmer_cocode()
[0, 1, 0]
sage: q = Permutation([3,1,2])
sage: q.to_lehmer_cocode()
[0, 1, 1]
```

to_lehmer_code()

Returns the Lehmer code of the permutation p.

EXAMPLES:

```
sage: p = Permutation([2,1,3])
sage: p.to_lehmer_code()
[1, 0, 0]
sage: q = Permutation([3,1,2])
sage: q.to_lehmer_code()
[2, 0, 0]
```

to_major_code(*final_descent=False*)

Returns the major code of the permutation p, which is defined as the list [m1-m2, m2-m3,..,mn] where mi := maj(pi) is the major indices of the permutation math obtained by erasing the letters smaller than math in p.

REFERENCES:

- Carlitz, L. 'q-Bernoulli and Eulerian Numbers' Trans. Amer. Math. Soc. 76 (1954) 332-350 Skandera, M. 'An Eulerian Partner for Inversions', Sem. Lothar. Combin. 46 (2001) B46d.

EXAMPLES:

```
sage: Permutation([9,3,5,7,2,1,4,6,8]).to_major_code()
[5, 0, 1, 0, 1, 2, 0, 1, 0]
sage: Permutation([2,8,4,3,6,7,9,5,1]).to_major_code()
[8, 3, 3, 1, 4, 0, 1, 0, 0]
```

to_matrix()

Returns a matrix representing the permutation.

EXAMPLES:

```
sage: Permutation([1,2,3]).to_matrix()
[1 0 0]
[0 1 0]
[0 0 1]

sage: Permutation([1,3,2]).to_matrix()
[1 0 0]
[0 0 1]
[0 1 0]
```

Notice that matrix multiplication corresponds to permutation multiplication only when the permutation option mult='r2l'

```
sage: PermutationOptions(mult='r2l')
sage: p = Permutation([2,1,3])
sage: q = Permutation([3,1,2])
sage: (p*q).to_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
sage: p.to_matrix()*q.to_matrix()
[0 0 1]
[0 1 0]
[1 0 0]
sage: PermutationOptions(mult='l2r')
sage: (p*q).to_matrix()
[1 0 0]
[0 0 1]
[0 1 0]
```

to_permutation_group_element()

Returns a PermutationGroupElement equal to self.

EXAMPLES:

```
sage: Permutation([2,1,4,3]).to_permutation_group_element()
(1,2)(3,4)
sage: Permutation([1,2,3]).to_permutation_group_element()
()
```

to_tableau_by_shape(*shape*)

Returns a tableau of shape shape with the entries in self.

EXAMPLES:

```
sage: Permutation([3,4,1,2,5]).to_tableau_by_shape([3,2])
[[1, 2, 5], [3, 4]]
sage: Permutation([3,4,1,2,5]).to_tableau_by_shape([3,2]).to_permutation()
[3, 4, 1, 2, 5]
```

weak_excedences()

Returns all the numbers self[i] such that self[i] = i+1.

EXAMPLES:

```
sage: Permutation([1,4,3,2,5]).weak_excedences()
[1, 4, 3, 5]
```

Permutations(*n=None, k=None, **kwargs*)

Returns a combinatorial class of permutations.

Permutations(n) returns the class of permutations of n, if n is an integer, list, set, or string.

Permutations(n, k) returns the class of permutations of n (where n is any of the above things) of length k; k must be an integer.

Valid keyword arguments are: 'descents', 'bruhat_smaller', 'bruhat_greater', 'recoils_finer', 'recoils_fatter', 'recoils', and 'avoiding'. With the exception of 'avoiding', you cannot specify n or k along with a keyword.

Permutations(descents=list) returns the class of permutations with descents in the positions specified by 'list'.

Permutations(bruhat_smaller,greater=p) returns the class of permutations smaller or greater, respectively, than the given permutation in Bruhat order.

Permutations(recoils=p) returns the class of permutations whose recoils composition is p.

Permutations(recoils_fatter,finer=p) returns the class of permutations whose recoils composition is fatter or finer, respectively, than the given permutation.

Permutations(n, avoiding=P) returns the class of permutations of n avoiding P. Here P may be a single permutation or a list of permutations; the returned class will avoid all patterns in P.

EXAMPLES:

```
sage: p = Permutations(3); p
Standard permutations of 3
sage: p.list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]

sage: p = Permutations(3, 2); p
Permutations of {1,...,3} of length 2
sage: p.list()
[[1, 2], [1, 3], [2, 1], [2, 3], [3, 1], [3, 2]]

sage: p = Permutations(['c', 'a', 't']); p
Permutations of the set ['c', 'a', 't']
sage: p.list()
[['c', 'a', 't'],
```

```
 ['c', 't', 'a'],
 ['a', 'c', 't'],
 ['a', 't', 'c'],
 ['t', 'c', 'a'],
 ['t', 'a', 'c']]

sage: p = Permutations(['c', 'a', 't'], 2); p
Permutations of the set ['c', 'a', 't'] of length 2
sage: p.list()
[['c', 'a'], ['c', 't'], ['a', 'c'], ['a', 't'], ['t', 'c'], ['t', 'a']]

sage: p = Permutations([1,1,2]); p
Permutations of the multi-set [1, 1, 2]
sage: p.list()
[[1, 1, 2], [1, 2, 1], [2, 1, 1]]

sage: p = Permutations([1,1,2], 2); p
Permutations of the multi-set [1, 1, 2] of length 2
sage: p.list()
[[1, 1], [1, 2], [2, 1]]

sage: p = Permutations(descents=[1,3]); p
Standard permutations of 4 with descents [1, 3]
sage: p.list()
[[1, 3, 2, 4], [1, 4, 2, 3], [2, 3, 1, 4], [2, 4, 1, 3], [3, 4, 1, 2]]

sage: p = Permutations(bruhat_smaller=[1,3,2,4]); p
Standard permutations that are less than or equal to [1, 3, 2, 4] in the Bruhat order
sage: p.list()
[[1, 2, 3, 4], [1, 3, 2, 4]]

sage: p = Permutations(bruhat_greater=[4,2,3,1]); p
Standard permutations that are greater than or equal to [4, 2, 3, 1] in the Bruhat order
sage: p.list()
[[4, 2, 3, 1], [4, 3, 2, 1]]

sage: p = Permutations(recoils_finer=[2,1]); p
Standard permutations whose recoils composition is finer than [2, 1]
sage: p.list()
[[1, 2, 3], [1, 3, 2], [3, 1, 2]]

sage: p = Permutations(recoils_fatter=[2,1]); p
Standard permutations whose recoils composition is fatter than [2, 1]
sage: p.list()
[[1, 3, 2], [3, 1, 2], [3, 2, 1]]

sage: p = Permutations(recoils=[2,1]); p
Standard permutations whose recoils composition is [2, 1]
sage: p.list()
[[1, 3, 2], [3, 1, 2]]
```

```
sage: p = Permutations(4, avoiding=[1,3,2]); p
Standard permutations of 4 avoiding [1, 3, 2]
sage: p.list()
[[4, 1, 2, 3],
 [4, 2, 1, 3],
 [4, 2, 3, 1],
 [4, 3, 1, 2],
 [4, 3, 2, 1],
 [3, 4, 1, 2],
 [3, 4, 2, 1],
 [2, 3, 4, 1],
 [3, 2, 4, 1],
 [1, 2, 3, 4],
 [2, 1, 3, 4],
 [2, 3, 1, 4],
 [3, 1, 2, 4],
 [3, 2, 1, 4]]

sage: p = Permutations(5, avoiding=[[3,4,1,2], [4,2,3,1]]); p
Standard permutations of 5 avoiding [[3, 4, 1, 2], [4, 2, 3, 1]]
sage: p.cardinality()
88
sage: p.random_element()
[5, 1, 2, 4, 3]
```

class **Permutations_mset**(*mset*)

cardinality()
EXAMPLES:

```
sage: Permutations([1,2,2]).cardinality()
3
```

class **Permutations_msetk**(*mset, k*)

list()
EXAMPLES:

```
sage: Permutations([1,2,2],2).list()
[[1, 2], [2, 1], [2, 2]]
```

class **Permutations_nk**(*n, k*)

cardinality()
EXAMPLES:

```
sage: Permutations(3,0).cardinality()
1
sage: Permutations(3,1).cardinality()
3
sage: Permutations(3,2).cardinality()
6
sage: Permutations(3,3).cardinality()
6
sage: Permutations(3,4).cardinality()
0
```

random_element()
EXAMPLES:

```
sage: Permutations(3,2).random_element()
[0, 1]
```

class **Permutations_set**(*s*)

cardinality()
EXAMPLES:

```
sage: Permutations([1,2,3]).cardinality()
6
```

random_element()
EXAMPLES:

```
sage: Permutations([1,2,3]).random_element()
[1, 2, 3]
```

class **Permutations_setk**(*s*, *k*)

random_element()
EXAMPLES:

```
sage: Permutations([1,2,3],2).random_element()
[1, 2]
```

class **StandardPermutations_all**()

class **StandardPermutations_avoiding_12**(*n*)

list()
EXAMPLES:

```
sage: Permutations(3, avoiding=[1,2]).list()
[[3, 2, 1]]
```

class **StandardPermutations_avoiding_123**(*n*)

cardinality()
EXAMPLES:

```
sage: Permutations(5, avoiding=[1, 2, 3]).cardinality()
42
sage: len( Permutations(5, avoiding=[1, 2, 3]).list() )
42
```

class **StandardPermutations_avoiding_132**(*n*)

cardinality()
EXAMPLES:

```
sage: Permutations(5, avoiding=[1, 3, 2]).cardinality()
42
sage: len( Permutations(5, avoiding=[1, 3, 2]).list() )
42
```

class **StandardPermutations_avoiding_21**(*n*)

list()
EXAMPLES:

```
sage: Permutations(3, avoiding=[2,1]).list()
[[1, 2, 3]]
```

class **StandardPermutations_avoiding_213**(*n*)

cardinality()
EXAMPLES:

```
sage: Permutations(5, avoiding=[2, 1, 3]).cardinality()
42
sage: len( Permutations(5, avoiding=[2, 1, 3]).list() )
42
```

class **StandardPermutations_avoiding_231**(*n*)

cardinality()
EXAMPLES:

```
sage: Permutations(5, avoiding=[2, 3, 1]).cardinality()
42
sage: len( Permutations(5, avoiding=[2, 3, 1]).list() )
42
```

class **StandardPermutations_avoiding_312**(*n*)

cardinality()
EXAMPLES:

```
sage: Permutations(5, avoiding=[3, 1, 2]).cardinality()
42
sage: len( Permutations(5, avoiding=[3, 1, 2]).list() )
42
```

class **StandardPermutations_avoiding_321**(*n*)

cardinality()
EXAMPLES:

```
sage: Permutations(5, avoiding=[3, 2, 1]).cardinality()
42
sage: len( Permutations(5, avoiding=[3, 2, 1]).list() )
42
```

class **StandardPermutations_avoiding_generic**(*n*, *a*)

class **StandardPermutations_bruhat_greater**(*p*)

list()
Returns a list of permutations greater than or equal to p in the Bruhat order.
EXAMPLES:

```
sage: Permutations(bruhat_greater=[4,1,2,3]).list()
[[4, 1, 2, 3],
 [4, 1, 3, 2],
```

```
 [4, 2, 1, 3],
 [4, 2, 3, 1],
 [4, 3, 1, 2],
 [4, 3, 2, 1]]
```

class StandardPermutations_bruhat_smaller(*p*)

list()

Returns a list of permutations smaller than or equal to p in the Bruhat order.

EXAMPLES:

```
sage: Permutations(bruhat_smaller=[4,1,2,3]).list()
[[1, 2, 3, 4],
 [1, 2, 4, 3],
 [1, 3, 2, 4],
 [1, 4, 2, 3],
 [2, 1, 3, 4],
 [2, 1, 4, 3],
 [3, 1, 2, 4],
 [4, 1, 2, 3]]
```

class StandardPermutations_descents(*d, n*)

first()

Returns the first permutation with descents d.

EXAMPLES:

```
sage: Permutations(descents=([1,0,4,8],12)).first()
[3, 2, 1, 4, 6, 5, 7, 8, 10, 9, 11, 12]
```

last()

Returns the last permutation with descents d.

EXAMPLES:

```
sage: Permutations(descents=([1,0,4,8],12)).last()
[12, 11, 8, 9, 10, 4, 5, 6, 7, 1, 2, 3]
```

list()

Returns a list of all the permutations that have the descents d.

EXAMPLES:

```
sage: Permutations(descents=([2,4,0],5)).list()
[[2, 1, 4, 3, 5],
 [2, 1, 5, 3, 4],
 [3, 1, 4, 2, 5],
 [3, 1, 5, 2, 4],
 [4, 1, 3, 2, 5],
 [5, 1, 3, 2, 4],
 [4, 1, 5, 2, 3],
 [5, 1, 4, 2, 3],
 [3, 2, 4, 1, 5],
 [3, 2, 5, 1, 4],
 [4, 2, 3, 1, 5],
 [5, 2, 3, 1, 4],
 [4, 2, 5, 1, 3],
 [5, 2, 4, 1, 3],
 [4, 3, 5, 1, 2],
 [5, 3, 4, 1, 2]]
```

object_class()

class **StandardPermutations_n**(*n*)

cardinality()
EXAMPLES:

```
sage: Permutations(0).cardinality()
1
sage: Permutations(3).cardinality()
6
sage: Permutations(4).cardinality()
24
```

identity()
Returns the identity permutation of length n.
EXAMPLES:

```
sage: Permutations(4).identity()
[1, 2, 3, 4]
sage: Permutations(0).identity()
[]
```

random_element()
EXAMPLES:

```
sage: Permutations(4).random_element()
[1, 3, 2, 4]
```

rank(*p*)
EXAMPLES:

```
sage: SP3 = Permutations(3)
sage: map(SP3.rank, SP3)
[0, 1, 2, 3, 4, 5]
sage: SP0 = Permutations(0)
sage: map(SP0.rank, SP0)
[0]
```

unrank(*r*)
EXAMPLES:

```
sage: SP3 = Permutations(3)
sage: l = map(SP3.unrank, range(6))
sage: l == SP3.list()
True
sage: SP0 = Permutations(0)
sage: l = map(SP0.unrank, range(1))
sage: l == SP0.list()
True
```

class **StandardPermutations_recoils**(*recoils*)

list()
Returns a list of all of the permutations whose recoils composition is equal to recoils.
EXAMPLES:

```
sage: Permutations(recoils=[2,2]).list()
[[1, 3, 2, 4], [1, 3, 4, 2], [3, 1, 2, 4], [3, 1, 4, 2], [3, 4, 1, 2]]
```

object_class()

class StandardPermutations_recoilsfatter(*recoils*)

list()

Returns a list of all of the permutations whose recoils composition is fatter than recoils.

EXAMPLES:

```
sage: Permutations(recoils_fatter=[2,2]).list()
[[1, 3, 2, 4],
 [1, 3, 4, 2],
 [1, 4, 3, 2],
 [3, 1, 2, 4],
 [3, 1, 4, 2],
 [3, 2, 1, 4],
 [3, 2, 4, 1],
 [3, 4, 1, 2],
 [3, 4, 2, 1],
 [4, 1, 3, 2],
 [4, 3, 1, 2],
 [4, 3, 2, 1]]
```

object_class()

class StandardPermutations_recoilsfiner(*recoils*)

list()

Returns a list of all of the permutations whose recoils composition is finer than recoils.

EXAMPLES:

```
sage: Permutations(recoils_finer=[2,2]).list()
[[1, 2, 3, 4],
 [1, 3, 2, 4],
 [1, 3, 4, 2],
 [3, 1, 2, 4],
 [3, 1, 4, 2],
 [3, 4, 1, 2]]
```

object_class()

bruhat_lequal(*p1*, *p2*)

Returns True if p1 is less than p2 in the Bruhat order.

Algorithm from mupad-combinat.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.bruhat_lequal([2,4,3,1],[3,4,2,1])
True
```

descents_composition_first(*dc*)

Computes the smallest element of a descent class having a descent decomposition dc.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.descents_composition_first([1,1,3,4,3])
[3, 2, 1, 4, 6, 5, 7, 8, 10, 9, 11, 12]
```

descents_composition_last(*dc*)
Returns the largest element of a descent class having a descent decomposition dc.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.descents_composition_last([1,1,3,4,3])
[12, 11, 8, 9, 10, 4, 5, 6, 7, 1, 2, 3]
```

descents_composition_list(*dc*)
Returns a list of all the permutations that have a descent compositions dc.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.descents_composition_list([1,2,2])
[[2, 1, 4, 3, 5],
 [2, 1, 5, 3, 4],
 [3, 1, 4, 2, 5],
 [3, 1, 5, 2, 4],
 [4, 1, 3, 2, 5],
 [5, 1, 3, 2, 4],
 [4, 1, 5, 2, 3],
 [5, 1, 4, 2, 3],
 [3, 2, 4, 1, 5],
 [3, 2, 5, 1, 4],
 [4, 2, 3, 1, 5],
 [5, 2, 3, 1, 4],
 [4, 2, 5, 1, 3],
 [5, 2, 4, 1, 3],
 [4, 3, 5, 1, 2],
 [5, 3, 4, 1, 2]]
```

from_cycles(*n*, *cycles*)
Returns the permutation corresponding to cycles.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_cycles(4, [[1,2]])
[2, 1, 3, 4]
```

from_inversion_vector(*iv*)
Returns the permutation corresponding to inversion vector iv.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_inversion_vector([3,1,0,0,0])
[3, 2, 4, 1, 5]
sage: permutation.from_inversion_vector([2,3,6,4,0,2,2,1,0])
[5, 9, 1, 8, 2, 6, 4, 7, 3]
```

from_lehmer_code(*lehmer*)
Returns the permutation with Lehmer code lehmer.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: Permutation([2,1,5,4,3]).to_lehmer_code()
[1, 0, 2, 1, 0]
sage: permutation.from_lehmer_code(_)
[2, 1, 5, 4, 3]
```

from_major_code(*mc, final_descent=False*)

Returns the permutation corresponding to major code mc.

REFERENCES:

- Skandera, M. 'An Eulerian Partner for Inversions', Sem. Lothar. Combin. 46 (2001) B46d.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_major_code([5, 0, 1, 0, 1, 2, 0, 1, 0])
[9, 3, 5, 7, 2, 1, 4, 6, 8]
sage: permutation.from_major_code([8, 3, 3, 1, 4, 0, 1, 0, 0])
[2, 8, 4, 3, 6, 7, 9, 5, 1]
sage: Permutation([2,1,6,4,7,3,5]).to_major_code()
[3, 2, 0, 2, 2, 0, 0]
sage: permutation.from_major_code([3, 2, 0, 2, 2, 0, 0])
[2, 1, 6, 4, 7, 3, 5]
```

from_permutation_group_element(*pge*)

Returns a Permutation give a PermutationGroupElement pge.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: pge = PermutationGroupElement([(1,2),(3,4)])
sage: permutation.from_permutation_group_element(pge)
[2, 1, 4, 3]
```

from_rank(*n, rank*)

Returns the permutation with the specified lexicographic rank. The permutation is of the set [1,...,n].

The permutation is computed without iterating through all of the permutations with lower rank. This makes it efficient for large permutations.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: Permutation([3, 6, 5, 4, 2, 1]).rank()
359
sage: [permutation.from_rank(3, i) for i in range(6)]
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
sage: Permutations(6)[10]
[1, 2, 4, 6, 3, 5]
sage: permutation.from_rank(6,10)
[1, 2, 4, 6, 3, 5]
```

from_reduced_word(*rw*)

Returns the permutation corresponding to the reduced word rw.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.from_reduced_word([3,2,3,1,2,3,1])
[3, 4, 2, 1]
sage: permutation.from_reduced_word([])
[]
```

permutohedron_lequal(*p1, p2, side='right'*)

Returns True if p1 is less than p2in the permutohedron order.

By default, the computations are done in the right permutohedron. If you pass the option side='left', then they will be done in the left permutohedron.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.permutohedron_lequal(Permutation([3,2,1,4]),Permutation([4,2,1,3]))
False
sage: permutation.permutohedron_lequal(Permutation([3,2,1,4]),Permutation([4,2,1,3]), side='left
True
```

robinson_schensted_inverse(*p, q*)

Returns the permutation corresponding to the pair of tableaux (p, q) using the inverse of Robinson-Schensted algorithm.

INPUT:

- p, q: two tableaux of the same shape and where q is standard.

EXAMPLES:

```
sage: from sage.combinat.permutation import robinson_schensted_inverse
sage: t1 = Tableau([[1, 2, 5], [3], [4]])
sage: t2 = Tableau([[1, 2, 3], [4], [5]])
sage: robinson_schensted_inverse(t1, t2)
[1, 4, 5, 3, 2]
sage: robinson_schensted_inverse(t1, t1)
[1, 4, 3, 2, 5]
sage: robinson_schensted_inverse(t2, t2)
[1, 2, 5, 4, 3]
sage: robinson_schensted_inverse(t2, t1)
[1, 5, 4, 2, 3]
```

If the first tableau is semistandard:

```
sage: p = Tableau([[1,2,2]]); q = Tableau([[1,2,3]])
sage: robinson_schensted_inverse(p, q)
[1, 2, 2]
sage: _.robinson_schensted()
[[[1, 2, 2]], [[1, 2, 3]]]
```

Note that currently the constructor of Tableau accept as input lists that are not even tableaux but only filling of a parition diagram. This feature should not be used with robinson_schensted_inverse.

TESTS:

From empty tableaux:

```
sage: robinson_schensted_inverse(Tableau([]), Tableau([]))
[]
```

This function is the inverse of robinson_shensted:

```
sage: f = lambda p: robinson_schensted_inverse(*p.robinson_schensted())
sage: all(p == f(p) for n in range(7) for p in Permutations(n))
True

sage: n = ZZ.random_element(200)
sage: p = Permutations(n).random_element()
sage: is_fine = True if p == f(p) else p ; is_fine
True
```

Both tableaux must be of the same shape:

```
sage: robinson_schensted_inverse(Tableau([[1,2,3]]), Tableau([[1,2]]))
...
ValueError: p(=[[1, 2, 3]]) and q(=[[1, 2]]) must have the same shape
```

The second tableau must be standard:

```
sage: robinson_schensted_inverse(Tableau([[1,2,3]]), Tableau([[1,3,2]]))
...
ValueError: q(=[[1, 3, 2]]) must be standard
```

to_standard(*p*)

Returns a standard permutation corresponding to the permutation p.

EXAMPLES:

```
sage: import sage.combinat.permutation as permutation
sage: permutation.to_standard([4,2,7])
[2, 1, 3]
sage: permutation.to_standard([1,2,3])
[1, 2, 3]
```

17.24 q-Analogues

q_binomial(*n, k, p=None*)

Returns the q-binomial coefficient.

If p is unspecified, then it defaults to using the generator q for a univariate polynomial ring over the integers.

EXAMPLES:

```
sage: import sage.combinat.q_analogues as q_analogues
sage: q_analogues.q_binomial(4,2)
q^4 + q^3 + 2*q^2 + q + 1
sage: p = ZZ['p'].0
sage: q_analogues.q_binomial(4,2,p)
p^4 + p^3 + 2*p^2 + p + 1
```

q_factorial(*n, p=None*)

Returns the q-analogue of the n!.

If p is unspecified, then it defaults to using the generator q for a univariate polynomial ring over the integers.

EXAMPLES:

```
sage: import sage.combinat.q_analogues as q_analogues
sage: q_analogues.q_factorial(3)
q^3 + 2*q^2 + 2*q + 1
sage: p = ZZ['p'].0
sage: q_analogues.q_factorial(3, p)
p^3 + 2*p^2 + 2*p + 1
```

q_int(*n, p=None*)

Returns the q-analogue of the integer n.

If p is unspecified, then it defaults to using the generator q for a univariate polynomial ring over the integers.

EXAMPLES:

```
sage: import sage.combinat.q_analogues as q_analogues
sage: q_analogues.q_int(3)
q^2 + q + 1
sage: p = ZZ['p'].0
sage: q_analogues.q_int(3,p)
p^2 + p + 1
```

qt_catalan_number(*n*)

Returns the q,t-Catalan number.

EXAMPLES:

```
sage: import sage.combinat.q_analogues as q_analogues
sage: q_analogues.qt_catalan_number(1)
1
sage: q_analogues.qt_catalan_number(2)
q + t
sage: q_analogues.qt_catalan_number(3)
q^3 + q^2*t + q*t^2 + t^3 + q*t
sage: q_analogues.qt_catalan_number(4)
q^6 + q^5*t + q^4*t^2 + q^3*t^3 + q^2*t^4 + q*t^5 + t^6 + q^4*t + q^3*t^2 + q^2*t^3 + q*t^4 + q^
```

17.25 Ordered Set Partitions

An ordered set partition p of a set s is a partition of s, into subsets called parts and represented as a list of sets. By extension, an ordered set partition of a nonnegative integer n is the set partition of the integers from 1 to n. The number of ordered set partitions of n is called the n-th ordered Bell number.

There is a natural integer composition associated with an ordered set partition, that is the sequence of sizes of all its parts in order.

EXAMPLES: There are 13 ordered set partitions of {1,2,3}.

```
sage: OrderedSetPartitions(3).cardinality()
13
```

Here is the list of them:

```
sage: OrderedSetPartitions(3).list() #random due to the sets
[[{1}, {2}, {3}],
 [{1}, {3}, {2}],
 [{2}, {1}, {3}],
```

```
 [{3}, {1}, {2}],
 [{2}, {3}, {1}],
 [{3}, {2}, {1}],
 [{1}, {2, 3}],
 [{2}, {1, 3}],
 [{3}, {1, 2}],
 [{1, 2}, {3}],
 [{1, 3}, {2}],
 [{2, 3}, {1}],
 [{1, 2, 3}]]
```

There are 12 ordered set partitions of {1,2,3,4} whose underlying composition is [1,2,1].

```
sage: OrderedSetPartitions(4,[1,2,1]).list() #random due to the sets
[[{1}, {2, 3}, {4}],
 [{1}, {2, 4}, {3}],
 [{1}, {3, 4}, {2}],
 [{2}, {1, 3}, {4}],
 [{2}, {1, 4}, {3}],
 [{3}, {1, 2}, {4}],
 [{4}, {1, 2}, {3}],
 [{3}, {1, 4}, {2}],
 [{4}, {1, 3}, {2}],
 [{2}, {3, 4}, {1}],
 [{3}, {2, 4}, {1}],
 [{4}, {2, 3}, {1}]]
```

AUTHORS:

- Mike Hansen
- MuPAD-Combinat developers (for algorithms and design inspiration)

OrderedSetPartitions(*s, c=None*)

Returns the combinatorial class of ordered set partitions of s.

EXAMPLES:

```
sage: OS = OrderedSetPartitions([1,2,3,4]); OS
Ordered set partitions of {1, 2, 3, 4}
sage: OS.cardinality()
75
sage: OS.first()
[{1}, {2}, {3}, {4}]
sage: OS.last()
[{1, 2, 3, 4}]
sage: OS.random_element()
[{3}, {1}, {2}, {4}]
```

```
sage: OS = OrderedSetPartitions([1,2,3,4], [2,2]); OS
Ordered set partitions of {1, 2, 3, 4} into parts of size [2, 2]
sage: OS.cardinality()
6
sage: OS.first()
[{1, 2}, {3, 4}]
sage: OS.last()
[{3, 4}, {1, 2}]
```

```
sage: OS.list()
[[{1, 2}, {3, 4}],
 [{1, 3}, {2, 4}],
 [{1, 4}, {2, 3}],
 [{2, 3}, {1, 4}],
 [{2, 4}, {1, 3}],
 [{3, 4}, {1, 2}]]

sage: OS = OrderedSetPartitions("cat"); OS
Ordered set partitions of ['c', 'a', 't']
sage: OS.list()
[[{'a'}, {'c'}, {'t'}],
 [{'a'}, {'t'}, {'c'}],
 [{'c'}, {'a'}, {'t'}],
 [{'t'}, {'a'}, {'c'}],
 [{'c'}, {'t'}, {'a'}],
 [{'t'}, {'c'}, {'a'}],
 [{'a'}, {'c', 't'}],
 [{'c'}, {'a', 't'}],
 [{'t'}, {'a', 'c'}],
 [{'a', 'c'}, {'t'}],
 [{'a', 't'}, {'c'}],
 [{'c', 't'}, {'a'}],
 [{'a', 'c', 't'}]]
```

class **OrderedSetPartitions_s**(*s*)

cardinality()

EXAMPLES:

```
sage: OrderedSetPartitions(0).cardinality()
1
sage: OrderedSetPartitions(1).cardinality()
1
sage: OrderedSetPartitions(2).cardinality()
3
sage: OrderedSetPartitions(3).cardinality()
13
sage: OrderedSetPartitions([1,2,3]).cardinality()
13
sage: OrderedSetPartitions(4).cardinality()
75
sage: OrderedSetPartitions(5).cardinality()
541
```

class **OrderedSetPartitions_scomp**(*s*, *comp*)

cardinality()

EXAMPLES:

```
sage: OrderedSetPartitions(5,[2,3]).cardinality()
10
sage: OrderedSetPartitions(0, []).cardinality()
1
sage: OrderedSetPartitions(0, [0]).cardinality()
1
sage: OrderedSetPartitions(0, [0,0]).cardinality()
```

```
1
sage: OrderedSetPartitions(5, [2,0,3]).cardinality()
10
```

class **OrderedSetPartitions_sn**(*s*, *n*)

cardinality()
EXAMPLES:

```
sage: OrderedSetPartitions(4,2).cardinality()
14
sage: OrderedSetPartitions(4,1).cardinality()
1
```

17.26 Set Partitions

A set partition s of a set set is a partition of set, into subsets called parts and represented as a set of sets. By extension, a set partition of a nonnegative integer n is the set partition of the integers from 1 to n. The number of set partitions of n is called the n-th Bell number.

There is a natural integer partition associated with a set partition, that is the non-decreasing sequence of sizes of all its parts.

There is a classical lattice associated with all set partitions of n. The infimum of two set partitions is the set partition obtained by intersecting all the parts of both set partitions. The supremum is obtained by transitive closure of the relation i related to j iff they are in the same part in at least one of the set partitions.

EXAMPLES: There are 5 set partitions of the set 1,2,3.

```
sage: SetPartitions(3).cardinality()
5
```

Here is the list of them

```
sage: SetPartitions(3).list() #random due to the sets
[{{1, 2, 3}}, {{2, 3}, {1}}, {{1, 3}, {2}}, {{1, 2}, {3}}, {{2}, {3}, {1}}]
```

There are 6 set partitions of 1,2,3,4 whose underlying partition is [2, 1, 1]:

```
sage: SetPartitions(4, [2,1,1]).list() #random due to the sets
[{{3, 4}, {2}, {1}},
 {{2, 4}, {3}, {1}},
 {{4}, {2, 3}, {1}},
 {{1, 4}, {2}, {3}},
 {{1, 3}, {4}, {2}},
 {{1, 2}, {4}, {3}}]
```

AUTHORS:

- Mike Hansen
- MuPAD-Combinat developers (for algorithms and design inspiration).

SetPartitions(*s, part=None*)

An unordered partition of a set S is a set of pairwise disjoint nonempty subsets with union S and is represented by a sorted list of such subsets.

SetPartitions(s) returns the class of all set partitions of the set s, which can be a set or a string; if a string, each character is considered an element.

SetPartitions(n), where n is an integer, returns the class of all set partitions of the set [1, 2,..., n].

You may specify a second argument k. If k is an integer, SetPartitions returns the class of set partitions into k parts; if it is an integer partition, SetPartitions returns the class of set partitions whose block sizes correspond to that integer partition.

The Bell number B_n, named in honor of Eric Temple Bell, is the number of different partitions of a set with n elements.

EXAMPLES:

```
sage: S = [1,2,3,4]
sage: SetPartitions(S,2)
Set partitions of [1, 2, 3, 4] with 2 parts
sage: SetPartitions([1,2,3,4], [3,1]).list()
[{{2, 3, 4}, {1}}, {{1, 3, 4}, {2}}, {{3}, {1, 2, 4}}, {{4}, {1, 2, 3}}]
sage: SetPartitions(7, [3,3,1]).cardinality()
70
```

In strings, repeated letters are considered distinct:

```
sage: SetPartitions('aabcd').cardinality()
52
sage: SetPartitions('abcde').cardinality()
52
```

REFERENCES:

- http://en.wikipedia.org/wiki/Partition_of_a_set

class SetPartitions_set(*set*)

cardinality()

EXAMPLES:

```
sage: SetPartitions(4).cardinality()
15
sage: bell_number(4)
15
```

class SetPartitions_setn(*set, n*)

cardinality()

The Stirling number of the second kind is the number of partitions of a set of size n into k blocks.

EXAMPLES:

```
sage: SetPartitions(5, 3).cardinality()
25
sage: stirling_number2(5,3)
25
```

class SetPartitions_setparts(*set, parts*)

cardinality()

Returns the number of set partitions of set. This number is given by the n-th Bell number where n is the number of elements in the set.

If a partition or partition length is specified, then count will generate all of the set partitions.

EXAMPLES:

```
sage: SetPartitions([1,2,3,4]).cardinality()
15
sage: SetPartitions(3).cardinality()
5
sage: SetPartitions(3,2).cardinality()
3
sage: SetPartitions([]).cardinality()
1
```

object_class()

A finite enumerated set.

inf(*s*, *t*)

Returns the infimum of the two set partitions s and t.

EXAMPLES:

```
sage: sp1 = Set([Set([2,3,4]),Set([1])])
sage: sp2 = Set([Set([1,3]), Set([2,4])])
sage: s = Set([ Set([2,4]), Set([3]), Set([1])]) #{{2, 4}, {3}, {1}}
sage: sage.combinat.set_partition.inf(sp1, sp2) == s
True
```

less(*s*, *t*)

Returns True if s < t otherwise it returns False.

EXAMPLES:

```
sage: z = SetPartitions(3).list()
sage: sage.combinat.set_partition.less(z[0], z[1])
False
sage: sage.combinat.set_partition.less(z[4], z[1])
True
sage: sage.combinat.set_partition.less(z[4], z[0])
True
sage: sage.combinat.set_partition.less(z[3], z[0])
True
sage: sage.combinat.set_partition.less(z[2], z[0])
True
sage: sage.combinat.set_partition.less(z[1], z[0])
True
sage: sage.combinat.set_partition.less(z[0], z[0])
False
```

standard_form(*sp*)

Returns the set partition as a list of lists.

EXAMPLES:

```
sage: map(sage.combinat.set_partition.standard_form, SetPartitions(4, [2,2]))
[[[3, 4], [1, 2]], [[2, 4], [1, 3]], [[2, 3], [1, 4]]]
```

sup(*s*, *t*)

Returns the supremum of the two set partitions s and t.

EXAMPLES:

```
sage: sp1 = Set([Set([2,3,4]),Set([1])])
sage: sp2 = Set([Set([1,3]), Set([2,4])])
sage: s = Set([ Set([1,2,3,4]) ])
sage: sage.combinat.set_partition.sup(sp1, sp2) == s
True
```

17.27 Skew Partitions

A skew partition skp of size n is a pair of partitions $[p_1, p_2]$ where p_1 is a partition of the integer n_1, p_2 is a partition of the integer n_2, p_2 is an inner partition of p_1, and $n = n_1 - n_2$. We say that p_1 and p_2 are respectively the *inner* and *outer* partitions of skp.

A skew partition can be depicted by a diagram made of rows of boxes, in the same way as a partition. Only the boxes of the outer partition p_1 which are not in the inner partition p_2 appear in the picture. For example, this is the diagram of the skew partition [[5,4,3,1],[3,3,1]].

```
sage: print SkewPartition([[5,4,3,1],[3,3,1]]).diagram()
   ##
   #
 ##
#
```

A skew partition can be *connected*, which can easily be described in graphic terms: for each pair of consecutive rows, there are at least two boxes (one in each row) which have a common edge. This is the diagram of the connected skew partition [[5,4,3,1],[3,1]]:

```
sage: print SkewPartition([[5,4,3,1],[3,1]]).diagram()
   ##
 ###
###
#
sage: SkewPartition([[5,4,3,1],[3,1]]).is_connected()
True
```

The first example of a skew partition is not a connected one.

Applying a reflection with respect to the main diagonal yields the diagram of the *conjugate skew partition*, here [[4,3,3,2,1],[3,3,2]]:

```
sage: SkewPartition([[5,4,3,1],[3,3,1]]).conjugate()
[[4, 3, 3, 2, 1], [3, 2, 2]]
sage: print SkewPartition([[5,4,3,1],[3,3,1]]).conjugate().diagram()
   #
  #
  #
##
#
```

The *outer corners* of a skew partition are the corners of its outer partition. The *inner corners* are the internal corners of the outer partition when the inner partition is taken off. Shown below are the coordinates of the inner and outer corners.

```
sage: SkewPartition([[5,4,3,1],[3,3,1]]).outer_corners()
[[0, 4], [1, 3], [2, 2], [3, 0]]
sage: SkewPartition([[5,4,3,1],[3,3,1]]).inner_corners()
[[0, 3], [2, 1], [3, 0]]
```

EXAMPLES: There are 9 skew partitions of size 3.

```
sage: SkewPartitions(3).cardinality()
9
sage: SkewPartitions(3).list()
[[[1, 1, 1], []],
 [[2, 2, 1], [1, 1]],
 [[2, 1, 1], [1]],
 [[3, 2, 1], [2, 1]],
 [[2, 2], [1]],
 [[3, 2], [2]],
 [[2, 1], []],
 [[3, 1], [1]],
 [[3], []]]
```

There are 4 connected skew partitions of size 3.

```
sage: SkewPartitions(3, overlap=1).cardinality()
4
sage: SkewPartitions(3, overlap=1).list()
[[[1, 1, 1], []], [[2, 2], [1]], [[2, 1], []], [[3], []]]
```

This is the conjugate of the skew partition [[4,3,1],[2]]

```
sage: SkewPartition([[4,3,1],[2]]).conjugate()
[[3, 2, 2, 1], [1, 1]]
```

Geometrically, we just applied a reflection with respect to the main diagonal on the diagram of the partition. Of course, this operation is an involution:

```
sage: SkewPartition([[4,3,1],[2]]).conjugate().conjugate()
[[4, 3, 1], [2]]
```

The jacobi_trudy() method computes the Jacobi-Trudi matrix. See Macdonald I.-G., (1995), "Symmetric Functions and Hall Polynomials", Oxford Science Publication for a definition and discussion.

```
sage: SkewPartition([[4,3,1],[2]]).jacobi_trudi()
[h[2]  h[]    0]
[h[5] h[3]  h[]]
[h[6] h[4] h[1]]
```

This example shows how to compute the corners of a skew partition.

```
sage: SkewPartition([[4,3,1],[2]]).inner_corners()
[[0, 2], [1, 0]]
sage: SkewPartition([[4,3,1],[2]]).outer_corners()
[[0, 3], [1, 2], [2, 0]]
```

SkewPartition(*skp*)

Returns the skew partition object corresponding to skp.

EXAMPLES:

```
sage: skp = SkewPartition([[3,2,1],[2,1]]); skp
[[3, 2, 1], [2, 1]]
sage: skp.inner()
[2, 1]
sage: skp.outer()
[3, 2, 1]
```

class **SkewPartition_class**(*skp*)

column_lengths()

Returns the column lengths of the skew partition.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).column_lengths()
[1, 2, 1]
sage: SkewPartition([[5,2,2,2],[2,1]]).column_lengths()
[2, 3, 1, 1, 1]
```

columns_intersection_set()

Returns the set of boxes in the lines of lambda which intersect the skew partition.

EXAMPLES:

```
sage: skp = SkewPartition([[3,2,1],[2,1]])
sage: boxes = Set([ (0,0), (0, 1), (0,2), (1, 0), (1, 1), (2, 0)])
sage: skp.columns_intersection_set() == boxes
True
```

conjugate()

Returns the conjugate of the skew partition skp.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2]]).conjugate()
[[3, 2, 1], [1, 1]]
```

diagram()

Return the Ferrers diagram of self.

EXAMPLES:

```
sage: print SkewPartition([[5,4,3,1],[3,3,1]]).ferrers_diagram()
   ##
   #
 ##
#
sage: print SkewPartition([[5,4,3,1],[3,1]]).diagram()
   ##
 ###
###
#
```

ferrers_diagram()

Return the Ferrers diagram of self.

EXAMPLES:

```
sage: print SkewPartition([[5,4,3,1],[3,3,1]]).ferrers_diagram()
   ##
    #
 ##
#
sage: print SkewPartition([[5,4,3,1],[3,1]]).diagram()
   ##
 ###
###
#
```

inner()

Returns the inner partition of the skew partition.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).inner()
[1, 1]
```

inner_corners()

Returns a list of the inner corners of the skew partition skp.

EXAMPLES:

```
sage: SkewPartition([[4, 3, 1], [2]]).inner_corners()
[[0, 2], [1, 0]]
```

is_connected()

Returns True if self is a connected skew partition.

A skew partition is said to be *connected* if for each pair of consecutive rows, there are at least two boxes (one in each row) which have a common edge.

EXAMPLES:

```
sage: SkewPartition([[5,4,3,1],[3,3,1]]).is_connected()
False
sage: SkewPartition([[5,4,3,1],[3,1]]).is_connected()
True
```

is_overlap(*n*)

Returns True if n = self.overlap()

See Also:

overlap()

EXAMPLES:

```
sage: SkewPartition([[5,4,3,1],[3,1]]).is_overlap(1)
True
```

jacobi_trudi()

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2,1]]).jacobi_trudi()
[h[1]    0    0]
[h[3] h[1]    0]
[h[5] h[3] h[1]]
sage: SkewPartition([[4,3,2],[2,1]]).jacobi_trudi()
[h[2]  h[]    0]
[h[4] h[2]  h[]]
[h[6] h[4] h[2]]
```

k_conjugate(*k*)

Returns the k-conjugate of the skew partition.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2,1]]).k_conjugate(3)
[[2, 1, 1, 1, 1], [2, 1]]
sage: SkewPartition([[3,2,1],[2,1]]).k_conjugate(4)
[[2, 2, 1, 1], [2, 1]]
sage: SkewPartition([[3,2,1],[2,1]]).k_conjugate(5)
[[3, 2, 1], [2, 1]]
```

outer()

Returns the outer partition of the skew partition.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).outer()
[3, 2, 1]
```

outer_corners()

Returns a list of the outer corners of the skew partition skp.

EXAMPLES:

```
sage: SkewPartition([[4, 3, 1], [2]]).outer_corners()
[[0, 3], [1, 2], [2, 0]]
```

overlap()

Returns the overlap of self.

The overlap of two consecutive rows in a skew partition is the number of pairs of boxes (one in each row) that share a common edge. This number can be positive, zero, or negative.

The overlap of a skew partition is the minimum of the overlap of the consecutive rows, or infinity in the case of at most one row. If the overlap is positive, then the skew partition is called *connected*.

EXAMPLES:

```
sage: SkewPartition([[],[]]).overlap()
+Infinity
sage: SkewPartition([[1],[]]).overlap()
+Infinity
sage: SkewPartition([[10],[]]).overlap()
+Infinity
sage: SkewPartition([[10],[2]]).overlap()
+Infinity
sage: SkewPartition([[10,1],[2]]).overlap()
-1
sage: SkewPartition([[10,10],[1]]).overlap()
9
```

pieri_macdonald_coeffs()

Computation of the coefficients which appear in the Pieri formula for Macdonald polynomails given in his book (Chapter 6.6 formula 6.24(ii))

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[2,1]]).pieri_macdonald_coeffs()
1
sage: SkewPartition([[3,2,1],[2,2]]).pieri_macdonald_coeffs()
(q^2*t^3 - q^2*t - t^2 + 1)/(q^2*t^3 - q*t^2 - q*t + 1)
sage: SkewPartition([[3,2,1],[2,2,1]]).pieri_macdonald_coeffs()
(q^6*t^8 - q^6*t^6 - q^4*t^7 - q^5*t^5 + q^4*t^5 - q^3*t^6 + q^5*t^3 + 2*q^3*t^4 + q*t^5 - q
sage: SkewPartition([[3,3,2,2],[3,2,2,1]]).pieri_macdonald_coeffs()
(q^5*t^6 - q^5*t^5 + q^4*t^6 - q^4*t^5 - q^4*t^3 + q^4*t^2 - q^3*t^3 - q^2*t^4 + q^3*t^2 + q
```

r_quotient(k)

The quotient map extended to skew partitions.

EXAMPLES:

```
sage: SkewPartition([[3, 3, 2, 1], [2, 1]]).r_quotient(2)
[[[3], []], [[], []]]
```

row_lengths()

Returns the row lengths of the skew partition.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).row_lengths()
[2, 1, 1]
```

rows_intersection_set()

Returns the set of boxes in the lines of lambda which intersect the skew partition.

EXAMPLES:

```
sage: skp = SkewPartition([[3,2,1],[2,1]])
sage: boxes = Set([ (0,0), (0, 1), (0,2), (1, 0), (1, 1), (2, 0)])
sage: skp.rows_intersection_set() == boxes
True
```

size()

Returns the size of the skew partition.

EXAMPLES:

```
sage: SkewPartition([[3,2,1],[1,1]]).size()
4
```

to_dag()

Returns a directed acyclic graph corresponding to the skew partition.

EXAMPLES:

```
sage: dag = SkewPartition([[3, 2, 1], [1, 1]]).to_dag()
sage: dag.edges()
[('0,1', '0,2', None), ('0,1', '1,1', None)]
sage: dag.vertices()
['0,1', '0,2', '1,1', '2,0']
```

to_list()

Returns self as a list of lists.

EXAMPLES:

```
sage: s = SkewPartition([[4,3,1],[2]])
sage: s.to_list()
[[4, 3, 1], [2]]
sage: type(s.to_list())
<type 'list'>
```

SkewPartitions(*n=None, row_lengths=None, overlap=0*)

Returns the combinatorial class of skew partitions.

EXAMPLES:

```
sage: SkewPartitions(4)
Skew partitions of 4
sage: SkewPartitions(4).cardinality()
28
sage: SkewPartitions(row_lengths=[2,1,2])
Skew partitions with row lengths [2, 1, 2]
sage: SkewPartitions(4, overlap=2)
```

```
Skew partitions of 4 with overlap of 2
sage: SkewPartitions(4, overlap=2).list()
[[[2, 2], []], [[4], []]]
```

class SkewPartitions_all()

list()
TESTS:

```
sage: SkewPartitions().list()
...
NotImplementedError
```

object_class()

class SkewPartitions_n(*n*, *overlap=0*)

cardinality()
Returns the number of skew partitions of the integer n.
EXAMPLES:

```
sage: SkewPartitions(0).cardinality()
1
sage: SkewPartitions(4).cardinality()
28
sage: SkewPartitions(5).cardinality()
87
sage: SkewPartitions(4, overlap=1).cardinality()
9
sage: SkewPartitions(5, overlap=1).cardinality()
20
sage: s = SkewPartitions(5, overlap=-1)
sage: s.cardinality() == len(s.list())
True
```

list()
Returns a list of the skew partitions of n.
EXAMPLES:

```
sage: SkewPartitions(3).list()
[[[1, 1, 1], []],
 [[2, 2, 1], [1, 1]],
 [[2, 1, 1], [1]],
 [[3, 2, 1], [2, 1]],
 [[2, 2], [1]],
 [[3, 2], [2]],
 [[2, 1], []],
 [[3, 1], [1]],
 [[3], []]]
sage: SkewPartitions(3, overlap=0).list()
[[[1, 1, 1], []],
 [[2, 2, 1], [1, 1]],
 [[2, 1, 1], [1]],
 [[3, 2, 1], [2, 1]],
 [[2, 2], [1]],
 [[3, 2], [2]],
 [[2, 1], []],
 [[3, 1], [1]],
```

```
 [[3], []]]
sage: SkewPartitions(3, overlap=1).list()
[[[1, 1, 1], []], [[2, 2], [1]], [[2, 1], []], [[3], []]]
sage: SkewPartitions(3, overlap=2).list()
[[[3], []]]
sage: SkewPartitions(3, overlap=3).list()
[[[3], []]]
sage: SkewPartitions(3, overlap=4).list()
[]
```

object_class()

class SkewPartitions_rowlengths(*co, overlap=0*)

The combinatorial class of all skew partitions with given row lengths.

list()

Returns a list of all the skew partitions that have row lengths given by the composition self.co.

EXAMPLES:

```
sage: SkewPartitions(row_lengths=[2,2]).list()
[[[2, 2], []], [[3, 2], [1]], [[4, 2], [2]]]
sage: SkewPartitions(row_lengths=[2,2], overlap=1).list()
[[[2, 2], []], [[3, 2], [1]]]
```

object_class()

row_lengths_aux(*skp*)

EXAMPLES:

```
sage: from sage.combinat.skew_partition import row_lengths_aux
sage: row_lengths_aux([[5,4,3,1],[3,3,1]])
[2, 1, 2]
sage: row_lengths_aux([[5,4,3,1],[3,1]])
[2, 3]
```

17.28 Subsets

The combinatorial class of the subsets of a finite set. The set can be given as a list or a Set or else as an integer n which encodes the set $\{1, 2, ..., n\}$. See the `Subsets` for more informations and examples.

AUTHORS:

- Mike Hansen: initial version
- Florent Hivert (2009/02/06): doc improvements + new methods

class SubMultiset_s(*s*)

The combinatorial class of the sub multisets of s.

EXAMPLES:

```
sage: S = Subsets([1,2,2,3], submultiset=True)
sage: S._s
[1, 2, 2, 3]
```

The positions of the unique elements in s are stored in:

```
sage: S._indices
[0, 1, 3]
```

and their multiplicities in:

```
sage: S._multiplicities
[1, 2, 1]
sage: Subsets([1,2,3,3], submultiset=True).cardinality()
12
sage: S == loads(dumps(S))
True
```

class SubMultiset_sk(*s, k*)

The combinatorial class of the subsets of size k of a multiset s. Note that each subset is represented by a list of the elements rather than a set since we can have multiplicities (no multiset data structure yet in sage).

EXAMPLES:

```
sage: S = Subsets([1,2,3,3],2, submultiset=True)
sage: S._k
2
sage: S.cardinality()
4
sage: S.first()
[1, 2]
sage: S.last()
[3, 3]
sage: [sub for sub in S]
[[1, 2], [1, 3], [2, 3], [3, 3]]
sage: S == loads(dumps(S))
True
```

Subsets(*s, k=None, submultiset=False*)

Returns the combinatorial class of the subsets of the finite set s. The set can be given as a list, Set or any iterable convertible to a set. It can alternatively be given a non-negative integer n which encode the set $\{1, 2, \ldots, n\}$ (i.e. the Sage `range(1,s+1)`).

A second optional parameter k can be given. In this case, Subsets returns the combinatorial class of subsets of s of size k.

Finally the option `submultiset` allows one to deal with sets with repeated elements usually called multisets.

EXAMPLES:

```
sage: S = Subsets([1, 2, 3]); S
Subsets of {1, 2, 3}
sage: S.cardinality()
8
sage: S.first()
{}
sage: S.last()
{1, 2, 3}
sage: S.random_element()
{2}
sage: S.list()
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

Here is the same example where the set is given as an integer:

```
sage: S = Subsets(3)
sage: S.list()
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

We demonstrate various the effect of the various options:

```
sage: S = Subsets(3, 2); S
Subsets of {1, 2, 3} of size 2
sage: S.list()
[{1, 2}, {1, 3}, {2, 3}]

sage: S = Subsets([1, 2, 2], submultiset=True); S
SubMultiset of [1, 2, 2]
sage: S.list()
[[], [1], [2], [1, 2], [2, 2], [1, 2, 2]]

sage: S = Subsets([1, 2, 2, 3], 3, submultiset=True); S
SubMultiset of [1, 2, 2, 3] of size 3
sage: S.list()
[[1, 2, 2], [1, 2, 3], [2, 2, 3]]
```

class Subsets_s (s)

cardinality()

Returns the number of subsets of the set s.

This is given by $2^{|s|}$.

EXAMPLES:

```
sage: Subsets(Set([1,2,3])).cardinality()
8
sage: Subsets([1,2,3,3]).cardinality()
8
sage: Subsets(3).cardinality()
8
```

first()

Returns the first subset of s. Since we aren't restricted to subsets of a certain size, this is always the empty set.

EXAMPLES:

```
sage: Subsets([1,2,3]).first()
{}
sage: Subsets(3).first()
{}
```

last()

Returns the last subset of s. Since we aren't restricted to subsets of a certain size, this is always the set s itself.

EXAMPLES:

```
sage: Subsets([1,2,3]).last()
{1, 2, 3}
sage: Subsets(3).last()
{1, 2, 3}
```

random_element()

Returns a random element of the class of subsets of s (in other words, a random subset of s).

EXAMPLES:

```
sage: Subsets(3).random_element()
{2}
sage: Subsets([4,5,6]).random_element()
{5}
```

rank(*sub*)

Returns the rank of sub as a subset of s.

EXAMPLES:

```
sage: Subsets(3).rank([])
0
sage: Subsets(3).rank([1,2])
4
sage: Subsets(3).rank([1,2,3])
7
sage: Subsets(3).rank([2,3,4]) == None
True
```

unrank(*r*)

Returns the subset of s that has rank k.

EXAMPLES:

```
sage: Subsets(3).unrank(0)
{}
sage: Subsets([2,4,5]).unrank(1)
{2}
sage: s = Subsets([2,4,5])
```

class **Subsets_sk**(*s*, *k*)

cardinality()

EXAMPLES:

```
sage: Subsets(Set([1,2,3]), 2).cardinality()
3
sage: Subsets([1,2,3,3], 2).cardinality()
3
sage: Subsets([1,2,3], 1).cardinality()
3
sage: Subsets([1,2,3], 3).cardinality()
1
sage: Subsets([1,2,3], 0).cardinality()
1
sage: Subsets([1,2,3], 4).cardinality()
0
sage: Subsets(3,2).cardinality()
3
sage: Subsets(3,4).cardinality()
0
```

first()

Returns the first subset of s of size k.

EXAMPLES:

```
sage: Subsets(Set([1,2,3]), 2).first()
{1, 2}
sage: Subsets([1,2,3,3], 2).first()
{1, 2}
```

```
sage: Subsets(3,2).first()
{1, 2}
sage: Subsets(3,4).first()
```

last()
Returns the last subset of s of size k.
EXAMPLES:

```
sage: Subsets(Set([1,2,3]), 2).last()
{2, 3}
sage: Subsets([1,2,3,3], 2).last()
{2, 3}
sage: Subsets(3,2).last()
{2, 3}
sage: Subsets(3,4).last()
```

random_element()
Returns a random element of the class of subsets of s of size k (in other words, a random subset of s of size k).
EXAMPLES:

```
sage: Subsets(3, 2).random_element()
{1, 2}
sage: Subsets(3,4).random_element() is None
True
```

rank(*sub*)
Returns the rank of sub as a subset of s of size k.
EXAMPLES:

```
sage: Subsets(3,2).rank([1,2])
0
sage: Subsets([2,3,4],2).rank([3,4])
2
sage: Subsets([2,3,4],2).rank([2])
sage: Subsets([2,3,4],4).rank([2,3,4,5])
```

unrank(*r*)
Returns the subset of s that has rank k.
EXAMPLES:

```
sage: Subsets(3,2).unrank(0)
{1, 2}
sage: Subsets([2,4,5],2).unrank(0)
{2, 4}
```

17.29 Subwords

A subword of a word w is a word obtained by deleting the letters at some (non necessarily adjacent) positions in w. It is not to be confused with the notion of factor where one keeps adjacent positions in w. Sometimes it is useful to allow repeated uses of the same letter of w in a "generalized" subword. We call this a subword with repetitions.

For example:

- "bnjr" is a subword of the word "bonjour" but not a factor;
- "njo" is both a factor and a subword of the word "bonjour";

- “nr” is a subword of “bonjour”;
- “rn” is not a subword of “bonjour”;
- “nnu” is not a subword of “bonjour”;
- “nnu” is a subword with repetitions of “bonjour”;

A word can be given either as a string or as a list.

AUTHORS:

- Mike Hansen: initial version
- Florent Hivert (2009/02/06): doc improvements + new methods + bug fixes

Subwords(*w, k=None*)

Returns the combinatorial class of subwords of w. The word w can be given by either a string or a list.

If k is specified, then it returns the combinatorial class of subwords of w of length k.

EXAMPLES:

```
sage: S = Subwords(['a','b','c']); S
Subwords of ['a', 'b', 'c']
sage: S.first()
[]
sage: S.last()
['a', 'b', 'c']
sage: S.list()
[[], ['a'], ['b'], ['c'], ['a', 'b'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c']]
```

```
sage: S = Subwords(['a','b','c'], 2); S
Subwords of ['a', 'b', 'c'] of length 2
sage: S.list()
[['a', 'b'], ['a', 'c'], ['b', 'c']]
```

class Subwords_w(*w*)

cardinality()

EXAMPLES:

```
sage: Subwords([1,2,3]).cardinality()
8
```

first()

EXAMPLES:

```
sage: Subwords([1,2,3]).first()
[]
```

last()

EXAMPLES:

```
sage: Subwords([1,2,3]).last()
[1, 2, 3]
```

class Subwords_wk(*w, k*)

cardinality()
Returns the number of subwords of w of length k.
EXAMPLES:

```
sage: Subwords([1,2,3], 2).cardinality()
3
```

first()
EXAMPLES:

```
sage: Subwords([1,2,3],2).first()
[1, 2]
sage: Subwords([1,2,3],0).first()
[]
```

last()
EXAMPLES:

```
sage: Subwords([1,2,3],2).last()
[2, 3]
```

smallest_positions(*word, subword, pos=0*)
Returns the smallest positions for which subword appears as a subword of word. If pos is specified, then it returns the positions of the first appearance of subword starting at pos.

If subword is not found in word, then it returns False.

EXAMPLES:

```
sage: sage.combinat.subword.smallest_positions([1,2,3,4], [2,4])
[1, 3]
sage: sage.combinat.subword.smallest_positions([1,2,3,4,4], [2,4])
[1, 3]
sage: sage.combinat.subword.smallest_positions([1,2,3,3,4,4], [3,4])
[2, 4]
sage: sage.combinat.subword.smallest_positions([1,2,3,3,4,4], [3,4],2)
[2, 4]
sage: sage.combinat.subword.smallest_positions([1,2,3,3,4,4], [3,4],3)
[3, 4]
sage: sage.combinat.subword.smallest_positions([1,2,3,4], [2,3])
[1, 2]
sage: sage.combinat.subword.smallest_positions([1,2,3,4], [5,5])
False
sage: sage.combinat.subword.smallest_positions([1,3,3,4,5],[3,5])
[1, 4]
sage: sage.combinat.subword.smallest_positions([1,3,3,5,4,5,3,5],[3,5,3])
[1, 3, 6]
sage: sage.combinat.subword.smallest_positions([1,3,3,5,4,5,3,5],[3,5,3],2)
[2, 3, 6]
sage: sage.combinat.subword.smallest_positions([1,2,3,4,3,4,4],[2,3,3,1])
False
sage: sage.combinat.subword.smallest_positions([1,3,3,5,4,5,3,5],[3,5,3],3)
False
```

TEST:

```
sage: w = ["a", "b", "c", "d"]; ww = ["b", "d"]
sage: x = sage.combinat.subword.smallest_positions(w, ww); ww # Trac #5534
['b', 'd']
```

17.30 Tuples

Tuples(*S*, *k*)

Returns the combinatorial class of ordered tuples of S of length k.

An ordered tuple of length k of set is an ordered selection with repetition and is represented by a list of length k containing elements of set.

EXAMPLES:

```
sage: S = [1,2]
sage: Tuples(S,3).list()
[[1, 1, 1], [2, 1, 1], [1, 2, 1], [2, 2, 1], [1, 1, 2], [2, 1, 2], [1, 2, 2], [2, 2, 2]]
sage: mset = ["s","t","e","i","n"]
sage: Tuples(mset,2).list()
[['s', 's'], ['t', 's'], ['e', 's'], ['i', 's'], ['n', 's'], ['s', 't'], ['t', 't'],
 ['e', 't'], ['i', 't'], ['n', 't'], ['s', 'e'], ['t', 'e'], ['e', 'e'], ['i', 'e'],
 ['n', 'e'], ['s', 'i'], ['t', 'i'], ['e', 'i'], ['i', 'i'], ['n', 'i'], ['s', 'n'],
 ['t', 'n'], ['e', 'n'], ['i', 'n'], ['n', 'n']]


sage: K.<a> = GF(4, 'a')
sage: mset = [x for x in K if x!=0]
sage: Tuples(mset,2).list()
[[a, a], [a + 1, a], [1, a], [a, a + 1], [a + 1, a + 1], [1, a + 1], [a, 1], [a + 1, 1], [1, 1]]
```

class Tuples_sk(*S*, *k*)

cardinality()

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: Tuples(S,2).cardinality()
25
sage: S = [1,1,2,3,4,5]
sage: Tuples(S,2).cardinality()
25
```

UnorderedTuples(*S*, *k*)

Returns the combinatorial class of unordered tuples of S of length k.

An unordered tuple of length k of set is a unordered selection with repetitions of set and is represented by a sorted list of length k containing elements from set.

EXAMPLES:

```
sage: S = [1,2]
sage: UnorderedTuples(S,3).list()
[[1, 1, 1], [1, 1, 2], [1, 2, 2], [2, 2, 2]]
sage: UnorderedTuples(["a","b","c"],2).list()
[['a', 'a'], ['a', 'b'], ['a', 'c'], ['b', 'b'], ['b', 'c'], ['c', 'c']]
```

class UnorderedTuples_sk(*S*, *k*)

cardinality()

EXAMPLES:

```
sage: S = [1,2,3,4,5]
sage: UnorderedTuples(S,2).cardinality()
15
```

list()
EXAMPLES:

```
sage: S = [1,2]
sage: UnorderedTuples(S,3).list()
[[1, 1, 1], [1, 1, 2], [1, 2, 2], [2, 2, 2]]
sage: UnorderedTuples(["a","b","c"],2).list()
[['a', 'a'], ['a', 'b'], ['a', 'c'], ['b', 'b'], ['b', 'c'], ['c', 'c']]
```

17.31 Combinatorial Algebras

17.31.1 Free modules

class CombinatorialFreeModule(*R, cc, element_class=<class 'sage.combinat.free_module.CombinatorialFreeModuleElement'>*, *prefix='B'*)
EXAMPLES: We construct a free module whose basis is indexed by the letters a,b,c:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: F
Free module generated by ['a', 'b', 'c'] over Rational Field
```

Its basis is a family, indexed by a,b,c: Caveat in family: the order of the indices is not preserved

```
sage: e = F.basis()
```

```
sage: list(sorted(e.keys()))
['a', 'b', 'c']
sage: list(sorted(e))
[B['a'], B['b'], B['c']]
```

Let us construct some elements, and compute with them:

```
sage: e['a']
B['a']
sage: 2*e['a']
2*B['a']
sage: e['a'] + 3*e['b']
B['a'] + 3*B['b']
```

class CombinatorialFreeModuleElement(*M, x*)

coefficient(*m*)
EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: z = s([4]) - 2*s([2,1]) + s([1,1,1]) + s([1])
sage: z.coefficient([4])
1
sage: z.coefficient([2,1])
-2
```

coefficients()
Returns a list of the coefficients appearing on the basiselements in self.
EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f.coefficients()
[1, -3]

sage: s = SFASchur(QQ)
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.coefficients()
[1, 1, 1, 1]
```

is_zero()

Returns True if and only self == 0.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f.is_zero()
False
sage: F(0).is_zero()
True

sage: s = SFASchur(QQ)
sage: s([2,1]).is_zero()
False
sage: s(0).is_zero()
True
sage: (s([2,1]) - s([2,1])).is_zero()
True
```

length()

Returns the number of basis elements of self with nonzero coefficients.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f.length()
2

sage: s = SFASchur(QQ)
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.length()
4
```

map_coefficients(*f*)

Returns a new element of self.parent() obtained by applying the function f to all of the coefficients of self.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f.map_coefficients(lambda x: x+5)
6*B['a'] + 2*B['c']

sage: s = SFASchur(QQ)
sage: a = s([2,1])+2*s([3,2])
```

```
sage: a.map_coefficients(lambda x: x*2)
2*s[2, 1] + 4*s[3, 2]
```

map_mc(*f*)

Returns a new element of self.parent() obtained by applying the function f to a monomial coefficient (m,c) pair. f returns a (new_m, new_c) pair.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: f = lambda m,c: (m.conjugate(), 2*c)
sage: a = s([2,1]) + s([1,1,1])
sage: a.map_monomial(f)
2*s[2, 1] + 2*s[3]
```

map_monomial(*f*)

Returns a new element of self.parent() obtained by applying the function f to a monomial coefficient (m,c) pair. f returns a (new_m, new_c) pair.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: f = lambda m,c: (m.conjugate(), 2*c)
sage: a = s([2,1]) + s([1,1,1])
sage: a.map_monomial(f)
2*s[2, 1] + 2*s[3]
```

map_support(*f*)

Returns a new element of self.parent() obtained by applying the function f to all of the combinatorial objects indexing the basis elements.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: a = s([2,1])+2*s([3,2])
sage: a.map_support(lambda x: x.conjugate())
s[2, 1] + 2*s[2, 2, 1]
```

monomial_coefficients()

Return the internal dictionary which has the combinatorial objects indexing the basis as keys and their corresponding coefficients as values.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 3*B['c']
sage: d = f.monomial_coefficients()
sage: d['a']
1
sage: d['c']
3
```

```
sage: s = SFASchur(QQ)
sage: a = s([2,1])+2*s([3,2])
sage: d = a.monomial_coefficients()
sage: type(d)
<type 'dict'>
sage: d[ Partition([2,1]) ]
1
sage: d[ Partition([3,2]) ]
2
```

monomials()
EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 2*B['c']
sage: f.monomials()
[B['a'], B['c']]
```

support()
Returns a list of the combinatorial objects indexing the basis elements of self which non-zero coefficients.
EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f.support()
['a', 'c']

sage: s = SFASchur(QQ)
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.support()
[[1], [1, 1, 1], [2, 1], [4]]
```

terms()
EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] + 2*B['c']
sage: f.terms()
[B['a'], 2*B['c']]
```

to_vector()
Returns a vector version of self.
EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: B = F.basis()
sage: f = B['a'] - 3*B['c']
sage: f.to_vector()
(1, 0, -3)

sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: a = 2*QS3([1,2,3])+4*QS3([3,2,1])
sage: a.to_vector()
(2, 0, 0, 0, 0, 4)
```

class **CombinatorialFreeModuleInterface**(*R*, *element_class*)

basis()
Returns the basis of self.
EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a','b','c'])
sage: F.basis()
Finite family {'a': B['a'], 'c': B['c'], 'b': B['b']}
```

```
sage: QS3 = SymmetricGroupAlgebra(QQ,3)
sage: list(QS3.basis())
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

combinatorial_class()

Returns the combinatorial class that indexes the basis elements.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.combinatorial_class()
Combinatorial class with elements in ['a', 'b', 'c']

sage: s = SFASchur(QQ)
sage: s.combinatorial_class()
Partitions
```

dimension()

Returns the dimension of the combinatorial algebra (which is given by the number of elements in the associated combinatorial class).

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.dimension()
3

sage: s = SFASchur(QQ)
sage: s.dimension()
+Infinity
```

get_order()

Returns the order of the elements in the basis.

EXAMPLES:

```
sage: QS2 = SymmetricGroupAlgebra(QQ,2)
sage: QS2.get_order()
[[1, 2], [2, 1]]
```

prefix()

Returns the prefix used when displaying elements of self.

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.prefix()
'B'

sage: X = SchubertPolynomialRing(QQ)
sage: X.prefix()
'X'
```

set_order(*order*)

Sets the order of the elements of the combinatorial class.

If .set_order() has not been called, then the ordering is the one used in the generation of the elements of self's associated combinatorial class.

EXAMPLES:

```
sage: QS2 = SymmetricGroupAlgebra(QQ,2)
sage: b = list(QS2.basis().keys())
sage: b.reverse()
sage: QS2.set_order(b)
```

```
sage: QS2.get_order()
[[2, 1], [1, 2]]
```

sum(*operands*)

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, [1,2,3,4])
sage: F.sum(F.term(i) for i in [1,2,3])
B[1] + B[2] + B[3]
```

term(*i*)

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.term('a')
B['a']
```

zero()

EXAMPLES:

```
sage: F = CombinatorialFreeModule(QQ, ['a', 'b', 'c'])
sage: F.zero()
0
```

17.31.2 Combinatorial Algebras

A combinatorial algebra is an algebra whose basis elements are indexed by a combinatorial class. Some examples of combinatorial algebras are the symmetric group algebra of order n (indexed by permutations of size n) and the algebra of symmetric functions (indexed by integer partitions).

The CombinatorialAlgebra base class makes it easy to define and work with new combinatorial algebras in Sage. For example, the following code constructs an algebra which models the power-sum symmetric functions.

```
sage: class PowerSums(CombinatorialAlgebra):
...       def __init__(self, R):
...           self._combinatorial_class = Partitions()
...           self._one = Partition([])
...           self._name = 'Power-sum symmetric functions'
...           self._prefix = 'p'
...           CombinatorialAlgebra.__init__(self, R)
...       def _multiply_basis(self, a, b):
...           l = list(a)+list(b)
...           l.sort(reverse=True)
...           return Partition(l)
...


sage: ps = PowerSums(QQ); ps
Power-sum symmetric functions over Rational Field
sage: ps([2,1])^2
p[2, 2, 1, 1]
sage: ps([2,1])+2*ps([1,1,1])
2*p[1, 1, 1] + p[2, 1]
sage: ps(2)
2*p[]
```

The important things to define are ._combinatorial_class which specifies the combinatorial class that indexes the basis elements, ._one which specifies the identity element in the algebra, ._name which specifies the name of the algebra,

._prefix which is the string put in front of each basis element, and finally a _multiply or _multiply basis method that defines the multiplication in the algebra.

class CombinatorialAlgebra(*R, element_class=None*)

multiply(*left, right*)

Returns left*right where left and right are elements of self. multiply() uses either _multiply or _multiply basis to carry out the actual multiplication.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: a = s([2])
sage: s.multiply(a,a)
s[2, 2] + s[3, 1] + s[4]
sage: ZS3 = SymmetricGroupAlgebra(ZZ, 3)
sage: a = 2 + ZS3([2,1,3])
sage: a*a
5*[1, 2, 3] + 4*[2, 1, 3]
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ,3)
sage: j2 = H3.jucys_murphy(2)
sage: j2*j2
(q^3-q^2+q)*T[1, 2, 3] + (q^3-q^2+q-1)*T[2, 1, 3]
sage: X = SchubertPolynomialRing(ZZ)
sage: X([1,2,3])*X([2,1,3])
X[2, 1]
```

class CombinatorialAlgebraElement(*A, x*)

to_matrix()

Returns a matrix version of self obtained by the action of self on the left. If new_BR is specified, then the matrix will be over new_BR.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: a = QS3([2,1,3])
sage: a._matrix_() # indirect doctest
[0 0 1 0 0 0]
[0 0 0 0 1 0]
[1 0 0 0 0 0]
[0 0 0 0 0 1]
[0 1 0 0 0 0]
[0 0 0 1 0 0]
sage: a._matrix_(RDF)
[0.0 0.0 1.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 1.0 0.0]
[1.0 0.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0 1.0]
[0.0 1.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 1.0 0.0 0.0]
```

17.31.3 Symmetric Group Algebra

class HeckeAlgebraSymmetricGroupElement_t(*A, x*)

HeckeAlgebraSymmetricGroupT(*R, n, q=None*)

Returns the Hecke algebra of the symmetric group on the T basis.

EXAMPLES:

```
sage: HeckeAlgebraSymmetricGroupT(QQ, 3)
Hecke algebra of the symmetric group of order 3 on the T basis over Univariate Polynomial Ring i

sage: HeckeAlgebraSymmetricGroupT(QQ, 3, 2)
Hecke algebra of the symmetric group of order 3 with q=2 on the T basis over Rational Field
```

class HeckeAlgebraSymmetricGroup_generic(*R, n, q=None*)

q()

EXAMPLES:

```
sage: HeckeAlgebraSymmetricGroupT(QQ, 3).q()
q
sage: HeckeAlgebraSymmetricGroupT(QQ, 3, 2).q()
2
```

class HeckeAlgebraSymmetricGroup_t(*R, n, q=None*)

algebra_generators()

Return the generators of the algebra.

EXAMPLES:

```
sage: HeckeAlgebraSymmetricGroupT(QQ,3).algebra_generators()
[T[2, 1, 3], T[1, 3, 2]]
```

jucys_murphy(*k*)

Returns the Jucys-Murphy element J_k of the Hecke algebra. The Jucys-Murphy elements generate the maximal commutative sub-algebra of the Hecke algebra.

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ,3)
sage: j2 = H3.jucys_murphy(2); j2
q*T[1, 2, 3] + (q-1)*T[2, 1, 3]
sage: j3 = H3.jucys_murphy(3); j3
q^2*T[1, 2, 3] + (q^2-q)*T[1, 3, 2] + (q-1)*T[3, 2, 1]
sage: j2*j3 == j3*j2
True
sage: H3.jucys_murphy(1)
...
ValueError: k must be between 2 and n (= 3)
```

t(*i*)

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ,3)
sage: H3.t(1)
T[2, 1, 3]
sage: H3.t(2)
T[1, 3, 2]
sage: H3.t(0)
...
ValueError: i must be between 1 and n-1 (= 2)
```

t_action(*a, i*)

Return the action of T_i on a.

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3)
sage: a = H3([2,1,3])+2*H3([1,2,3])
sage: H3.t_action(a, 1)
q*T[1, 2, 3] + (q+1)*T[2, 1, 3]
sage: H3.t(1)*a
q*T[1, 2, 3] + (q+1)*T[2, 1, 3]
```

t_action_on_basis(*perm*, *i*)

EXAMPLES:

```
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3)
sage: H3.t_action_on_basis(Permutation([2,1,3]), 1)
q*T[1, 2, 3] + (q-1)*T[2, 1, 3]
sage: H3.t_action_on_basis(Permutation([1,2,3]), 1)
T[2, 1, 3]
sage: H3 = HeckeAlgebraSymmetricGroupT(QQ, 3, 1)
sage: H3.t_action_on_basis(Permutation([2,1,3]), 1)
T[1, 2, 3]
sage: H3.t_action_on_basis(Permutation([1,3,2]), 2)
T[1, 2, 3]
```

SymmetricGroupAlgebra(*R*, *n*)

Returns the symmetric group algebra of order n over R.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3); QS3
Symmetric group algebra of order 3 over Rational Field
sage: QS3(1)
[1, 2, 3]
sage: QS3(2)
2*[1, 2, 3]
sage: basis = [QS3(p) for p in Permutations(3)]
sage: a = sum(basis); a
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: a^2
6*[1, 2, 3] + 6*[1, 3, 2] + 6*[2, 1, 3] + 6*[2, 3, 1] + 6*[3, 1, 2] + 6*[3, 2, 1]
sage: a^2 == 6*a
True
sage: b = QS3([3, 1, 2])
sage: b
[3, 1, 2]
sage: b*a
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: b*a == a
True
```

class **SymmetricGroupAlgebraElement_n**(*A*, *x*)

class **SymmetricGroupAlgebra_n**(*R*, *n*)

cpi(*p*)

Returns the centrally primitive idempotent for the symmetric group of order n for the irreducible corresponding indexed by the partition p.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ,3)
sage: QS3.cpi([2,1])
2/3*[1, 2, 3] - 1/3*[2, 3, 1] - 1/3*[3, 1, 2]
```

```
sage: QS3.cpi([3])
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/6*[3, 2, 1
sage: QS3.cpi([1,1,1])
1/6*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] - 1/6*[3, 2, 1
```

cpis()

Returns a list of the centrally primitive idempotents.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ,3)
sage: a = QS3.cpis()
sage: a[0]  # [3]
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/6*[3, 2, 1
sage: a[1]  # [2, 1]
2/3*[1, 2, 3] - 1/3*[2, 3, 1] - 1/3*[3, 1, 2]
```

dft(*form='seminormal'*)

Returns the discrete Fourier transform for self.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS3.dft()
[   1    1    1    1    1    1]
[   1  1/2   -1 -1/2 -1/2  1/2]
[   0  3/4    0  3/4 -3/4 -3/4]
[   0    1    0   -1    1   -1]
[   1 -1/2    1 -1/2 -1/2 -1/2]
[   1   -1   -1    1    1   -1]
```

epsilon_ik(*itab, ktab, star=0*)

Returns the seminormal basis element of self corresponding to the pair of tableaux itab and ktab.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: a = QS3.epsilon_ik([[1,2,3]], [[1,2,3]]); a
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/6*[3, 2, 1
sage: QS3.dft()*vector(a)
(1, 0, 0, 0, 0, 0)
sage: a = QS3.epsilon_ik([[1,2],[3]], [[1,2],[3]]); a
1/3*[1, 2, 3] - 1/6*[1, 3, 2] + 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2] - 1/6*[3, 2, 1
sage: QS3.dft()*vector(a)
(0, 0, 0, 0, 1, 0)
```

jucys_murphy(*k*)

Returns the Jucys-Murphy element J_k for the symmetric group algebra.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ, 3)
sage: QS3.jucys_murphy(2)
[2, 1, 3]
sage: QS3.jucys_murphy(3)
[1, 3, 2] + [3, 2, 1]
sage: QS5 = SymmetricGroupAlgebra(QQ, 5)
sage: QS5.jucys_murphy(4)
[1, 2, 4, 3, 5] + [1, 4, 3, 2, 5] + [4, 2, 3, 1, 5]
```

seminormal_basis()

Returns a list of the seminormal basis elements of self.

EXAMPLES:

```
sage: QS3 = SymmetricGroupAlgebra(QQ,3)
sage: QS3.seminormal_basis()
[1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/6*[3, 2,
1/3*[1, 2, 3] + 1/6*[1, 3, 2] - 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2] + 1/6*[3, 2, 1
1/3*[1, 3, 2] + 1/3*[2, 3, 1] - 1/3*[3, 1, 2] - 1/3*[3, 2, 1],
1/4*[1, 3, 2] - 1/4*[2, 3, 1] + 1/4*[3, 1, 2] - 1/4*[3, 2, 1],
1/3*[1, 2, 3] - 1/6*[1, 3, 2] + 1/3*[2, 1, 3] - 1/6*[2, 3, 1] - 1/6*[3, 1, 2] - 1/6*[3, 2, 1
1/6*[1, 2, 3] - 1/6*[1, 3, 2] - 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] - 1/6*[3, 2, 1
```

e (*tableau, star=0*)

The unnormalized Young projection operator.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import e
sage: e([[1,2]])
[1, 2] + [2, 1]
sage: e([[1],[2]])
[1, 2] - [2, 1]
```

There are differing conventions for the order of the symmetrizers and antisymmetrizers. This example illustrates our conventions:

```
sage: e([[1,2],[3]])
[1, 2, 3] + [2, 1, 3] - [3, 1, 2] - [3, 2, 1]
```

e_hat (*tab, star=0*)

The Young projection operator, an idempotent in the rational group algebra.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import e_hat
sage: e_hat([[1,2,3]])
1/6*[1, 2, 3] + 1/6*[1, 3, 2] + 1/6*[2, 1, 3] + 1/6*[2, 3, 1] + 1/6*[3, 1, 2] + 1/6*[3, 2, 1]
sage: e_hat([[1],[2]])
1/2*[1, 2] - 1/2*[2, 1]
```

There are differing conventions for the order of the symmetrizers and antisymmetrizers. This example illustrates our conventions:

```
sage: e_hat([[1,2],[3]])
1/3*[1, 2, 3] + 1/3*[2, 1, 3] - 1/3*[3, 1, 2] - 1/3*[3, 2, 1]
```

e_ik (*itab, ktab, star=0*)

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import e_ik
sage: e_ik([[1,2,3]], [[1,2,3]])
[1, 2, 3] + [1, 3, 2] + [2, 1, 3] + [2, 3, 1] + [3, 1, 2] + [3, 2, 1]
sage: e_ik([[1,2,3]], [[1,2,3]], star=1)
[1, 2] + [2, 1]
```

epsilon (*tab, star=0*)

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import epsilon
sage: epsilon([[1,2]])
1/2*[1, 2] + 1/2*[2, 1]
```

```
sage: epsilon([[1],[2]])
1/2*[1, 2] - 1/2*[2, 1]
```

epsilon_ik(*itab, ktab, star=0*)

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import epsilon_ik
sage: epsilon_ik([[1,2],[3]], [[1,3],[2]])
1/4*[1, 3, 2] - 1/4*[2, 3, 1] + 1/4*[3, 1, 2] - 1/4*[3, 2, 1]
sage: epsilon_ik([[1,2],[3]], [[1,3],[2]], star=1)
...
ValueError: the two tableaux must be of the same shape
```

kappa(*alpha*)

Returns κ_α which is n! divided by the number of standard tableaux of shape α.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import kappa
sage: kappa(Partition([2,1]))
3
sage: kappa([2,1])
3
```

pi_ik(*itab, ktab*)

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import pi_ik
sage: pi_ik([[1,3],[2]], [[1,2],[3]])
[1, 3, 2]
```

seminormal_test(*n*)

Runs a variety of tests to verify that the construction of the seminormal basis works as desired. The numbers appearing are Theorems in James and Kerber's 'Representation Theory of the Symmetric Group'.

EXAMPLES:

```
sage: from sage.combinat.symmetric_group_algebra import seminormal_test
sage: seminormal_test(3)
True
```

17.31.4 Schubert Polynomials

SchubertPolynomialRing(*R*)

Returns the Schubert polynomial ring over R on the X basis.

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ); X
Schubert polynomial ring with X basis over Integer Ring
sage: X(1)
X[1]
sage: X([1,2,3])*X([2,1,3])
X[2, 1]
sage: X([2,1,3])*X([2,1,3])
X[3, 1, 2]
sage: X([2,1,3])+X([3,1,2,4])
```

```
X[2, 1] + X[3, 1, 2]
sage: a = X([2,1,3])+X([3,1,2,4])
sage: a^2
X[3, 1, 2] + 2*X[4, 1, 2, 3] + X[5, 1, 2, 3, 4]
```

class SchubertPolynomialRing_xbasis(R)

class SchubertPolynomial_class(A, x)

divided_difference(i)

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: a = X([3,2,1])
sage: a.divided_difference(1)
X[2, 3, 1]
sage: a.divided_difference([3,2,1])
X[1]
```

expand()

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: X([2,1,3]).expand()
x0
sage: map(lambda x: x.expand(), [X(p) for p in Permutations(3)])
[1, x0 + x1, x0, x0*x1, x0^2, x0^2*x1]
```

TESTS: Calling .expand() should always return an element of an MPolynomialRing

```
sage: X = SchubertPolynomialRing(ZZ)
sage: f = X([1]); f
X[1]
sage: type(f.expand())
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>
sage: f.expand()
1
sage: f = X([1,2])
sage: type(f.expand())
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>
sage: f = X([1,3,2,4])
sage: type(f.expand())
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>
```

multiply_variable(i)

Returns the Schubert polynomial obtained by multiplying self by the variable x_i.

EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: a = X([3,2,4,1])
sage: a.multiply_variable(0)
X[4, 2, 3, 1]
sage: a.multiply_variable(1)
X[3, 4, 2, 1]
sage: a.multiply_variable(2)
X[3, 2, 5, 1, 4] - X[3, 4, 2, 1] - X[4, 2, 3, 1]
sage: a.multiply_variable(3)
X[3, 2, 4, 5, 1]
```

scalar_product(x)

Returns the standard scalar product of self and x.
EXAMPLES:

```
sage: X = SchubertPolynomialRing(ZZ)
sage: a = X([3,2,4,1])
sage: a.scalar_product(a)
0
sage: b = X([4,3,2,1])
sage: b.scalar_product(a)
X[1, 3, 4, 6, 2, 5]
sage: Permutation([1, 3, 4, 6, 2, 5, 7]).to_lehmer_code()
[0, 1, 1, 2, 0, 0, 0]
sage: s = SFASchur(ZZ)
sage: c = s([2,1,1])
sage: b.scalar_product(a).expand()
x0^2*x1*x2 + x0*x1^2*x2 + x0*x1*x2^2 + x0^2*x1*x3 + x0*x1^2*x3 + x0^2*x2*x3 + 3*x0*x1*x2*x3
sage: c.expand(4)
x0^2*x1*x2 + x0*x1^2*x2 + x0*x1*x2^2 + x0^2*x1*x3 + x0*x1^2*x3 + x0^2*x2*x3 + 3*x0*x1*x2*x3
```

is_SchubertPolynomial(*x*)

Returns True if x is a Schubert polynomial and False otherwise.

EXAMPLES:

```
sage: from sage.combinat.schubert_polynomial import is_SchubertPolynomial
sage: X = SchubertPolynomialRing(ZZ)
sage: a = 1
sage: is_SchubertPolynomial(a)
False
sage: b = X(1)
sage: is_SchubertPolynomial(b)
True
sage: c = X([2,1,3])
sage: is_SchubertPolynomial(c)
True
```

17.31.5 Partition/Diagram Algebras

class PartitionAlgebraElement_ak(*A*, *x*)

class PartitionAlgebraElement_bk(*A*, *x*)

class PartitionAlgebraElement_generic(*A*, *x*)

class PartitionAlgebraElement_pk(*A*, *x*)

class PartitionAlgebraElement_prk(*A*, *x*)

class PartitionAlgebraElement_rk(*A*, *x*)

class PartitionAlgebraElement_sk(*A*, *x*)

class PartitionAlgebraElement_tk(*A*, *x*)

class PartitionAlgebra_ak(*R*, *k*, *n*, *name=None*)

class PartitionAlgebra_bk(*R*, *k*, *n*, *name=None*)

class PartitionAlgebra_generic(*R*, *cclass*, *n*, *k*, *name=None*, *prefix=None*)

class PartitionAlgebra_pk(*R*, *k*, *n*, *name=None*)

class PartitionAlgebra_prk(*R*, *k*, *n*, *name=None*)

class **PartitionAlgebra_rk**(*R, k, n, name=None*)

class **PartitionAlgebra_sk**(*R, k, n, name=None*)

class **PartitionAlgebra_tk**(*R, k, n, name=None*)

class **SetPartitionsAk_k**(*k*)

class **SetPartitionsAkhalf_k**(*k*)

cardinality()

TESTS:

```
sage: SetPartitionsAk(1.5).cardinality()
5
sage: SetPartitionsAk(2.5).cardinality()
52
sage: SetPartitionsAk(3.5).cardinality()
877
```

class **SetPartitionsBk_k**(*k*)

cardinality()

Returns the number of set partitions in B_k where k is an integer. This is given by (2k)!! = (2k-1)*(2k-3)*...*5*3*1.

EXAMPLES:

```
sage: SetPartitionsBk(3).cardinality()
15
sage: SetPartitionsBk(2).cardinality()
3
sage: SetPartitionsBk(1).cardinality()
1
sage: SetPartitionsBk(4).cardinality()
105
sage: SetPartitionsBk(5).cardinality()
945
```

class **SetPartitionsBkhalf_k**(*k*)

cardinality()

TESTS:

```
sage: B3p5 = SetPartitionsBk(3.5)
sage: B3p5.cardinality()
15
```

class **SetPartitionsIk_k**(*k*)

cardinality()

TESTS:

```
sage: SetPartitionsIk(2).cardinality()
13
```

class **SetPartitionsIkhalf_k**(*k*)

cardinality()

TESTS:

```
sage: SetPartitionsIk(1.5).cardinality()
4
sage: SetPartitionsIk(2.5).cardinality()
50
sage: SetPartitionsIk(3.5).cardinality()
871
```

class SetPartitionsPRk_k(k)

cardinality()
TESTS:

```
sage: SetPartitionsPRk(2).cardinality()
6
sage: SetPartitionsPRk(3).cardinality()
20
sage: SetPartitionsPRk(4).cardinality()
70
sage: SetPartitionsPRk(5).cardinality()
252
```

class SetPartitionsPRkhalf_k(k)

cardinality()
TESTS:

```
sage: SetPartitionsPRk(2.5).cardinality()
6
sage: SetPartitionsPRk(3.5).cardinality()
20
sage: SetPartitionsPRk(4.5).cardinality()
70
```

class SetPartitionsPk_k(k)

cardinality()
TESTS:

```
sage: SetPartitionsPk(2).cardinality()
14
sage: SetPartitionsPk(3).cardinality()
132
sage: SetPartitionsPk(4).cardinality()
1430
```

class SetPartitionsPkhalf_k(k)

cardinality()
TESTS:

```
sage: SetPartitionsPk(2.5).cardinality()
42
sage: SetPartitionsPk(1.5).cardinality()
5
```

class SetPartitionsRk_k(k)

cardinality()
TESTS:

```
sage: SetPartitionsRk(2).cardinality()
7
sage: SetPartitionsRk(3).cardinality()
34
sage: SetPartitionsRk(4).cardinality()
209
sage: SetPartitionsRk(5).cardinality()
1546
```

class **SetPartitionsRkhalf_k**(k)

cardinality()
TESTS:

```
sage: SetPartitionsRk(2.5).cardinality()
7
sage: SetPartitionsRk(3.5).cardinality()
34
sage: SetPartitionsRk(4.5).cardinality()
209
```

class **SetPartitionsSk_k**(k)

cardinality()
Returns k!.
TESTS:

```
sage: SetPartitionsSk(2).cardinality()
2
sage: SetPartitionsSk(3).cardinality()
6
sage: SetPartitionsSk(4).cardinality()
24
sage: SetPartitionsSk(5).cardinality()
120
```

class **SetPartitionsSkhalf_k**(k)

cardinality()
TESTS:

```
sage: SetPartitionsSk(2.5).cardinality()
2
sage: SetPartitionsSk(3.5).cardinality()
6
sage: SetPartitionsSk(4.5).cardinality()
24

sage: ks = [2.5, 3.5, 4.5, 5.5]
sage: sks = [SetPartitionsSk(k) for k in ks]
sage: all([ sk.cardinality() == len(sk.list()) for sk in sks])
True
```

class **SetPartitionsTk_k**(k)

cardinality()
TESTS:

```
sage: SetPartitionsTk(2).cardinality()
2
sage: SetPartitionsTk(3).cardinality()
5
sage: SetPartitionsTk(4).cardinality()
14
sage: SetPartitionsTk(5).cardinality()
42
```

class **SetPartitionsTkhalf_k**(*k*)

cardinality()
TESTS:

```
sage: SetPartitionsTk(2.5).cardinality()
2
sage: SetPartitionsTk(3.5).cardinality()
5
sage: SetPartitionsTk(4.5).cardinality()
14
```

create_set_partition_function(*letter, k*)
EXAMPLES:

```
sage: from sage.combinat.partition_algebra import create_set_partition_function
sage: create_set_partition_function('A', 3)
Set partitions of {1, ..., 3, -1, ..., -3}
```

identity(*k*)
Returns the identity set partition 1, -1, ..., k, -k

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.identity(2)
{{2, -2}, {1, -1}}
```

is_planar(*sp*)
Returns True if the diagram corresponding to the set partition is planar; otherwise, it returns False.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.is_planar( pa.to_set_partition([[1,-2],[2,-1]]))
False
sage: pa.is_planar( pa.to_set_partition([[1,-1],[2,-2]]))
True
```

pair_to_graph(*sp1, sp2*)
Returns a graph consisting of the graphs of set partitions sp1 and sp2 along with edges joining the bottom row (negative numbers) of sp1 to the top row (positive numbers) of sp2.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,-2],[2,-1]])
```

```
sage: g = pa.pair_to_graph( sp1, sp2 ); g
Graph on 8 vertices

sage: g.vertices() #random
[(1, 2), (-1, 1), (-2, 2), (-1, 2), (-2, 1), (2, 1), (2, 2), (1, 1)]
sage: g.edges() #random
[((1, 2), (-1, 1), None),
 ((1, 2), (-2, 2), None),
 ((-1, 1), (2, 1), None),
 ((-1, 2), (2, 2), None),
 ((-2, 1), (1, 1), None),
 ((-2, 1), (2, 2), None)]
```

propagating_number(*sp*)

Returns the propagating number of the set partition sp. The propagating number is the number of blocks with both a positive and negative number.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,2],[-2,-1]])
sage: pa.propagating_number(sp1)
2
sage: pa.propagating_number(sp2)
0
```

set_partition_composition(*sp1, sp2*)

Returns a tuple consisting of the composition of the set partitions sp1 and sp2 and the number of components removed from the middle rows of the graph.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: sp1 = pa.to_set_partition([[1,-2],[2,-1]])
sage: sp2 = pa.to_set_partition([[1,-2],[2,-1]])
sage: pa.set_partition_composition(sp1, sp2) == (pa.identity(2), 0)
True
```

to_graph(*sp*)

Returns a graph representing the set partition sp.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: g = pa.to_graph( pa.to_set_partition([[1,-2],[2,-1]])); g
Graph on 4 vertices

sage: g.vertices() #random
[1, 2, -2, -1]
sage: g.edges() #random
[(1, -2, None), (2, -1, None)]
```

to_set_partition(*l, k=None*)

Coverts a list of a list of numbers to a set partitions. Each list of numbers in the outer list specifies the numbers contained in one of the blocks in the set partition.

If k is specified, then the set partition will be a set partition of 1, ..., k, -1, ..., -k. Otherwise, k will default to the minimum number needed to contain all of the specified numbers.

EXAMPLES:

```
sage: import sage.combinat.partition_algebra as pa
sage: pa.to_set_partition([[1,-1],[2,-2]]) == pa.identity(2)
True
```

17.32 Tableaux and Tableaux-like Objects

17.32.1 Tableaux

SemistandardTableaux(*p=None, mu=None*)

Returns the combinatorial class of semistandard tableaux.

If p is specified and is a partition, then it returns the class of semistandard tableaux of shape p (and max entry sum(p))

If p is specified and is an integer, it returns the class of semistandard tableaux of size p.

If mu is also specified, then it returns the class of semistandard tableaux with evaluation/content mu.

EXAMPLES:

```
sage: SST = SemistandardTableaux([2,1]); SST
Semistandard tableaux of shape [2, 1]
sage: SST.list()
[[[1, 1], [2]],
 [[1, 1], [3]],
 [[1, 2], [2]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[1, 3], [3]],
 [[2, 2], [3]],
 [[2, 3], [3]]]
```

```
sage: SST = SemistandardTableaux(3); SST
Semistandard tableaux of size 3
sage: SST.list()
[[[1, 1, 1]],
 [[1, 1, 2]],
 [[1, 1, 3]],
 [[1, 2, 2]],
 [[1, 2, 3]],
 [[1, 3, 3]],
 [[2, 2, 2]],
 [[2, 2, 3]],
 [[2, 3, 3]],
 [[3, 3, 3]],
 [[1, 1], [2]],
 [[1, 1], [3]],
 [[1, 2], [2]],
 [[1, 2], [3]],
 [[1, 3], [2]],
 [[1, 3], [3]],
 [[2, 2], [3]],
```

```
[[2, 3], [3]],
[[1], [2], [3]]]
```

class **SemistandardTableaux_all**()

list()
TESTS:

```
sage: SemistandardTableaux().list()
...
NotImplementedError
```

class **SemistandardTableaux_n**(*n*)

cardinality()
EXAMPLES:

```
sage: SemistandardTableaux(3).cardinality()
19
sage: SemistandardTableaux(4).cardinality()
116
sage: ns = range(1, 6)
sage: ssts = [ SemistandardTableaux(n) for n in ns ]
sage: all([sst.cardinality() == len(sst.list()) for sst in ssts])
True
```

object_class()

class **SemistandardTableaux_nmu**(*n*, *mu*)

cardinality()
EXAMPLES:

```
sage: SemistandardTableaux(3, [2,1]).cardinality()
2
sage: SemistandardTableaux(4, [2,2]).cardinality()
3
```

class **SemistandardTableaux_p**(*p*)

cardinality()
EXAMPLES:

```
sage: SemistandardTableaux([2,1]).cardinality()
8
sage: SemistandardTableaux([2,2,1]).cardinality()
75
sage: s = SFASchur(QQ)
sage: s([2,2,1]).expand(5)(1,1,1,1,1)
75
sage: SemistandardTableaux([5]).cardinality()
126
sage: SemistandardTableaux([3,2,1]).cardinality()
896
```

object_class()

class **SemistandardTableaux_pmu**(*p*, *mu*)

cardinality()

EXAMPLES:

```
sage: SemistandardTableaux([2,2], [2, 1, 1]).cardinality()
1
sage: SemistandardTableaux([2,2,2], [2, 2, 1,1]).cardinality()
1
sage: SemistandardTableaux([2,2,2], [2, 2, 2]).cardinality()
1
sage: SemistandardTableaux([3,2,1], [2, 2, 2]).cardinality()
2
```

list()

EXAMPLES:

```
sage: SemistandardTableaux([2,2], [2, 1, 1]).list()
[[[1, 1], [2, 3]]]
sage: SemistandardTableaux([2,2,2], [2, 2, 1,1]).list()
[[[1, 1], [2, 2], [3, 4]]]
sage: SemistandardTableaux([2,2,2], [2, 2, 2]).list()
[[[1, 1], [2, 2], [3, 3]]]
sage: SemistandardTableaux([3,2,1], [2, 2, 2]).list()
[[[1, 1, 2], [2, 3], [3]], [[1, 1, 3], [2, 2], [3]]]
```

object_class()

StandardTableau(t)

Returns the standard tableau object corresponding to t.

Note that Sage uses the English convention for partitions and tableaux.

EXAMPLES:

```
sage: t = StandardTableau([[1,2,3],[4,5]]); t
[[1, 2, 3], [4, 5]]
sage: t.shape()
[3, 2]
sage: t.is_standard()
True

sage: t = StandardTableau([[1,2,3],[4,4]])
...
ValueError: not a standard tableau
```

class **StandardTableau_class**(t)

content(k)

Returns the content of k in a standard tableau. That is, if k appears in row r and column c of the tableau then we return $c - r$.

EXAMPLES:

```
sage: StandardTableau([[1,2],[3,4]]).content(3)
-1

sage: StandardTableau([[1,2],[3,4]]).content(6)
...
ValueError: 6 does not appear in tableau
```

StandardTableaux(*n=None*)

Returns the combinatorial class of standard tableaux. If n is specified and is an integer, then it returns the combinatorial class of all standard tableaux of size n. If n is a partition, then it returns the class of all standard tableaux of shape n.

EXAMPLES:

```
sage: ST = StandardTableaux(3); ST
Standard tableaux of size 3
sage: ST.first()
[[1, 2, 3]]
sage: ST.last()
[[1], [2], [3]]
sage: ST.cardinality()
4
sage: ST.list()
[[[1, 2, 3]], [[1, 3], [2]], [[1, 2], [3]], [[1], [2], [3]]]


sage: ST = StandardTableaux([2,2]); ST
Standard tableaux of shape [2, 2]
sage: ST.first()
[[1, 3], [2, 4]]
sage: ST.last()
[[1, 2], [3, 4]]
sage: ST.cardinality()
2
sage: ST.list()
[[[1, 3], [2, 4]], [[1, 2], [3, 4]]]
```

class **StandardTableaux_all**()

class **StandardTableaux_n**(*n*)

cardinality()

EXAMPLES:

```
sage: StandardTableaux(3).cardinality()
4
sage: ns = [1,2,3,4,5,6]
sage: sts = [StandardTableaux(n) for n in ns]
sage: all([st.cardinality() == len(st.list()) for st in sts])
True
```

object_class()

class **StandardTableaux_partition**(*p*)

cardinality()

Returns the number of standard Young tableaux associated with a partition pi

A formula for the number of Young tableaux associated with a given partition. In each box, write the sum of one plus the number of boxes horizontally to the right and vertically below the box (the hook length). The number of tableaux is then n! divided by the product of all hook lengths.

For example, consider the partition [3,2,1] of 6 with Ferrers Diagram:

```
# # #
# #
#
```

When we fill in the boxes with the hook lengths, we obtain:

```
5 3 1
3 1
1
```

The hook length formula returns 6!/(5*3*1*3*1*1) = 16.

EXAMPLES:

```
sage: StandardTableaux([3,2,1]).cardinality()
16
sage: StandardTableaux([2,2]).cardinality()
2
sage: StandardTableaux([5]).cardinality()
1
sage: StandardTableaux([6,5,5,3]).cardinality()
6651216
```

REFERENCES:

- http://mathworld.wolfram.com/HookLengthFormula.html

list()

Returns a list of the standard Young tableau associated with a partition p.

EXAMPLES:

```
sage: StandardTableaux([2,2]).list()
[[[1, 3], [2, 4]], [[1, 2], [3, 4]]]
sage: StandardTableaux([5]).list()
[[[1, 2, 3, 4, 5]]]
sage: StandardTableaux([3,2,1]).list()
[[[1, 4, 6], [2, 5], [3]],
 [[1, 3, 6], [2, 5], [4]],
 [[1, 2, 6], [3, 5], [4]],
 [[1, 3, 6], [2, 4], [5]],
 [[1, 2, 6], [3, 4], [5]],
 [[1, 4, 5], [2, 6], [3]],
 [[1, 3, 5], [2, 6], [4]],
 [[1, 2, 5], [3, 6], [4]],
 [[1, 3, 4], [2, 6], [5]],
 [[1, 2, 4], [3, 6], [5]],
 [[1, 2, 3], [4, 6], [5]],
 [[1, 3, 5], [2, 4], [6]],
 [[1, 2, 5], [3, 4], [6]],
 [[1, 3, 4], [2, 5], [6]],
 [[1, 2, 4], [3, 5], [6]],
 [[1, 2, 3], [4, 5], [6]]]
```

random_element()

Returns a random standard tableau of shape p using the Green-Nijenhuis-Wilf Algorithm.

EXAMPLES:

```
sage: StandardTableaux([2,2]).random_element()
[[1, 2], [3, 4]]
```

Tableau(*t*)

Returns the tableau object corresponding to t.

Note that Sage uses the English convention for partitions and tableaux.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[4,5]]); t
[[1, 2, 3], [4, 5]]
sage: t.shape()
[3, 2]
sage: t.is_standard()
True
```

class Tableau_class(*t*)

anti_restrict(*n*)

Returns the skew tableau formed by removing all of the boxes from self that are filled with a number less than

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[4,5]]); t
[[1, 2, 3], [4, 5]]
sage: t.anti_restrict(1)
[[None, 2, 3], [4, 5]]
sage: t.anti_restrict(2)
[[None, None, 3], [4, 5]]
sage: t.anti_restrict(3)
[[None, None, None], [4, 5]]
sage: t.anti_restrict(4)
[[None, None, None], [None, 5]]
```

atom()

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).atom()
[2, 2]
sage: Tableau([[1,2,3],[4,5],[6]]).atom()
[3, 2, 1]
```

attacking_pairs()

Returns a list of the attacking pairs of self. An pair of boxes (c, d) is said to be attacking if one of the following conditions hold:

1.c and d lie in the same row with c to the west of d

2.c is in the row immediately to the south of d and c lies strictly east of d.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[2,5]])
sage: t.attacking_pairs()
[((0, 0), (0, 1)),
 ((0, 0), (0, 2)),
 ((0, 1), (0, 2)),
 ((1, 0), (1, 1)),
 ((1, 1), (0, 0))]
```

boxes()

Returns a list of the coordinates of the boxes of self.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).boxes()
[(0, 0), (0, 1), (1, 0), (1, 1)]
```

bump(*x*)

Schensted's row-bumping (or row-insertion) algorithm.

EXAMPLES:

```
sage: t = Tableau([[1,2],[3]])
sage: t.bump(1)
[[1, 1], [2], [3]]
sage: t
[[1, 2], [3]]
sage: t.bump(2)
[[1, 2, 2], [3]]
sage: t.bump(3)
[[1, 2, 3], [3]]
sage: t
[[1, 2], [3]]
sage: t = Tableau([[1,2,2,3],[2,3,5,5],[4,4,6],[5,6]])
sage: t.bump(2)
[[1, 2, 2, 2], [2, 3, 3, 5], [4, 4, 5], [5, 6, 6]]
```

bump_multiply(*left, right*)

Multiply two tableaux using Schensted's bump.

This product makes the set of tableaux into an associative monoid. The empty tableaux is the unit in this monoid.

Fulton, William. 'Young Tableaux' p11-12

EXAMPLES:

```
sage: t = Tableau([[1,2,2,3],[2,3,5,5],[4,4,6],[5,6]])
sage: t2 = Tableau([[1,2],[3]])
sage: t.bump_multiply(t2)
[[1, 1, 2, 2, 3], [2, 2, 3, 5], [3, 4, 5], [4, 6, 6], [5]]
```

charge()

EXAPMLES:

```
sage: Tableau([[1,1],[2,2],[3]]).charge()
0
sage: Tableau([[1,1,3],[2,2]]).charge()
1
sage: Tableau([[1,1,2],[2],[3]]).charge()
1
sage: Tableau([[1,1,2],[2,3]]).charge()
2
sage: Tableau([[1,1,2,3],[2]]).charge()
2
sage: Tableau([[1,1,2,2],[3]]).charge()
3
sage: Tableau([[1,1,2,2,3]]).charge()
4
```

cocharge()

EXAPMLES:

```
sage: Tableau([[1,1],[2,2],[3]]).cocharge()
4
sage: Tableau([[1,1,3],[2,2]]).cocharge()
3
sage: Tableau([[1,1,2],[2],[3]]).cocharge()
2
sage: Tableau([[1,1,2],[2,3]]).cocharge()
2
sage: Tableau([[1,1,2,3],[2]]).cocharge()
1
sage: Tableau([[1,1,2,2],[3]]).cocharge()
```

```
1
sage: Tableau([[1,1,2,2,3]]).cocharge()
0
```

column_stabilizer()
Return the PermutationGroup corresponding to the column stabilizer of self.
EXAMPLES:

```
sage: cs = Tableau([[1,2,3],[4,5]]).column_stabilizer()
sage: cs.order() == factorial(2)*factorial(2)
True
sage: PermutationGroupElement([(1,3,2),(4,5)]) in cs
False
sage: PermutationGroupElement([(1,4)]) in cs
True
```

conjugate()
Returns the conjugate of the tableau t.
EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).conjugate()
[[1, 3], [2, 4]]
```

corners()
Returns the corners of the tableau t.
EXAMPLES:

```
sage: Tableau([[1, 4, 6], [2, 5], [3]]).corners()
[[0, 2], [1, 1], [2, 0]]
sage: Tableau([[1, 3], [2, 4]]).corners()
[[1, 1]]
```

descents()
Returns a list of the boxes (i,j) such that self[i][j] > self[i-1][j].
EXAMPLES:

```
sage: Tableau( [[1,4],[2,3]] ).descents()
[(1, 0)]
sage: Tableau( [[1,2],[3,4]] ).descents()
[(1, 0), (1, 1)]
```

down()
An iterator for all the tableaux that can be obtained from self by removing a box. Note that this iterates just over a single tableaux. EXAMPLES:

```
sage: t = Tableau([[1,2],[3]])
sage: [x for x in t.down()]
[[[1, 2]]]
```

down_list()
Returns a list of all the tableaux that can be obtained from self by removing a box. Note that this is just a single tableaux.
EXAMPLES:

```
sage: t = Tableau([[1,2],[3]])
sage: t.down_list()
[[[1, 2]]]
```

entry(*box*)
Returns the entry of box in self. Box is a tuple (i,j) of coordinates.

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,4]])
sage: t.entry( (0,0) )
1
sage: t.entry( (1,1) )
4
```

evaluation()

Returns the evaluation of the word from tableau t.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).evaluation()
[1, 1, 1, 1]
```

height()

Returns the height of the tableau.

EXAMPLES:

```
sage: Tableau([[1,2,3],[4,5]]).height()
2
sage: Tableau([[1,2,3]]).height()
1
sage: Tableau([]).height()
0
```

insert_word(*w, left=False*)

EXAMPLES:

```
sage: t0 = Tableau([])
sage: w = [1,1,2,3,3,3,3]
sage: t0.insert_word(w)
[[1, 1, 2, 3, 3, 3, 3]]
sage: t0.insert_word(w,left=True)
[[1, 1, 2, 3, 3, 3, 3]]
sage: w.reverse()
sage: t0.insert_word(w)
[[1, 1, 3, 3], [2, 3], [3]]
sage: t0.insert_word(w,left=True)
[[1, 1, 3, 3], [2, 3], [3]]
```

inversion_number()

Returns the inversion number of self.

The inversion number is defined to be the number of inversion of self minus the sum of the arm lengths of the descents of self.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[2,5]])
sage: t.inversion_number()
0
```

inversions()

Returns a list of the inversions of self. An inversion is an attacking pair (c,d) such that the entry of c in self is greater than the entry of d.

EXAMPLES:

```
sage: t = Tableau([[1,2,3],[2,5]])
sage: t.inversions()
[((1, 1), (0, 0))]
```

is_rectangular()
Returns True if the tableau t is rectangular and False otherwise.
EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).is_rectangular()
True
sage: Tableau([[1,2,3],[4,5],[6]]).is_rectangular()
False
```

is_standard()
Returns True if t is a standard tableau and False otherwise.
EXAMPLES:

```
sage: Tableau([[1, 3], [2, 4]]).is_standard()
True
sage: Tableau([[1, 2], [2, 4]]).is_standard()
False
sage: Tableau([[2, 3], [2, 4]]).is_standard()
False
sage: Tableau([[5, 3], [2, 4]]).is_standard()
False
```

k_weight(*k*)
Returns the k-weight of self.
EXAMPLES:

```
sage: Tableau([[1,2],[2,3]]).k_weight(1)
[1, 1, 1]
sage: Tableau([[1,2],[2,3]]).k_weight(2)
[1, 2, 1]
sage: t = Tableau([[1,1,1,2,5],[2,3,6],[3],[4]])
sage: t.k_weight(1)
[2, 1, 1, 1, 1, 1]
sage: t.k_weight(2)
[3, 2, 2, 1, 1, 1]
sage: t.k_weight(3)
[3, 1, 2, 1, 1, 1]
sage: t.k_weight(4)
[3, 2, 2, 1, 1, 1]
sage: t.k_weight(5)
[3, 2, 2, 1, 1, 1]
```

katabolism()
EXAMPLES:

```
sage: Tableau([]).katabolism()
[]
sage: Tableau([[1,2,3,4,5]]).katabolism()
[[1, 2, 3, 4, 5]]
sage: Tableau([[1,1,3,3],[2,3],[3]]).katabolism()
[[1, 1, 2, 3, 3, 3], [3]]
sage: Tableau([[1, 1, 2, 3, 3, 3], [3]]).katabolism()
[[1, 1, 2, 3, 3, 3, 3]]
```

katabolism_projector(*parts*)
EXAMPLES:

```
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.katabolism_projector([[4,2,1]])
[[1, 1, 3, 3], [2, 3], [3]]
```

```
sage: t.katabolism_projector([[1]])
[]
sage: t.katabolism_projector([[2,1],[1]])
[]
sage: t.katabolism_projector([[1,1],[4,1]])
[[1, 1, 3, 3], [2, 3], [3]]
```

katabolism_sequence()

EXAMPLES:

```
sage: t = Tableau([[1,2,3,4,5,6,8],[7,9]])
sage: t.katabolism_sequence()
[[[1, 2, 3, 4, 5, 6, 8], [7, 9]],
 [[1, 2, 3, 4, 5, 6, 7, 9], [8]],
 [[1, 2, 3, 4, 5, 6, 7, 8], [9]],
 [[1, 2, 3, 4, 5, 6, 7, 8, 9]]]
```

lambda_katabolism(*part*)

EXAMPLES:

```
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.lambda_katabolism([])
[[1, 1, 3, 3], [2, 3], [3]]
sage: t.lambda_katabolism([1])
[[1, 2, 3, 3, 3], [3]]
sage: t.lambda_katabolism([1,1])
[[1, 3, 3, 3], [3]]
sage: t.lambda_katabolism([2,1])
[[3, 3, 3, 3]]
sage: t.lambda_katabolism([4,2,1])
[]
sage: t.lambda_katabolism([5,1])
[[3, 3]]
sage: t.lambda_katabolism([4,1])
[[3, 3]]
```

last_letter_lequal(*tab2*)

Returns True if self is less than or equal to tab2 in the last letter ordering.

EXAMPLES:

```
sage: st = StandardTableaux([3,2])
sage: f = lambda b: 1 if b else 0
sage: matrix( [ [ f(t1.last_letter_lequal(t2)) for t2 in st] for t1 in st] )
[1 1 1 1 1]
[0 1 1 1 1]
[0 0 1 1 1]
[0 0 0 1 1]
[0 0 0 0 1]
```

major_index()

Returns the major index of self. The major index is defined to be the sum of the number of descents of self and the sum of their legs.

EXAMPLES

```
sage: Tableau( [[1,4],[2,3]] ).major_index()
1
sage: Tableau( [[1,2],[3,4]] ).major_index()
2
```

pp()

Returns a pretty print string of the tableau.

EXAMPLES:

```
sage: Tableau([[1,2,3],[3,4],[5]]).pp()
  1  2  3
  3  4
  5
```

promotion(*n*)

Promotion operator defined on rectangular tableaux using jeu de taquin

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,3]])
sage: t.promotion(2)
[[1, 1], [2, 3]]
sage: t = Tableau([[1,1,1],[2,2,3],[3,4,4]])
sage: t.promotion(3)
[[1, 1, 2], [2, 2, 3], [3, 4, 4]]
sage: t = Tableau([[1,2],[2]])
sage: t.promotion(3)
...
ValueError: Tableau is not rectangular
```

promotion_inverse(*n*)

Inverse promotion operator defined on rectangular tableaux using jeu de taquin

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,3]])
sage: t.promotion_inverse(2)
[[1, 2], [2, 3]]
sage: t = Tableau([[1,2],[2,3]])
sage: t.promotion_inverse(2)
[[1, 1], [2, 3]]
```

promotion_operator(*i*)

EXAMPLES:

```
sage: t = Tableau([[1,2],[3]])
sage: t.promotion_operator(1)
[[[1, 2], [3], [4]], [[1, 2], [3, 4]], [[1, 2, 4], [3]]]
sage: t.promotion_operator(2)
[[[1, 1], [2, 3], [4]],
 [[1, 1, 2], [3], [4]],
 [[1, 1, 4], [2, 3]],
 [[1, 1, 2, 4], [3]]]
sage: Tableau([[1]]).promotion_operator(2)
[[[1, 1], [2]], [[1, 1, 2]]]
sage: Tableau([[1,1],[2]]).promotion_operator(3)
[[[1, 1, 1], [2, 2], [3]],
 [[1, 1, 1, 2], [2], [3]],
 [[1, 1, 1, 3], [2, 2]],
 [[1, 1, 1, 2, 3], [2]]]
```

TESTS:

```
sage: Tableau([]).promotion_operator(2)
[[[1, 1]]]
sage: Tableau([]).promotion_operator(1)
[[[1]]]
```

raise_action_from_words(*f*, **args*)

EXAMPLES:

```
sage: from sage.combinat.tableau import symmetric_group_action_on_values
sage: import functools
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: f = functools.partial(t.raise_action_from_words, symmetric_group_action_on_values)
sage: f([1,2,3])
[[1, 1, 3, 3], [2, 3], [3]]
sage: f([3,2,1])
[[1, 1, 1, 1], [2, 3], [3]]
sage: f([1,3,2])
[[1, 1, 2, 2], [2, 2], [3]]
```

reduced_lambda_katabolism(*part*)

EXAMPLES:

```
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.reduced_lambda_katabolism([])
[[1, 1, 3, 3], [2, 3], [3]]
sage: t.reduced_lambda_katabolism([1])
[[1, 2, 3, 3, 3], [3]]
sage: t.reduced_lambda_katabolism([1,1])
[[1, 3, 3, 3], [3]]
sage: t.reduced_lambda_katabolism([2,1])
[[3, 3, 3, 3]]
sage: t.reduced_lambda_katabolism([4,2,1])
[]
sage: t.reduced_lambda_katabolism([5,1])
0
sage: t.reduced_lambda_katabolism([4,1])
0
```

restrict(*n*)

Returns the restriction of the standard tableau to n.

EXAMPLES:

```
sage: Tableau([[1,2],[3],[4]]).restrict(3)
[[1, 2], [3]]
sage: Tableau([[1,2],[3],[4]]).restrict(2)
[[1, 2]]
sage: Tableau([[1,1],[2]]).restrict(1)
[[1, 1]]
```

rotate_180()

Returns the tableau obtained by rotating t by 180 degrees.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).rotate_180()
[[4, 3], [2, 1]]
```

row_stabilizer()

Return the PermutationGroup corresponding to the row stabilizer of self.

EXAMPLES:

```
sage: rs = Tableau([[1,2,3],[4,5]]).row_stabilizer()
sage: rs.order() == factorial(3)*factorial(2)
True
sage: PermutationGroupElement([(1,3,2),(4,5)]) in rs
True
```

```
sage: PermutationGroupElement([(1,4)]) in rs
False
sage: rs = Tableau([[1],[2],[3]]).row_stabilizer()
sage: rs.order()
1
```

schensted_insert(*i, left=False*)

EXAMPLES:

```
sage: t = Tableau([[3,5],[7]])
sage: t.schensted_insert(8)
[[3, 5, 8], [7]]
sage: t.schensted_insert(8, left=True)
[[3, 5], [7], [8]]
```

shape()

Returns the shape of a tableau t.

EXAMPLES:

```
sage: Tableau([[1,2,3],[4,5],[6]]).shape()
[3, 2, 1]
```

size()

Returns the size of the shape of the tableau t.

EXAMPLES:

```
sage: Tableau([[1, 4, 6], [2, 5], [3]]).size()
6
sage: Tableau([[1, 3], [2, 4]]).size()
4
```

slide_multiply(*left, right*)

Multiply two tableaux using jeu de taquin.

This product makes the set of tableaux into an associative monoid. The empty tableaux is the unit in this monoid.

Fulton, William. 'Young Tableaux' p15

EXAMPLES:

```
sage: t = Tableau([[1,2,2,3],[2,3,5,5],[4,4,6],[5,6]])
sage: t2 = Tableau([[1,2],[3]])
sage: t.slide_multiply(t2)
[[1, 1, 2, 2, 3], [2, 2, 3, 5], [3, 4, 5], [4, 6, 6], [5]]
```

socle()

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).socle()
2
sage: Tableau([[1,2,3,4]]).socle()
4
```

symmetric_group_action_on_values(*perm*)

EXAMPLES:

```
sage: t = Tableau([[1,1,3,3],[2,3],[3]])
sage: t.symmetric_group_action_on_values([1,2,3])
[[1, 1, 3, 3], [2, 3], [3]]
sage: t.symmetric_group_action_on_values([3,2,1])
[[1, 1, 1, 1], [2, 3], [3]]
```

```
sage: t.symmetric_group_action_on_values([1,3,2])
[[1, 1, 2, 2], [2, 2], [3]]
```

to_chain()

Returns the chain of partitions corresponding to the (semi)standard tableau.

EXAMPLES:

```
sage: Tableau([[1,2],[3],[4]]).to_chain()
[[], [1], [2], [2, 1], [2, 1, 1]]
sage: Tableau([[1,1],[2]]).to_chain()
[[], [2], [2, 1]]
sage: Tableau([[1,1],[3]]).to_chain()
[[], [2], [2], [2, 1]]
sage: Tableau([]).to_chain()
[[]]
```

to_list()

EXAMPLES:

```
sage: t = Tableau([[1,2],[3,4]])
sage: l = t.to_list(); l
[[1, 2], [3, 4]]
sage: l[0][0] = 2
sage: t
[[1, 2], [3, 4]]
```

to_permutation()

Returns a permutation with the entries of self obtained by reading self in the reading order.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).to_permutation()
[3, 4, 1, 2]
```

to_word()

An alias for to_word_by_row.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).to_word()
word: 3412
sage: Tableau([[1, 4, 6], [2, 5], [3]]).to_word()
word: 325146
```

to_word_by_column()

Returns the word obtained from a column reading of the tableau t.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).to_word_by_column()
word: 3142
sage: Tableau([[1, 4, 6], [2, 5], [3]]).to_word_by_column()
word: 321546
```

to_word_by_row()

Returns a word obtained from a row reading of the tableau t.

EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).to_word_by_row()
word: 3412
sage: Tableau([[1, 4, 6], [2, 5], [3]]).to_word_by_row()
word: 325146
```

up()
An iterator for all the tableaux that can be obtained from self by adding a box. EXAMPLES:

```
sage: t = Tableau([[1,2]])
sage: [x for x in t.up()]
[[[1, 2, 3]], [[1, 2], [3]]]
```

up_list()
Returns a list of all the tableaux that can be obtained from self by adding a box.
EXAMPLES:

```
sage: t = Tableau([[1,2]])
sage: t.up_list()
[[[1, 2, 3]], [[1, 2], [3]]]
```

vertical_flip()
Returns the tableau obtained by vertically flipping the tableau t. This only works for rectangular tableau.
EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).vertical_flip()
[[3, 4], [1, 2]]
```

weight()
Returns the evaluation of the word from tableau t.
EXAMPLES:

```
sage: Tableau([[1,2],[3,4]]).evaluation()
[1, 1, 1, 1]
```

Tableaux(*n=None*)
Returns the combinatorial class of tableaux. If n is specified, then it returns the combinatorial class of all tableaux of size n.
EXAMPLES:

```
sage: T = Tableaux(); T
Tableaux
sage: [[1,2],[3,4]] in T
True
sage: [[1,2],[3]] in T
True
sage: [1,2,3] in T
False

sage: T = Tableaux(4); T
Tableaux of size 4
sage: [[1,2],[3,4]] in T
True
sage: [[1,2],[3]] in T
False
sage: [1,2,3] in T
False
```

class **Tableaux_all**()

list()
TESTS:

```
sage: Tableaux().list()
...
NotImplementedError
```

class Tableaux_n(*n*)

list()

TESTS:

```
sage: Tableaux(3).list()
...
NotImplementedError
```

from_chain(*chain*)

Returns a semistandard tableau from a chain of partitions.

EXAMPLES:

```
sage: from sage.combinat.tableau import from_chain
sage: from_chain([[], [2], [2, 1], [3, 2, 1]])
[[1, 1, 3], [2, 3], [3]]
```

from_shape_and_word(*shape, w*)

Returns a tableau from a shape and word.

EXAMPLES:

```
sage: from sage.combinat.tableau import from_shape_and_word
sage: t = Tableau([[1, 3], [2], [4]])
sage: shape = t.shape(); shape
[2, 1, 1]
sage: word  = t.to_word(); word
word: 4213
sage: from_shape_and_word(shape, word)
[[1, 3], [2], [4]]
```

symmetric_group_action_on_values(*word, perm*)

EXAMPLES:

```
sage: from sage.combinat.tableau import symmetric_group_action_on_values
sage: symmetric_group_action_on_values([1,1,1],[1,3,2])
[1, 1, 1]
sage: symmetric_group_action_on_values([1,1,1],[2,1,3])
[2, 2, 2]
sage: symmetric_group_action_on_values([1,2,1],[2,1,3])
[2, 2, 1]
sage: symmetric_group_action_on_values([2,2,2],[2,1,3])
[1, 1, 1]
sage: symmetric_group_action_on_values([2,1,2],[2,1,3])
[2, 1, 1]
sage: symmetric_group_action_on_values([2,2,3,1,1,2,2,3],[1,3,2])
[2, 3, 3, 1, 1, 2, 3, 3]
sage: symmetric_group_action_on_values([2,1,1],[2,1])
[2, 1, 2]
sage: symmetric_group_action_on_values([2,2,1],[2,1])
[1, 2, 1]
sage: symmetric_group_action_on_values([1,2,1],[2,1])
[2, 2, 1]
```

unmatched_places(*w, open, close*)

EXAMPLES:

```
sage: from sage.combinat.tableau import unmatched_places
sage: unmatched_places([2,2,2,1,1,1],2,1)
([], [])
sage: unmatched_places([1,1,1,2,2,2],2,1)
([0, 1, 2], [3, 4, 5])
sage: unmatched_places([], 2, 1)
([], [])
sage: unmatched_places([1,2,4,6,2,1,5,3],2,1)
([0], [1])
sage: unmatched_places([2,2,1,2,4,6,2,1,5,3], 2, 1)
([], [0, 3])
sage: unmatched_places([3,1,1,1,2,1,2], 2, 1)
([1, 2, 3], [6])
```

17.32.2 Skew Tableaux

SemistandardSkewTableaux(*p=None, mu=None*)

Returns a combinatorial class of semistandard skew tableaux.

EXAMPLES:

```
sage: SemistandardSkewTableaux()
Semistandard skew tableaux
```

```
sage: SemistandardSkewTableaux(3)
Semistandard skew tableaux of size 3
```

```
sage: SemistandardSkewTableaux([[2,1],[]])
Semistandard skew tableaux of shape [[2, 1], []]
```

```
sage: SemistandardSkewTableaux([[2,1],[]],[2,1])
Semistandard skew tableaux of shape [[2, 1], []] and weight [2, 1]
```

```
sage: SemistandardSkewTableaux(3, [2,1])
Semistandard skew tableaux of size 3 and weight [2, 1]
```

class **SemistandardSkewTableaux_all**()

class **SemistandardSkewTableaux_n**(*n*)

cardinality()

EXAMPLES:

```
sage: SemistandardSkewTableaux(2).cardinality()
8
```

class **SemistandardSkewTableaux_nmu**(*n, mu*)

cardinality()

EXAMPLES:

```
sage: SemistandardSkewTableaux(2,[1,1]).cardinality()
4
```

class SemistandardSkewTableaux_p(*p*)

cardinality()
EXAMPLES:

```
sage: SemistandardSkewTableaux([[2,1],[]]).cardinality()
8
```

class SemistandardSkewTableaux_pmu(*p, mu*)

list()
EXAMPLES:

```
sage: SemistandardSkewTableaux([[2,1],[]],[2,1]).list()
[[[1, 1], [2]]]
```

SkewTableau(*st=None, expr=None*)
Returns the skew tableau object corresponding to st.

Note that Sage uses the English convention for partitions and tableaux.

EXAMPLES:

```
sage: st = SkewTableau([[None, 1],[2,3]]); st
[[None, 1], [2, 3]]
sage: st.inner_shape()
[1]
sage: st.outer_shape()
[2, 2]
```

The expr form of a skew tableau consists of the inner partition followed by a list of the entries in row from bottom to top.

```
sage: SkewTableau(expr=[[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

class SkewTableau_class(*t*)

boxes()
Returns on the entries in self with content c.
EXAMPLES:

```
sage: s = SkewTableau([[None,1,2],[3],[6]])
sage: s.boxes()
[(0, 1), (0, 2), (1, 0), (2, 0)]
```

boxes_by_content(*c*)
Returns the coordinates of the boxes in self with content c.

```
sage: s = SkewTableau([[None,1,2],[3,4,5],[6]])
sage: s.boxes_by_content(0)
[(1, 1)]
sage: s.boxes_by_content(1)
[(0, 1), (1, 2)]
sage: s.boxes_by_content(2)
```

```
[(0, 2)]
sage: s.boxes_by_content(-1)
[(1, 0)]
sage: s.boxes_by_content(-2)
[(2, 0)]
```

conjugate()
Returns the conjugate of the skew tableau.
EXAMPLES:

```
sage: SkewTableau([[None,1],[2,3]]).conjugate()
[[None, 2], [1, 3]]
```

entries_by_content(*c*)
Returns on the entries in self with content c.
EXAMPLES:

```
sage: s = SkewTableau([[None,1,2],[3,4,5],[6]])
sage: s.entries_by_content(0)
[4]
sage: s.entries_by_content(1)
[1, 5]
sage: s.entries_by_content(2)
[2]
sage: s.entries_by_content(-1)
[3]
sage: s.entries_by_content(-2)
[6]
```

evaluation()
Returns the evaluation of the word from skew tableau.
EXAMPLES:

```
sage: SkewTableau([[1,2],[3,4]]).evaluation()
[1, 1, 1, 1]
```

filling()
Returns a list of the non-empty entries in self.
EXAMPLES:

```
sage: t = SkewTableau([[None,1],[2,3]])
sage: t.filling()
[[1], [2, 3]]
```

inner_shape()
Returns the inner shape of the tableau.
EXAMPLES:

```
sage: SkewTableau([[None,1,2],[None,3],[4]]).inner_shape()
[1, 1]
```

inner_size()
Returns the size of the inner shape of the skew tableau.
EXAMPLES:

```
sage: SkewTableau([[None, 2, 4], [None, 3], [1]]).inner_size()
2
sage: SkewTableau([[None, 2], [1, 3]]).inner_size()
1
```

is_ribbon()

Returns True if and only if self is a ribbon, that is if it has no 2x2 boxes.

EXAMPLES:

```
sage: SkewTableau([[None,1],[2,3]]).is_ribbon()
True
sage: SkewTableau([[None,1,2],[3,4,5]]).is_ribbon()
False
```

is_semistandard()

Returns True if self is a semistandard skew tableau and False otherwise.

EXAMPLES:

```
sage: SkewTableau([[None, 2, 2], [1, 3]]).is_semistandard()
True
sage: SkewTableau([[None, 2], [2, 4]]).is_semistandard()
True
sage: SkewTableau([[None, 3], [2, 4]]).is_semistandard()
True
sage: SkewTableau([[None, 2], [1, 2]]).is_semistandard()
False
```

is_standard()

Returns True if self is a standard skew tableau and False otherwise.

EXAMPLES:

```
sage: SkewTableau([[None, 2], [1, 3]]).is_standard()
True
sage: SkewTableau([[None, 2], [2, 4]]).is_standard()
False
sage: SkewTableau([[None, 3], [2, 4]]).is_standard()
False
sage: SkewTableau([[None, 2], [2, 4]]).is_standard()
False
```

outer_shape()

Returns the outer shape of the tableau.

EXAMPLES:

```
sage: SkewTableau([[None,1,2],[None,3],[4]]).outer_shape()
[3, 2, 1]
```

outer_size()

Returns the size of the outer shape of the skew tableau.

EXAMPLES:

```
sage: SkewTableau([[None, 2, 4], [None, 3], [1]]).outer_size()
6
sage: SkewTableau([[None, 2], [1, 3]]).outer_size()
4
```

pp()

Returns a pretty print string of the tableau.

EXAMPLES:

```
sage: SkewTableau([[None,2,3],[None,4],[5]]).pp()
  .  2  3
  .  4
  5
```

rectify()

Returns a Tableau formed by applying the jeu de taquin process to self.

Fulton, William. 'Young Tableaux'. p15

EXAMPLES:

```
sage: s = SkewTableau([[None,1],[2,3]])
sage: s.rectify()
[[1, 3], [2]]
```

restrict(*n*)

Returns the restriction of the (semi)standard skew tableau to all the numbers less than or equal to n.

EXAMPLES:

```
sage: SkewTableau([[None,1],[2],[3]]).restrict(2)
[[None, 1], [2]]
sage: SkewTableau([[None,1],[2],[3]]).restrict(1)
[[None, 1]]
sage: SkewTableau([[None,1],[1],[2]]).restrict(1)
[[None, 1], [1]]
```

shape()

Returns the shape of a tableau t.

EXAMPLES:

```
sage: SkewTableau([[None,1,2],[None,3],[4]]).shape()
[[3, 2, 1], [1, 1]]
```

size()

Returns the number of boxes in the skew tableau.

EXAMPLES:

```
sage: SkewTableau([[None, 2, 4], [None, 3], [1]]).size()
4
sage: SkewTableau([[None, 2], [1, 3]]).size()
3
```

slide(*corner=None*)

Fulton, William. 'Young Tableaux'. p12-13

EXAMPLES:

```
sage: st = SkewTableau([[None, None, None, None,2],[None, None, None, None,6], [None, 2, 4,
sage: st.slide([2,0])
[[None, None, None, None, 2], [None, None, None, None, 6], [2, 2, 4, 4], [3, 5, 6], [5]]
```

to_chain()

Returns the chain of partitions corresponding to the (semi)standard skew tableau.

EXAMPLES:

```
sage: SkewTableau([[None,1],[2],[3]]).to_chain()
[[1], [2], [2, 1], [2, 1, 1]]
sage: SkewTableau([[None,1],[1],[2]]).to_chain()
[[1], [2, 1], [2, 1, 1]]
```

to_expr()

The first list in a result corresponds to the inner partition of the skew shape. The second list is a list of the rows in the skew tableau read from the bottom up.

Provided for compatibility with MuPAD-Combinat. In MuPAD-Combinat, if t is a skew tableau, then to_expr gives the same result as expr(t) would give in MuPAD-Combinat.

EXAMPLES:

```
sage: SkewTableau([[None,1,1,3],[None,2,2],[1]]).to_expr()
[[1, 1], [[1], [2, 2], [1, 1, 3]]]
sage: SkewTableau([]).to_expr()
[[], []]
```

to_ribbon()

Returns the ribbon version of self.

EXAMPLES:

```
sage: SkewTableau([[None,1],[2,3]]).to_ribbon()
[[1], [2, 3]]
```

to_tableau()

Returns a tableau with the same filling. This only works if the inner shape of the skew tableau has size zero.

EXAMPLES:

```
sage: SkewTableau([[1,2],[3,4]]).to_tableau()
[[1, 2], [3, 4]]
```

to_word()

An alias for SkewTableau.to_word_by_row().

EXAMPLES:

```
sage: SkewTableau([[None,1],[2,3]]).to_word()
word: 231
sage: SkewTableau([[None, 2, 4], [None, 3], [1]]).to_word()
word: 1324
```

to_word_by_column()

Returns the word obtained from a column reading of the skew tableau

EXAMPLES:

```
sage: s = SkewTableau([[None,1],[2,3]])
sage: s.pp()
  .  1
  2  3
sage: s.to_word_by_column()
word: 132

sage: s = SkewTableau([[None, 2, 4], [None, 3], [1]])
sage: s.pp()
.  2  4
.  3
1
sage: s.to_word_by_column()
word: 4231
```

to_word_by_row()

Returns a word obtained from a row reading of the skew tableau.

EXAMPLES:

```
sage: s = SkewTableau([[None,1],[2,3]])
sage: s.pp()
  .  1
  2  3
sage: s.to_word_by_row()
word: 231
sage: s = SkewTableau([[None, 2, 4], [None, 3], [1]])
```

```
sage: s.pp()
  .  2  4
  .  3
  1
sage: s.to_word_by_row()
word: 1324
```

weight()

Returns the evaluation of the word from skew tableau.

EXAMPLES:

```
sage: SkewTableau([[1,2],[3,4]]).evaluation()
[1, 1, 1, 1]
```

StandardSkewTableaux(*skp=None*)

Returns the combinatorial class of standard skew tableaux of shape skp (where skp is a skew partition).

EXAMPLES:

```
sage: StandardSkewTableaux([[3, 2, 1], [1, 1]]).list()
[[[None, 1, 2], [None, 3], [4]],
 [[None, 1, 2], [None, 4], [3]],
 [[None, 1, 3], [None, 2], [4]],
 [[None, 1, 4], [None, 2], [3]],
 [[None, 1, 3], [None, 4], [2]],
 [[None, 1, 4], [None, 3], [2]],
 [[None, 2, 3], [None, 4], [1]],
 [[None, 2, 4], [None, 3], [1]]]
```

class **StandardSkewTableaux_all**()

class **StandardSkewTableaux_n**(*n*)

cardinality()

EXAMPLES:

```
sage: StandardSkewTableaux(1).cardinality()
1
sage: StandardSkewTableaux(2).cardinality()
4
sage: StandardSkewTableaux(3).cardinality()
24
sage: StandardSkewTableaux(4).cardinality()
194
```

class **StandardSkewTableaux_skewpartition**(*skp*)

cardinality()

Returns the number of standard skew tableaux with shape of the skew partition skp.

EXAMPLES:

```
sage: StandardSkewTableaux([[3, 2, 1], [1, 1]]).cardinality()
8
```

list()

Returns a list for all the standard skew tableaux with shape of the skew partition skp. The standard skew tableaux are ordered lexicographically by the word obtained from their row reading.

EXAMPLES:

```
sage: StandardSkewTableaux([[3, 2, 1], [1, 1]]).list()
[[[None, 1, 2], [None, 3], [4]],
 [[None, 1, 2], [None, 4], [3]],
 [[None, 1, 3], [None, 2], [4]],
 [[None, 1, 4], [None, 2], [3]],
 [[None, 1, 3], [None, 4], [2]],
 [[None, 1, 4], [None, 3], [2]],
 [[None, 2, 3], [None, 4], [1]],
 [[None, 2, 4], [None, 3], [1]]]
```

object_class()

from_expr(*expr*)

Returns a SkewTableau from a MuPAD-Combinat expr for a skew tableau. The first list in expr is the inner shape of the skew tableau. The second list are the entries in the rows of the skew tableau from bottom to top.

Provided primarily for compatibility with MuPAD-Combinat.

EXAMPLES:

```
sage: import sage.combinat.skew_tableau as skew_tableau
sage: sage.combinat.skew_tableau.from_expr([[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

from_shape_and_word(*shape, word*)

Returns the skew tableau corresponding to the skew partition shape and the word obtained from the row reading.

EXAMPLES:

```
sage: import sage.combinat.skew_tableau as skew_tableau
sage: t = SkewTableau([[None, 1, 3], [None, 2], [4]])
sage: shape = t.shape()
sage: word  = t.to_word()
sage: skew_tableau.from_shape_and_word(shape, word)
[[None, 1, 3], [None, 2], [4]]
```

17.32.3 Ribbons

Ribbon(*r*)

Returns a ribbon tableau object.

A ribbon is a skew tableau that does not contain a 2x2 box. A ribbon is given by a list of the rows from top to bottom.

EXAMPLES:

```
sage: Ribbon([[2,3],[1,4,5]])
[[2, 3], [1, 4, 5]]
sage: Ribbon([[2,3],[1,4,5]]).to_skew_tableau()
[[None, None, 2, 3], [1, 4, 5]]
```

class **Ribbon_class**(*l*)

evaluation()

Returns the evaluation of the word from ribbon.

EXAMPLES:

```
sage: Ribbon([[1,2],[3,4]]).evaluation()
[1, 1, 1, 1]
```

height()
Returns the height of the ribbon.
EXAMPLES:

```
sage: Ribbon([[2,3],[1,4,5]]).height()
2
```

is_standard()
Returns True is the ribbon is standard and False otherwise. ribbon are standard if they are filled with the numbers 1...size and they are increasing along both rows and columns.
EXAMPLES:

```
sage: Ribbon([[2,3],[1,4,5]]).is_standard()
True
sage: Ribbon([[2,2],[1,4,5]]).is_standard()
False
sage: Ribbon([[4,5],[1,2,3]]).is_standard()
False
```

ribbon_shape()
Returns the ribbon shape. The ribbon shape is given just by the number of boxes in each row.
EXAMPLES:

```
sage: Ribbon([[2,3],[1,4,5]]).ribbon_shape()
[2, 3]
```

shape()
Returns the skew partition corresponding to the shape of the ribbon.
EXAMPLES:

```
sage: Ribbon([[2,3],[1,4,5]]).shape()
[[4, 3], [2]]
```

size()
Returns the size (number of boxes) in the ribbon.
EXAMPLES:

```
sage: Ribbon([[2,3],[1,4,5]]).size()
5
```

spin()
Returns the spin of self.
EXAMPLES:

```
sage: Ribbon([[2,3],[1,4,5]]).spin()
1/2
```

to_permutation()
Returns the permutation corresponding to the ribbon tableau.
EXAMPLES:

```
sage: r = Ribbon([[1], [2,3], [4, 5, 6]])
sage: r.to_permutation()
[4, 5, 6, 2, 3, 1]
```

to_skew_tableau()
Returns the skew tableau corresponding to the ribbon tableau.

EXAMPLES:

```
sage: Ribbon([[2,3],[1,4,5]]).to_skew_tableau()
[[None, None, 2, 3], [1, 4, 5]]
```

to_word()

An alias for Ribbon.to_word_by_row().

EXAMPLES:

```
sage: Ribbon([[1],[2,3]]).to_word_by_row()
word: 231
sage: Ribbon([[2, 4], [3], [1]]).to_word_by_row()
word: 1324
```

to_word_by_column()

Returns the word obtained from a column reading of the ribbon

EXAMPLES:

```
sage: Ribbon([[1],[2,3]]).to_word_by_column()
word: 132
sage: Ribbon([[2, 4], [3], [1]]).to_word_by_column()
word: 4231
```

to_word_by_row()

Returns a word obtained from a row reading of the ribbon.

EXAMPLES:

```
sage: Ribbon([[1],[2,3]]).to_word_by_row()
word: 231
sage: Ribbon([[2, 4], [3], [1]]).to_word_by_row()
word: 1324
```

width()

Returns the width of the ribbon.

EXAMPLES:

```
sage: Ribbon([[2,3],[1,4,5]]).width()
4
```

StandardRibbons(*shape*)

Returns the combinatorial class of standard ribbon tableaux of shape shape.

EXAMPLES:

```
sage: StandardRibbons([2,2])
Standard ribbon tableaux of shape [2, 2]
sage: StandardRibbons([2,2]).first()
[[2, 4], [1, 3]]
sage: StandardRibbons([2,2]).last()
[[1, 2], [3, 4]]
sage: StandardRibbons([2,2]).cardinality()
5
sage: StandardRibbons([2,2]).list()
[[[2, 4], [1, 3]],
 [[2, 3], [1, 4]],
 [[1, 4], [2, 3]],
 [[1, 3], [2, 4]],
 [[1, 2], [3, 4]]]
sage: StandardRibbons([3,2,2]).cardinality()
155
```

class **StandardRibbons_shape**(*shape*)

first()
Returns the first standard ribbon of ribbon shape shape.
EXAMPLES:

```
sage: StandardRibbons([2,2]).first()
[[2, 4], [1, 3]]
```

last()
Returns the first standard ribbon of ribbon shape shape.
EXAMPLES:

```
sage: StandardRibbons([2,2]).last()
[[1, 2], [3, 4]]
```

from_permutation(*p*)
Returns a standard ribbon of size len(p) from a Permutation p. The lengths of each row are given by the distance between the descents of the permutation p.
EXAMPLES:

```
sage: import sage.combinat.ribbon as ribbon
sage: [ribbon.from_permutation(p) for p in Permutations(3)]
[[[1, 2, 3]], [[2], [1, 3]], [[1, 3], [2]], [[1], [2, 3]], [[1, 2], [3]], [[1], [2], [3]]]
```

from_shape_and_word(*shape, word*)
Returns the ribbon corresponding to the given ribbon shape and word.
EXAMPLES:

```
sage: import sage.combinat.ribbon as ribbon
sage: ribbon.from_shape_and_word([2,3],[1,2,3,4,5])
[[1, 2], [3, 4, 5]]
```

17.32.4 Ribbon Tableaux

MultiSkewTableau(*x*)
Returns a multi skew tableau object which is a tuple of skew tableau.
EXAMPLES:

```
sage: s = MultiSkewTableau([ [[None,1],[2,3]], [[1,2],[2]] ])
sage: s.size()
6
sage: s.weight()
[2, 3, 1]
sage: s.shape()
[[[2, 2], [1]], [[2, 1], []]]
```

class **MultiSkewTableau_class**(*l*)

inversion_pairs()
Returns a list of the inversion pairs of self.
EXAMPLES:

```
sage: s = MultiSkewTableau([ [[2,3],[5,5]], [[1,1],[3,3]], [[2],[6]] ])
sage: s.inversion_pairs()
[((0, (0, 0)), (1, (0, 0))),
 ((0, (1, 0)), (1, (0, 1))),
 ((0, (1, 1)), (1, (0, 0))),
 ((0, (1, 1)), (1, (1, 1))),
 ((0, (1, 1)), (2, (0, 0))),
 ((1, (0, 1)), (2, (0, 0))),
 ((1, (1, 1)), (2, (0, 0)))]
```

inversions()

Returns the number of inversion pairs of self.

EXAMPLES:

```
sage: t1 = SkewTableau([[1]])
sage: t2 = SkewTableau([[2]])
sage: MultiSkewTableau([t1,t1]).inversions()
0
sage: MultiSkewTableau([t1,t2]).inversions()
0
sage: MultiSkewTableau([t2,t2]).inversions()
0
sage: MultiSkewTableau([t2,t1]).inversions()
1
sage: s = MultiSkewTableau([ [[2,3],[5,5]], [[1,1],[3,3]], [[2],[6]] ])
sage: s.inversions()
7
```

shape()

Returns the shape of self.

EXAMPLES:

```
sage: s = SemistandardSkewTableaux([[2,2],[1]]).list()
sage: a = MultiSkewTableau([s[0],s[1],s[2]])
sage: a.shape()
[[[2, 2], [1]], [[2, 2], [1]], [[2, 2], [1]]]
```

size()

Returns the size of self, which is the sum of the sizes of the skew tableaux in self.

EXAMPLES:

```
sage: s = SemistandardSkewTableaux([[2,2],[1]]).list()
sage: a = MultiSkewTableau([s[0],s[1],s[2]])
sage: a.size()
9
```

weight()

Returns the weight of self.

EXAMPLES:

```
sage: s = SemistandardSkewTableaux([[2,2],[1]]).list()
sage: a = MultiSkewTableau([s[0],s[1],s[2]])
sage: a.weight()
[5, 3, 1]
```

RibbonTableau(*rt=None, expr=None*)

Returns a ribbon tableau object.

EXAMPLES:

```
sage: rt = RibbonTableau([[None, 1],[2,3]]); rt
[[None, 1], [2, 3]]
sage: rt.inner_shape()
[1]
sage: rt.outer_shape()
[2, 2]

sage: RibbonTableau(expr=[[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
```

class RibbonTableau_class(*t*)

evaluation()

Returns the evaluation of the ribbon tableau

EXAMPLES:

```
sage: RibbonTableau([[0, 0, 3, 0], [1, 1, 0], [2, 0, 4]]).evaluation()
[2, 1, 1, 1]
```

length()

Returns the length of the ribbons into a ribbon tableau.

EXAMPLES:

```
sage: RibbonTableau([[None, 1],[2,3]]).length()
1
sage: RibbonTableau([[1,0],[2,0]]).length()
2
```

to_word()

Returns a word obtained from a row reading of self.

EXAMPLES:

```
sage: R = RibbonTableau([[0, 0, 3, 0], [1, 1, 0], [2, 0, 4]])
sage: R.to_word()
word: 2041100030
```

RibbonTableaux(*shape, weight, length*)

Returns the combinatorial class of ribbon tableaux of skew shape shape and weight weight tiled by ribbons of length length.

EXAMPLES:

```
sage: RibbonTableaux([[2,1],[]],[1,1,1],1)
Ribbon tableaux of shape [[2, 1], []] and weight [1, 1, 1] with 1-ribbons
```

class RibbonTableaux_shapeweightlength(*shape, weight, length*)

cardinality()

EXAMPLES:

```
sage: RibbonTableaux([[2,1],[]],[1,1,1],1).cardinality()
2
sage: RibbonTableaux([[2,2],[]],[1,1],2).cardinality()
2
sage: RibbonTableaux([[4,3,3],[]],[2,1,1,1],2).cardinality()
5
```

TESTS:

```
sage: RibbonTableaux([6,6,6], [4,2], 3).cardinality()
6
sage: RibbonTableaux([3,3,3,2,1], [3,1], 3).cardinality()
1
sage: RibbonTableaux([3,3,3,2,1], [2,2], 3).cardinality()
2
sage: RibbonTableaux([3,3,3,2,1], [2,1,1], 3).cardinality()
5
sage: RibbonTableaux([3,3,3,2,1], [1,1,1,1], 3).cardinality()
12
sage: RibbonTableaux([5,4,3,2,1],[2,2,1],3).cardinality()
10

sage: RibbonTableaux([8,7,6,5,1,1],[3,2,2,1],3).cardinality()
85
sage: RibbonTableaux([5,4,3,2,1,1,1],[2,2,1],3).cardinality()
10

sage: RibbonTableaux([7,7,7,2,1,1],[3,2,0,1,1],3).cardinality()
25
```

Weights with some zeros in the middle and end

```
sage: RibbonTableaux([3,3,3], [0,1,0,2,0], 3).cardinality()
3
sage: RibbonTableaux([3,3,3],[1,0,1,0,1,0,0,0],3).cardinality()
6
```

list()

EXAMPLES:

```
sage: RibbonTableaux([[2,1],[]],[1,1,1],1).list()
[[[1, 3], [2]], [[1, 2], [3]]]
sage: RibbonTableaux([[2,2],[]],[1,1],2).list()
[[[0, 0], [1, 2]], [[1, 0], [2, 0]]]
```

SemistandardMultiSkewTableaux(*shape, weight*)

Returns the combinatorial class of semistandard multi skew tableaux. A multi skew tableau is a k-tuple of skew tableaux of givens shape with a specified total weight.

EXAMPLES:

```
sage: s = SemistandardMultiSkewTableaux([ [[2,1],[]], [[2,2],[1]] ], [2,2,2]); s
Semistandard multi skew tableaux of shape [[[2, 1], []], [[2, 2], [1]]] and weight [2, 2, 2]
sage: s.list()
[[[[1, 1], [2]], [[None, 2], [3, 3]]],
 [[[1, 2], [2]], [[None, 1], [3, 3]]],
 [[[1, 3], [2]], [[None, 2], [1, 3]]],
 [[[1, 3], [2]], [[None, 1], [2, 3]]],
 [[[1, 1], [3]], [[None, 2], [2, 3]]],
 [[[1, 2], [3]], [[None, 2], [1, 3]]],
 [[[1, 2], [3]], [[None, 1], [2, 3]]],
 [[[2, 2], [3]], [[None, 1], [1, 3]]],
 [[[1, 3], [3]], [[None, 1], [2, 2]]],
 [[[2, 3], [3]], [[None, 1], [1, 2]]]]
```

class **SemistandardMultiSkewTtableaux_shapeweight**(*shape, weight*)

list()

EXAMPLES:

```
sage: sp = SkewPartitions(3).list()
sage: SemistandardMultiSkewTableaux([sp[0], sp[-1]],[2,2,2]).list()
[[[[1], [2], [3]], [[1, 2, 3]]]]

sage: a = SkewPartition([[8,7,6,5,1,1],[2,1,1]])
sage: weight = [3,3,2]
sage: k = 3
sage: s = SemistandardMultiSkewTableaux(a.r_quotient(k),weight)
sage: len(s.list())
34
sage: RibbonTableaux(a,weight,k).cardinality()
34
```

cospin_polynomial (*part, weight, length*)

Returns the cospin polynomial associated to part, weight, and length.

EXAMPLES:

```
sage: from sage.combinat.ribbon_tableau import cospin_polynomial
sage: cospin_polynomial([6,6,6],[4,2],3)
t^4 + t^3 + 2*t^2 + t + 1
sage: cospin_polynomial([3,3,3,2,1], [3,1], 3)
1
sage: cospin_polynomial([3,3,3,2,1], [2,2], 3)
t + 1
sage: cospin_polynomial([3,3,3,2,1], [2,1,1], 3)
t^2 + 2*t + 2
sage: cospin_polynomial([3,3,3,2,1], [1,1,1,1], 3)
t^3 + 3*t^2 + 5*t + 3
sage: cospin_polynomial([5,4,3,2,1,1,1], [2,2,1], 3)
2*t^2 + 6*t + 2
sage: cospin_polynomial([[6]*6, [3,3]], [4,4,2], 3)
3*t^4 + 6*t^3 + 9*t^2 + 5*t + 3
```

count_rec (*nexts, current, part, weight, length*)

INPUT:

- `nexts, current, part` - skew partitions
- `weight` - non-negative integer list
- `length` - integer

TESTS:

```
sage: from sage.combinat.ribbon_tableau import count_rec
sage: count_rec([], [], [[2, 1, 1], []], [2], 2)
[0]
sage: count_rec([[0], [1]], [[[2, 1, 1], [0, 0, 2, 0]], [[4], [2, 0, 0, 0]]], [[4, 1, 1], []], [
[1]
sage: count_rec([], [[[], [2, 2]]], [[2, 2], []], [2], 2)
[1]
sage: count_rec([], [[[], [2, 0, 2, 0]]], [[4], []], [2], 2)
[1]
sage: count_rec([[1], [1]], [[[2, 2], [0, 0, 2, 0]], [[4], [2, 0, 0, 0]]], [[4, 2], []], [2, 1],
[2]
sage: count_rec([[1], [1], [2]], [[[2, 2, 2], [0, 0, 2, 0]], [[4, 1, 1], [0, 2, 0, 0]], [[4, 2],
[4]
```

```
sage: count_rec([[4], [1]], [[[4, 2, 2], [0, 0, 2, 0]], [[4, 3, 1], [0, 2, 0, 0]]], [[4, 3, 3],
[5]
```

from_expr(*l*)

Returns a RibbonTableau from a MuPAD-Combinat expr for a skew tableau. The first list in expr is the inner shape of the skew tableau. The second list are the entries in the rows of the skew tableau from bottom to top.

Provided primarily for compatibility with MuPAD-Combinat.

EXAMPLES:

```
sage: import sage.combinat.ribbon_tableau as ribbon_tableau
sage: sage.combinat.ribbon_tableau.from_expr([[1,1],[[5],[3,4],[1,2]]])
[[None, 1, 2], [None, 3, 4], [5]]
sage: type(_)
<class 'sage.combinat.ribbon_tableau.RibbonTableau_class'>
```

graph_implementation_rec(*skp, weight, length, function*)

TESTS:

```
sage: from sage.combinat.ribbon_tableau import graph_implementation_rec, list_rec
sage: graph_implementation_rec(SkewPartition([[1], []]), [1], 1, list_rec)
[[[], [[1]]]]
sage: graph_implementation_rec(SkewPartition([[2, 1], []]), [1, 2], 1, list_rec)
[[[], [[2], [1, 2]]]]
sage: graph_implementation_rec(SkewPartition([[], []]), [0], 1, list_rec)
[[[], []]]
```

insertion_tableau(*skp, perm, evaluation, tableau, length*)

INPUT:

- `skp` - skew partitions
- `perm, evaluation` - non-negative integers
- `tableau` - skew tableau
- `length` - integer

TESTS:

```
sage: from sage.combinat.ribbon_tableau import insertion_tableau
sage: insertion_tableau([[1], []], [1], 1, [[], []], 1)
[[], [[1]]]
sage: insertion_tableau([[2, 1], []], [1, 1], 2, [[], [[1]]], 1)
[[], [[2], [1, 2]]]
sage: insertion_tableau([[2, 1], []], [0, 0], 3, [[], [[2], [1, 2]]], 1)
[[], [[2], [1, 2]]]
sage: insertion_tableau([[1, 1], []], [1], 2, [[], [[1]]], 1)
[[], [[2], [1]]]
sage: insertion_tableau([[2], []], [0, 1], 2, [[], [[1]]], 1)
[[], [[1, 2]]]
sage: insertion_tableau([[2, 1], []], [0, 1], 3, [[], [[2], [1]]], 1)
[[], [[2], [1, 3]]]
sage: insertion_tableau([[1, 1], []], [2], 1, [[], []], 2)
[[], [[1], [0]]]
sage: insertion_tableau([[2], []], [2, 0], 1, [[], []], 2)
[[], [[1, 0]]]
sage: insertion_tableau([[2, 2], []], [0, 2], 2, [[], [[1], [0]]], 2)
[[], [[1, 2], [0, 0]]]
```

```
sage: insertion_tableau([[2, 2], []], [2, 0], 2, [[], [[1, 0]]], 2)
[[], [[2, 0], [1, 0]]]
sage: insertion_tableau([[2, 2], [1]], [3, 0], 1, [[], []], 3)
[[1], [[1, 0], [0]]]
```

list_rec (*nexts, current, part, weight, length*)

INPUT:

- `nexts, current, part` - skew partitions
- `weight` - non-negative integer list
- `length` - integer

TESTS:

```
sage: from sage.combinat.ribbon_tableau import list_rec
sage: list_rec([], [[[], [1]]], [[1], []], [1], 1)
[[[], [[1]]]]
sage: list_rec([[[[], [[1]]]]], [[[1], [1, 1]]], [[2, 1], []], [1, 2], 1)
[[[], [[2], [1, 2]]]]
sage: list_rec([], [[[1], [3, 0]]], [[2, 2], [1]], [1], 3)
[[[1], [[1, 0], [0]]]]
sage: list_rec([[[[], [[2]]]]], [[[1], [1, 1]]], [[2, 1], []], [0, 1, 2], 1)
[[[], [[3], [2, 3]]]]
sage: list_rec([], [[[], [2]]], [[1, 1], []], [1], 2)
[[[], [[1], [0]]]]
sage: list_rec([], [[[], [2, 0]]], [[2], []], [1], 2)
[[[], [[1, 0]]]]
sage: list_rec([[[[], [[1], [0]]]]], [[[], [[1, 0]]]]], [[[1, 1], [0, 2]], [[2], [2, 0]]], [[2, 2
[[[], [[1, 2], [0, 0]]], [[], [[2, 0], [1, 0]]]]
sage: list_rec([], [[[], [2, 2]]], [[2, 2], []], [2], 2)
[[[], [[1, 1], [0, 0]]]]
sage: list_rec([], [[[], [1, 1]]], [[2], []], [2], 1)
[[[], [[1, 1]]]]
sage: list_rec([[[[], [[1, 1]]]]], [[[2], [1, 1]]], [[2, 2], []], [2, 2], 1)
[[[], [[2, 2], [1, 1]]]]
```

spin_polynomial (*part, weight, length*)

Returns the spin polynomial associated to part, weight, and length.

EXAMPLES:

```
sage: from sage.combinat.ribbon_tableau import spin_polynomial
sage: spin_polynomial([6,6,6],[4,2],3)
t^6 + t^5 + 2*t^4 + t^3 + t^2
sage: spin_polynomial([6,6,6],[4,1,1],3)
t^6 + 2*t^5 + 3*t^4 + 2*t^3 + t^2
sage: spin_polynomial([3,3,3,2,1], [2,2], 3)
t^(7/2) + t^(5/2)
sage: spin_polynomial([3,3,3,2,1], [2,1,1], 3)
2*t^(7/2) + 2*t^(5/2) + t^(3/2)
sage: spin_polynomial([3,3,3,2,1], [1,1,1,1], 3)
3*t^(7/2) + 5*t^(5/2) + 3*t^(3/2) + sqrt(t)
sage: spin_polynomial([5,4,3,2,1,1,1], [2,2,1], 3)
2*t^(9/2) + 6*t^(7/2) + 2*t^(5/2)
sage: spin_polynomial([[6]*6, [3,3]], [4,4,2], 3)
3*t^9 + 5*t^8 + 9*t^7 + 6*t^6 + 3*t^5
```

spin_polynomial_square(*part, weight, length*)

Returns the spin polynomial associated with part, weight, and length, with the substitution t -> t^2 made.

EXAMPLES:

```
sage: from sage.combinat.ribbon_tableau import spin_polynomial_square
sage: spin_polynomial_square([6,6,6],[4,2],3)
t^12 + t^10 + 2*t^8 + t^6 + t^4
sage: spin_polynomial_square([6,6,6],[4,1,1],3)
t^12 + 2*t^10 + 3*t^8 + 2*t^6 + t^4
sage: spin_polynomial_square([3,3,3,2,1], [2,2], 3)
t^7 + t^5
sage: spin_polynomial_square([3,3,3,2,1], [2,1,1], 3)
2*t^7 + 2*t^5 + t^3
sage: spin_polynomial_square([3,3,3,2,1], [1,1,1,1], 3)
3*t^7 + 5*t^5 + 3*t^3 + t
sage: spin_polynomial_square([5,4,3,2,1,1,1], [2,2,1], 3)
2*t^9 + 6*t^7 + 2*t^5
sage: spin_polynomial_square([[6]*6, [3,3]], [4,4,2], 3)
3*t^18 + 5*t^16 + 9*t^14 + 6*t^12 + 3*t^10
```

spin_rec(*t, nexts, current, part, weight, length*)

Routine used for constructing the spin polynomial.

INPUT:

- `weight` - list of non-negative integers
- `length` - the length of the ribbons we're tiling with
- `t` - the variable

EXAMPLES:

```
sage: from sage.combinat.ribbon_tableau import spin_rec
sage: sp = SkewPartition
sage: t = ZZ['t'].gen()
sage: spin_rec(t, [], [[[], [3, 3]]], sp([[2, 2, 2], []]), [2], 3)
[t^4]
sage: spin_rec(t, [[0], [t^4]], [[[2, 1, 1, 1, 1], [0, 3]], [[2, 2, 2], [3, 0]]], sp([[2, 2, 2,
[t^5]
sage: spin_rec(t, [], [[[], [3, 3, 0]]], sp([[3, 3], []]), [2], 3)
[t^2]
sage: spin_rec(t, [[t^4], [t^3], [t^2]], [[[2, 2, 2], [0, 0, 3]], [[3, 2, 1], [0, 3, 0]], [[3, 3
[t^6 + t^4 + t^2]
sage: spin_rec(t, [[t^5], [t^4], [t^6 + t^4 + t^2]], [[[2, 2, 2, 2, 1], [0, 0, 3]], [[3, 3, 1, 1
[2*t^7 + 2*t^5 + t^3]
```

17.33 Symmetric Functions

17.33.1 Symmetric Functions

AUTHORS:

- Mike Hansen (2007-06-15)

```
sage: s = SymmetricFunctionAlgebra(QQ, basis='schur')
sage: e = SymmetricFunctionAlgebra(QQ, basis='elementary')
sage: f1 = s([2,1]); f1
s[2, 1]
sage: f2 = e(f1); f2
e[2, 1] - e[3]
sage: f1 == f2
True
sage: f1.expand(3, alphabet=['x','y','z'])
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2
sage: f2.expand(3, alphabet=['x','y','z'])
x^2*y + x*y^2 + x^2*z + 2*x*y*z + y^2*z + x*z^2 + y*z^2


sage: m = SFAMonomial(QQ)
sage: m([3,1])
m[3, 1]
sage: m(4)
4*m[]
sage: m([4])
m[4]
sage: 3*m([3,1])-1/2*m([4])
3*m[3, 1] - 1/2*m[4]
```

Code needs to be added to coerce symmetric polynomials into symmetric functions.

```
sage: p = SFAPower(QQ)
sage: m = p(3)
sage: m
3*p[]
sage: m.parent()
Symmetric Function Algebra over Rational Field, Power symmetric functions as basis
sage: m + p([3,2])
3*p[] + p[3, 2]


sage: s = SFASchur(QQ)
sage: h = SFAHomogeneous(QQ)
sage: P = SFAPower(QQ)
sage: e = SFAElementary(QQ)
sage: m = SFAMonomial(QQ)
sage: a = s([3,1])
sage: s(a)
s[3, 1]
sage: h(a)
h[3, 1] - h[4]
sage: p(a)
1/8*p[1, 1, 1, 1] + 1/4*p[2, 1, 1] - 1/8*p[2, 2] - 1/4*p[4]
sage: e(a)
e[2, 1, 1] - e[2, 2] - e[3, 1] + e[4]
sage: m(a)
3*m[1, 1, 1, 1] + 2*m[2, 1, 1] + m[2, 2] + m[3, 1]
sage: a.expand(4)
x0^3*x1 + x0^2*x1^2 + x0*x1^3 + x0^3*x2 + 2*x0^2*x1*x2 + 2*x0*x1^2*x2 + x1^3*x2 + x0^2*x2^2 + 2*x0*x1


sage: h(m([1]))
h[1]
```

```
sage: h( m([2]) +m([1,1]) )
h[2]
sage: h( m([3]) + m([2,1]) + m([1,1,1]) )
h[3]
sage: h( m([4]) + m([3,1]) + m([2,2]) + m([2,1,1]) + m([1,1,1,1]) )
h[4]
sage: k = 5
sage: h( sum([ m(part) for part in Partitions(k)]) )
h[5]
sage: k = 10
sage: h( sum([ m(part) for part in Partitions(k)]) )
h[10]


sage: P3 = Partitions(3)
sage: P3.list()
[[3], [2, 1], [1, 1, 1]]
sage: m = SFAMonomial(QQ)
sage: f = sum([m(p) for p in P3])
sage: m.get_print_style()
'lex'
sage: f
m[1, 1, 1] + m[2, 1] + m[3]
sage: m.set_print_style('length')
sage: f
m[3] + m[2, 1] + m[1, 1, 1]
sage: m.set_print_style('maximal_part')
sage: f
m[1, 1, 1] + m[2, 1] + m[3]
sage: m.set_print_style('lex')


sage: s = SFASchur(QQ)
sage: m = SFAMonomial(QQ)
sage: m([3])*s([2,1])
2*m[3, 1, 1, 1] + m[3, 2, 1] + 2*m[4, 1, 1] + m[4, 2] + m[5, 1]
sage: s(m([3])*s([2,1]))
s[2, 1, 1, 1, 1] - s[2, 2, 2] - s[3, 3] + s[5, 1]
sage: s(s([2,1])*m([3]))
s[2, 1, 1, 1, 1] - s[2, 2, 2] - s[3, 3] + s[5, 1]
sage: e = SFAElementary(QQ)
sage: e([4])*e([3])*e([1])
e[4, 3, 1]


sage: s = SFASchur(QQ)
sage: z = s([2,1]) + s([1,1,1])
sage: z.coefficient([2,1])
1
sage: z.length()
2
sage: z.support()
[[1, 1, 1], [2, 1]]
sage: z.degree()
3
```

SFAElementary(*R*)

Returns the symmetric function algebra over R with the elementary symmetric functions as the basis.

EXAMPLES:

```
sage: SFAElementary(QQ)
Symmetric Function Algebra over Rational Field, Elementary symmetric functions as basis
```

SFAHomogeneous(*R*)

Returns the symmetric function algebra over R with the Homogeneous symmetric functions as the basis.

EXAMPLES:

```
sage: SFAHomogeneous(QQ)
Symmetric Function Algebra over Rational Field, Homogeneous symmetric functions as basis
```

SFAMonomial(*R*)

Returns the symmetric function algebra over R with the monomial symmetric functions as the basis.

EXAMPLES:

```
sage: SFAMonomial(QQ)
Symmetric Function Algebra over Rational Field, Monomial symmetric functions as basis
```

SFAPower(*R*)

Returns the symmetric function algebra over R with the power-sum symmetric functions as the basis.

EXAMPLES:

```
sage: SFAPower(QQ)
Symmetric Function Algebra over Rational Field, Power symmetric functions as basis
```

SFASchur(*R*)

Returns the symmetric function algebra over R with the Schur symmetric functions as the basis.

EXAMPLES:

```
sage: SFASchur(QQ)
Symmetric Function Algebra over Rational Field, Schur symmetric functions as basis
```

SymmetricFunctionAlgebra(*R*, *basis='schur'*)

Return the free algebra over the ring R on n generators with given names.

INPUT:

- R - ring with identity basis

OUTPUT: A SymmetricFunctionAlgebra

EXAMPLES:

```
sage: SymmetricFunctionAlgebra(QQ)
Symmetric Function Algebra over Rational Field, Schur symmetric functions as basis
```

```
sage: SymmetricFunctionAlgebra(QQ, basis='m')
Symmetric Function Algebra over Rational Field, Monomial symmetric functions as basis
```

```
sage: SymmetricFunctionAlgebra(QQ, basis='power')
Symmetric Function Algebra over Rational Field, Power symmetric functions as basis
```

class **SymmetricFunctionAlgebraElement_generic**(*A*, *x*)

degree()

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1]) + 3
sage: z.degree()
4
```

derivative_with_respect_to_p1(*n=1*)

Returns the symmetric function obtained by taking the derivative of self with respect to the power-sum symmetric function p([1]) when the expansion of self in the power-sum basis is considered as a polynomial in p([1])'s.

EXAMPLES:

```
sage: p = SFAPower(QQ)
sage: a = p([1,1,1])
sage: a.derivative_with_respect_to_p1()
3*p[1, 1]
sage: a.derivative_with_respect_to_p1(1)
3*p[1, 1]
sage: a.derivative_with_respect_to_p1(2)
6*p[1]
sage: a.derivative_with_respect_to_p1(3)
6*p[]

sage: s = SFASchur(QQ)
sage: s([3]).derivative_with_respect_to_p1()
s[2]
sage: s([2,1]).derivative_with_respect_to_p1()
s[1, 1] + s[2]
sage: s([1,1,1]).derivative_with_respect_to_p1()
s[1, 1]
```

expand(*n, alphabet='x'*)

Expands the symmetric function as a symmetric polynomial in n variables.

EXAMPLES:

```
sage: J =JackPolynomialsJ(QQ, t=2)
sage: J([2,1]).expand(3)
4*x0^2*x1 + 4*x0*x1^2 + 4*x0^2*x2 + 6*x0*x1*x2 + 4*x1^2*x2 + 4*x0*x2^2 + 4*x1*x2^2
```

hl_creation_operator(*nu*)

This is the vertex operator that generalizes Jing's operator It is from: Hall-Littlewood Vertex Operators and Kostka Polynomials, Shimizono-Zabrocki, Proposition 5 It is a linear operator that rases the degree by sum(nu) This creation operator is a t-analogue of multiplication by s(nu)

INPUT:

- nu - a partition

EXAMPLES:

```
sage: s = SFASchur(QQ['t'])
sage: s([2]).hl_creation_operator([3,2])
s[3, 2, 2] + t*s[3, 3, 1] + t*s[4, 2, 1] + t^2*s[4, 3] + t^2*s[5, 2]
sage: HLQp = HallLittlewoodQp(QQ)
sage: HLQp(s([2]).hl_creation_operator([2]).hl_creation_operator([3]))
Qp[3, 2, 2]
sage: s([2,2]).hl_creation_operator([2,1])
t*s[2, 2, 2, 1] + t^2*s[3, 2, 1, 1] + t^2*s[3, 2, 2] + t^3*s[3, 3, 1] + t^3*s[4, 2, 1] + t^4
sage: s(1).hl_creation_operator([2,1,1])
```

```
s[2, 1, 1]
sage: s(0).hl_creation_operator([2,1,1])
0
sage: s([3,2]).hl_creation_operator([2,1,1])
(t^2-t)*s[2, 2, 2, 2, 1] + t^3*s[3, 2, 2, 1, 1] + (t^3-t^2)*s[3, 2, 2, 2] + t^3*s[3, 3, 1, 1
```

inner_plethysm(*x*)

Returns the inner plethysm of self with x.

The result of f.inner_plethysm(g) is linear in f and linear in 'homogeneous pieces' of g. So, to describe this function, we assume without loss that f is some Schur function s(la) and g is a homogeneous symmetric function of degree n. The function g can be thought of as the character of an irreducible representation, rho, of the symmetric group S_n. Let N be the dimension of this representation. If the number of parts of la is greater then N, then f.inner_plethysm(g) = 0 by definition. Otherwise, we can interpret f as the character of an irreducible GL_N representation, call it σ. Now $\sigma \circ \rho$ is an S_n representation and, by definition, the character of this representation is f.inner_plethysm(g).

REFERENCES:

- King, R. 'Branching rules for $GL_m \supset \Sigma_n$ and the evaluation of inner plethysms.' J. Math. Phys. 15, 258 (1974)

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: p = SFAPower(QQ)
sage: h = SFAHomogeneous(QQ)
sage: s([2,1]).inner_plethysm(s([1,1,1]))
0
sage: h([2]).inner_plethysm(s([2,1]))
h[2, 1]
sage: s(_)
s[2, 1] + s[3]

sage: f = s([2,1]) + 2*s([3,1])
sage: f.itensor(f)
s[1, 1, 1] + s[2, 1] + 4*s[2, 1, 1] + 4*s[2, 2] + s[3] + 4*s[3, 1] + 4*s[4]
sage: s( h([1,1]).inner_plethysm(f) )
s[1, 1, 1] + s[2, 1] + 4*s[2, 1, 1] + 4*s[2, 2] + s[3] + 4*s[3, 1] + 4*s[4]

sage: s([]).inner_plethysm(s([1,1]) + 2*s([2,1])+s([3]))
s[2] + s[3]
sage: [s([]).inner_plethysm(s(p)) for p in Partitions(4)]
[s[4], s[4], s[4], s[4], s[4]]
```

inner_tensor(*x*)

Returns the inner tensor product of self and x in the basis of self.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2] + 4*s[3, 1, 1, 1]
```

TESTS:

```
sage: s = SFASchur(QQ)
sage: a = s([8,8])
sage: a.itensor(a) #long
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2] + s[6, 6, 2, 2] + s[6, 6, 4] +
```

internal_product(*x*)

Returns the inner tensor product of self and x in the basis of self.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2] + 4*s[3, 1, 1, 1]
```

TESTS:

```
sage: s = SFASchur(QQ)
sage: a = s([8,8])
sage: a.itensor(a) #long
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2] + s[6, 6, 2, 2] + s[6, 6, 4] +
```

is_schur_positive()

Returns True if and only if self is Schur positive. If s is the space of Schur functions over self's base ring, then this is the same as self._is_positive(s).

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: a = s([2,1]) + s([3])
sage: a.is_schur_positive()
True
sage: a = s([2,1]) - s([3])
sage: a.is_schur_positive()
False

sage: QQx = QQ['x']
sage: s = SFASchur(QQx)
sage: x = QQx.gen()
sage: a = (1+x)*s([2,1])
sage: a.is_schur_positive()
True
sage: a = (1-x)*s([2,1])
sage: a.is_schur_positive()
False
```

itensor(*x*)

Returns the inner tensor product of self and x in the basis of self.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
```

```
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2] + 4*s[3, 1, 1, 1]
```

TESTS:

```
sage: s = SFASchur(QQ)
sage: a = s([8,8])
sage: a.itensor(a) #long
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2] + s[6, 6, 2, 2] + s[6, 6, 4] +
```

kronecker_product(*x*)

Returns the inner tensor product of self and x in the basis of self.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: a = s([2,1])
sage: b = s([3])
sage: a.itensor(b)
s[2, 1]
sage: c = s([3,2,1])
sage: c.itensor(c)
s[1, 1, 1, 1, 1, 1] + 2*s[2, 1, 1, 1, 1] + 3*s[2, 2, 1, 1] + 2*s[2, 2, 2] + 4*s[3, 1, 1, 1]
```

TESTS:

```
sage: s = SFASchur(QQ)
sage: a = s([8,8])
sage: a.itensor(a) #long
s[4, 4, 4, 4] + s[5, 5, 3, 3] + s[5, 5, 5, 1] + s[6, 4, 4, 2] + s[6, 6, 2, 2] + s[6, 6, 4] +
```

omega()

Returns the image of self under the Frobenius / omega automorphism. The default implementation converts to the Schurs performs the automorphism and changes back.

EXAMPLES:

```
sage: J = JackPolynomialsP(QQ,1)
sage: a = J([2,1]) + J([1,1,1])
sage: a.omega()
JackP[2, 1] + JackP[3]
```

plethysm(*x, include=None, exclude=None*)

Returns the outer plethysm of self with x.

By default, the degree one elements are the generators for the self's base ring.

INPUT:

- `x` - a symmetric function
- `include` - a list of variables to be treated as degree one elements instead of the default degree one elements
- `exclude` - a list of variables to be excluded from the default degree one elements

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: h = SFAHomogeneous(QQ)
sage: s ( h([3])( h([2]) ) )
s[2, 2, 2] + s[4, 2] + s[6]
sage: p = SFAPower(QQ)
sage: p([3])( s([2,1]) )
1/3*p[3, 3, 3] - 1/3*p[9]
sage: e = SFAElementary(QQ)
sage: e([3])( e([2]) )
e[3, 3] + e[4, 1, 1] - 2*e[4, 2] - e[5, 1] + e[6]
```

```
sage: R.<t> = QQ[]
sage: s = SFASchur(R);
sage: a = s([3])
sage: f = t*s([2])
sage: a(f)
t^3*s[2, 2, 2] + t^3*s[4, 2] + t^3*s[6]
sage: f(a)
t*s[4, 2] + t*s[6]
```

restrict_degree(*d, exact=True*)

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.restrict_degree(2)
0
sage: z.restrict_degree(1)
s[1]
sage: z.restrict_degree(3)
s[1, 1, 1] + s[2, 1]
sage: z.restrict_degree(3, exact=False)
s[1] + s[1, 1, 1] + s[2, 1]
```

restrict_partition_lengths(*l, exact=True*)

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.restrict_partition_lengths(2)
s[2, 1]
sage: z.restrict_partition_lengths(2, exact=False)
s[1] + s[2, 1] + s[4]
```

restrict_parts(*n*)

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: z = s([4]) + s([2,1]) + s([1,1,1]) + s([1])
sage: z.restrict_parts(2)
s[1] + s[1, 1, 1] + s[2, 1]
sage: z.restrict_parts(1)
s[1] + s[1, 1, 1]
```

scalar(*x*)

Returns standard scalar product between self and s.

This is the default implementation that converts both self and x into Schur functions and performs the scalar product that basis.

EXAMPLES:

```
sage: e = SFAElementary(QQ)
sage: h = SFAHomogeneous(QQ)
sage: m = SFAMonomial(QQ)
sage: p4 = Partitions(4)
sage: matrix([ [e(a).scalar(h(b)) for a in p4] for b in p4])
[ 0  0  0  0  1]
[ 0  0  0  1  4]
[ 0  0  1  2  6]
[ 0  1  2  5 12]
[ 1  4  6 12 24]
```

```
sage: matrix([ [h(a).scalar(e(b)) for a in p4] for b in p4])
[ 0  0  0  0  1]
[ 0  0  0  1  4]
[ 0  0  1  2  6]
[ 0  1  2  5 12]
[ 1  4  6 12 24]
sage: matrix([ [m(a).scalar(e(b)) for a in p4] for b in p4])
[-1  2  1 -3  1]
[ 0  1  0 -2  1]
[ 0  0  1 -2  1]
[ 0  0  0 -1  1]
[ 0  0  0  0  1]
sage: matrix([ [m(a).scalar(h(b)) for a in p4] for b in p4])
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

scalar_hl (*x, t=None*)

Returns the standard Hall-Littlewood scalar product of self and x.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: a = s([2,1])
sage: sp = a.scalar_t(a); sp
(-t^2 - 1)/(t^5 - 2*t^4 + t^3 - t^2 + 2*t - 1)
sage: sp.parent()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

scalar_t (*x, t=None*)

Returns the standard Hall-Littlewood scalar product of self and x.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: a = s([2,1])
sage: sp = a.scalar_t(a); sp
(-t^2 - 1)/(t^5 - 2*t^4 + t^3 - t^2 + 2*t - 1)
sage: sp.parent()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

skew_by (*x*)

Returns the element whose result is the dual to multiplication by x applied to self.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: s([3,2]).skew_by(s([2]))
s[2, 1] + s[3]
sage: s([3,2]).skew_by(s([1,1,1]))
0
sage: s([3,2,1]).skew_by(s([2,1]))
s[1, 1, 1] + 2*s[2, 1] + s[3]

sage: p = SFAPower(QQ)
sage: p([4,3,3,2,2,1]).skew_by(p([2,1]))
4*p[4, 3, 3, 2]
sage: zee = sage.combinat.sf.sfa.zee
sage: zee([4,3,3,2,2,1])/zee([4,3,3,2])
4
```

theta(*a*)

Returns the image of self under the theta automorphism which sends $p[k]$ to $a * p[k]$.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: s([2,1]).theta(2)
2*s[1, 1, 1] + 6*s[2, 1] + 2*s[3]
sage: p = SFAPower(QQ)
sage: p([2]).theta(2)
2*p[2]
```

theta_qt(*q, t*)

Returns the image of self under the theta automorphism which sends $p[k]$ to $(1-q^k)/(1-t^k) * p[k]$.

EXAMPLES:

```
sage: QQqt = QQ['q,t'].fraction_field()
sage: q,t = QQqt.gens()
sage: p = SFAPower(QQqt)
sage: p([2]).theta_qt(q,t)
((-q^2+1)/(-t^2+1))*p[2]
sage: p([2,1]).theta_qt(q,t)
((q^3-q^2-q+1)/(t^3-t^2-t+1))*p[2, 1]
```

class **SymmetricFunctionAlgebra_generic**(*R, element_class=None*)

basis_name()

Returns the name of the basis of self.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: s.basis_name()
'schur'
sage: p = SFAPower(QQ)
sage: p.basis_name()
'power'
sage: h = SFAHomogeneous(QQ)
sage: h.basis_name()
'homogeneous'
sage: e = SFAElementary(QQ)
sage: e.basis_name()
'elementary'
sage: m = SFAMonomial(QQ)
sage: m.basis_name()
'monomial'
```

dual_basis(*scalar=None, scalar_name='', prefix=None*)

Returns the dual basis of self with respect to the scalar product scalar. If scalar is None, then the standard (Hall) scalar product is used.

EXAMPLES: The duals of the elementary symmetric functions with respect to the Hall scalar product are the forgotten symmetric functions.

```
sage: e = SFAElementary(QQ)
sage: f = e.dual_basis(prefix='f'); f
Dual basis to Symmetric Function Algebra over Rational Field, Elementary symmetric functions
sage: f([2,1])^2
4*f[2, 2, 1, 1] + 6*f[2, 2, 2] + 2*f[3, 2, 1] + 2*f[3, 3] + 2*f[4, 1, 1] + f[4, 2]
sage: f([2,1]).scalar(e([2,1]))
1
```

```
sage: f([2,1]).scalar(e([1,1,1]))
0
```

Since the power-sum symmetric functions are orthogonal, their duals with respect to the Hall scalar product are scalar multiples of themselves.

```
sage: p = SFAPower(QQ)
sage: q = p.dual_basis(prefix='q'); q
Dual basis to Symmetric Function Algebra over Rational Field, Power symmetric functions as b
sage: q([2,1])^2
4*q[2, 2, 1, 1]
sage: p([2,1]).scalar(q([2,1]))
1
sage: p([2,1]).scalar(q([1,1,1]))
0
```

get_print_style()

Returns the value of the current print style for self.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: s.get_print_style()
'lex'
sage: s.set_print_style('length')
sage: s.get_print_style()
'length'
sage: s.set_print_style('lex')
```

prefix()

Returns the prefix on the elements of self.

EXAMPLES:

```
sage: schur = SFASchur(QQ)
sage: schur([3,2,1])
s[3, 2, 1]
sage: schur.prefix()
's'
```

set_print_style(*ps*)

Set the value of the current print style to ps.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: s.get_print_style()
'lex'
sage: s.set_print_style('length')
sage: s.get_print_style()
'length'
sage: s.set_print_style('lex')
```

transition_matrix(*basis, n*)

Returns the transitions matrix between self and basis for the homogenous component of degree n.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: m = SFAMonomial(QQ)
sage: s.transition_matrix(m,5)
[1 1 1 1 1 1 1]
[0 1 1 2 2 3 4]
```

```
[0 0 1 1 2 3 5]
[0 0 0 1 1 3 6]
[0 0 0 0 1 2 5]
[0 0 0 0 0 1 4]
[0 0 0 0 0 0 1]

sage: p = SFAPower(QQ)
sage: s.transition_matrix(p, 4)
[ 1/4  1/3  1/8  1/4 1/24]
[-1/4    0 -1/8  1/4  1/8]
[   0 -1/3  1/4    0 1/12]
[ 1/4    0 -1/8 -1/4  1/8]
[-1/4  1/3  1/8 -1/4 1/24]
sage: StoP = s.transition_matrix(p,4)
sage: a = s([3,1])+5*s([1,1,1,1])-s([4])
sage: a
5*s[1, 1, 1, 1] + s[3, 1] - s[4]
sage: mon = a.support()
sage: coeffs = a.coefficients()
sage: coeffs
[5, 1, -1]
sage: mon
[[1, 1, 1, 1], [3, 1], [4]]
sage: cm = matrix([[-1,1,0,0,5]])
sage: cm * StoP
[-7/4  4/3  3/8 -5/4 7/24]
sage: p(a)
7/24*p[1, 1, 1, 1] - 5/4*p[2, 1, 1] + 3/8*p[2, 2] + 4/3*p[3, 1] - 7/4*p[4]

sage: h = SFAHomogeneous(QQ)
sage: e = SFAElementary(QQ)
sage: s.transition_matrix(m,7) == h.transition_matrix(s,7).transpose()
True

sage: h.transition_matrix(m, 7) == h.transition_matrix(m, 7).transpose()
True

sage: h.transition_matrix(e, 7) == e.transition_matrix(h, 7)
True

sage: p.transition_matrix(s, 5)
[ 1 -1  0  1  0 -1  1]
[ 1  0 -1  0  1  0 -1]
[ 1 -1  1  0 -1  1 -1]
[ 1  1 -1  0 -1  1  1]
[ 1  0  1 -2  1  0  1]
[ 1  2  1  0 -1 -2 -1]
[ 1  4  5  6  5  4  1]

sage: e.transition_matrix(m,7) == e.transition_matrix(m,7).transpose()
True
```

is_SymmetricFunction(*x*)

Returns True if x is a symmetric function.

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import is_SymmetricFunction
sage: s = SFASchur(QQ)
```

```
sage: is_SymmetricFunction(2)
False
sage: is_SymmetricFunction(s(2))
True
sage: is_SymmetricFunction(s([2,1]))
True
```

is_SymmetricFunctionAlgebra(*x*)

Return True if x is a symmetric function algebra; otherwise, return False.

EXAMPLES:

```
sage: sage.combinat.sf.sfa.is_SymmetricFunctionAlgebra(5)
False
sage: sage.combinat.sf.sfa.is_SymmetricFunctionAlgebra(ZZ)
False
sage: sage.combinat.sf.sfa.is_SymmetricFunctionAlgebra(SymmetricFunctionAlgebra(ZZ,'schur'))
True
```

zee(*part*)

Returns the size of the centralizer of permutations of cycle type part. Note that this is the inner product between p(part) and itself where p is the power-sum symmetric functions.

INPUT:

- `part` - an integer partition (for example, [2,1,1])

EXAMPLES:

```
sage: from sage.combinat.sf.sfa import zee
sage: zee([2,1,1])
4
```

17.33.2 Classical symmetric functions.

class SymmetricFunctionAlgebraElement_classical(*A, x*)

A symmetric function.

class SymmetricFunctionAlgebra_classical(*R, basis, element_class, prefix*)

is_commutative()

Return True if this symmetric function algebra is commutative.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: s.is_commutative()
True
```

is_field()

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: s.is_field()
False
```

init()

Set up the conversion functions between the classical bases.

EXAMPLES:

```
sage: from sage.combinat.sf.classical import init
sage: sage.combinat.sf.classical.conversion_functions = {}
sage: init()
sage: sage.combinat.sf.classical.conversion_functions[('schur', 'power')]
<built-in function t_SCHUR_POWSYM_symmetrica>
```

17.33.3 Schur symmetric functions

class SymmetricFunctionAlgebraElement_schur(*A*, *x*)

expand(*n*, *alphabet='x'*)

Expands the symmetric function as a symmetric polynomial in n variables.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: a = s([2,1])
sage: a.expand(2)
x0^2*x1 + x0*x1^2
sage: a.expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
sage: a.expand(4)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 2*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2 + x0^2*x3 + 2*x0*x1*x
sage: a.expand(2, alphabet='y')
y0^2*y1 + y0*y1^2
sage: a.expand(2, alphabet=['a','b'])
a^2*b + a*b^2
sage: s([1,1,1,1]).expand(3)
0
```

omega()

Returns the image of self under the Frobenius / omega automorphism.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: s([2,1]).omega()
s[2, 1]
sage: s([2,1,1]).omega()
s[3, 1]
```

scalar(*x*)

Returns the standard scalar product between self and x.

Note that the Schur functions are self-dual with respect to this scalar product. They are also lower-triangularly related to the monomial symmetric functions with respect to this scalar product.

EXAMPLES:

```
sage: s = SFASchur(ZZ)
sage: a = s([2,1])
sage: b = s([1,1,1])
sage: c = 2*s([1,1,1])
sage: d = a + b
sage: a.scalar(a)
1
sage: b.scalar(b)
1
sage: b.scalar(a)
```

```
0
sage: b.scalar(c)
2
sage: c.scalar(c)
4
sage: d.scalar(a)
1
sage: d.scalar(b)
1
sage: d.scalar(c)
2

sage: m = SFAMonomial(ZZ)
sage: p4 = Partitions(4)
sage: l = [ [s(p).scalar(m(q)) for q in p4] for p in p4]
sage: matrix(l)
[ 1  0  0  0  0]
[-1  1  0  0  0]
[ 0 -1  1  0  0]
[ 1 -1 -1  1  0]
[-1  2  1 -3  1]
```

class **SymmetricFunctionAlgebra_schur**(*R*)

dual_basis(*scalar=None, scalar_name='', prefix=None*)

The dual basis to the Schur basis with respect to the standard scalar product is the Schur basis since it is self-dual.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: ds = s.dual_basis()
sage: s is ds
True
```

is_schur_basis()

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: s.is_schur_basis()
True
```

17.33.4 Monomial symmetric functions

class **SymmetricFunctionAlgebraElement_monomial**(*A, x*)

expand(*n, alphabet='x'*)

Expands the symmetric function as a symmetric polynomial in n variables.

EXAMPLES:

```
sage: m = SFAMonomial(QQ)
sage: m([2,1]).expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
sage: m([1,1,1]).expand(2)
0
sage: m([2,1]).expand(3,alphabet='z')
z0^2*z1 + z0*z1^2 + z0^2*z2 + z1^2*z2 + z0*z2^2 + z1*z2^2
```

```
sage: m([2,1]).expand(3,alphabet='x,y,z')
x^2*y + x*y^2 + x^2*z + y^2*z + x*z^2 + y*z^2
```

class SymmetricFunctionAlgebra_monomial(*R*)

dual_basis(*scalar=None, scalar_name='', prefix=None*)

The dual basis of the monomial basis with respect to the standard scalar product is the homogeneous basis.

EXAMPLES:

```
sage: m = SFAMonomial(QQ)
sage: h = SFAHomogeneous(QQ)
sage: m.dual_basis() == h
True
```

17.33.5 Multiplicative symmetric functions

class SymmetricFunctionAlgebra_multiplicative(*R, basis, element_class, prefix*)

17.33.6 Elementary symmetric functions

class SymmetricFunctionAlgebraElement_elementary(*A, x*)

expand(*n, alphabet='x'*)

Expands the symmetric function as a symmetric polynomial in n variables.

EXAMPLES:

```
sage: e = SFAElementary(QQ)
sage: e([2,1]).expand(3)
x0^2*x1 + x0*x1^2 + x0^2*x2 + 3*x0*x1*x2 + x1^2*x2 + x0*x2^2 + x1*x2^2
sage: e([1,1,1]).expand(2)
x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + x1^3
sage: e([3]).expand(2)
0
sage: e([2]).expand(3)
x0*x1 + x0*x2 + x1*x2
sage: e([3]).expand(4,alphabet='x,y,z,t')
x*y*z + x*y*t + x*z*t + y*z*t
sage: e([3]).expand(4,alphabet='y')
y0*y1*y2 + y0*y1*y3 + y0*y2*y3 + y1*y2*y3
```

omega()

Returns the image of self under the Frobenius / omega automorphism.

EXAMPLES:

```
sage: e = SFAElementary(QQ)
sage: a = e([2,1]); a
e[2, 1]
sage: a.omega()
e[1, 1, 1] - e[2, 1]

sage: h = SFAHomogeneous(QQ)
sage: h(e([2,1]).omega())
h[2, 1]
```

class SymmetricFunctionAlgebra_elementary(*R*)

17.33.7 Homogenous symmetric functions

class SymmetricFunctionAlgebraElement_homogeneous(*A*, *x*)

expand(*n*, *alphabet='x'*)

Expands the symmetric function as a symmetric polynomial in n variables.

EXAMPLES:

```
sage: h = SFAHomogeneous(QQ)
sage: h([3]).expand(2)
x0^3 + x0^2*x1 + x0*x1^2 + x1^3
sage: h([1,1,1]).expand(2)
x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + x1^3
sage: h([2,1]).expand(3)
x0^3 + 2*x0^2*x1 + 2*x0*x1^2 + x1^3 + 2*x0^2*x2 + 3*x0*x1*x2 + 2*x1^2*x2 + 2*x0*x2^2 + 2*x1*
sage: h([3]).expand(2,alphabet='y')
y0^3 + y0^2*y1 + y0*y1^2 + y1^3
sage: h([3]).expand(2,alphabet='x,y')
x^3 + x^2*y + x*y^2 + y^3
sage: h([3]).expand(3,alphabet='x,y,z')
x^3 + x^2*y + x*y^2 + y^3 + x^2*z + x*y*z + y^2*z + x*z^2 + y*z^2 + z^3
```

omega()

Returns the image of self under the Frobenius / omega automorphism.

EXAMPLES:

```
sage: h = SFAHomogeneous(QQ)
sage: a = h([2,1]); a
h[2, 1]
sage: a.omega()
h[1, 1, 1] - h[2, 1]
sage: e = SFAElementary(QQ)
sage: e(h([2,1]).omega())
e[2, 1]
```

class SymmetricFunctionAlgebra_homogeneous(*R*)

dual_basis(*scalar=None*, *prefix=None*)

The dual basis of the homogeneous basis with respect to the standard scalar product is the monomial basis.

EXAMPLES:

```
sage: m = SFAMonomial(QQ)
sage: h = SFAHomogeneous(QQ)
sage: h.dual_basis() == m
True
```

17.33.8 Power-sum symmetric functions

class SymmetricFunctionAlgebraElement_power(*A*, *x*)

expand(*n*, *alphabet='x'*)

Expands the symmetric function as a symmetric polynomial in n variables.

EXAMPLES:

```
sage: p = SFAPower(QQ)
sage: a = p([2])
sage: a.expand(2)
x0^2 + x1^2
sage: a.expand(3, alphabet=['a','b','c'])
a^2 + b^2 + c^2
sage: p([2,1,1]).expand(2)
x0^4 + 2*x0^3*x1 + 2*x0^2*x1^2 + 2*x0*x1^3 + x1^4
sage: p([7]).expand(4)
x0^7 + x1^7 + x2^7 + x3^7
sage: p([7]).expand(4,alphabet='t')
t0^7 + t1^7 + t2^7 + t3^7
sage: p([7]).expand(4,alphabet='x,y,z,t')
x^7 + y^7 + z^7 + t^7
```

omega()

Returns the image of self under the Frobenius / omega automorphism.

EXAMPLES:

```
sage: p = SFAPower(QQ)
sage: a = p([2,1]); a
p[2, 1]
sage: a.omega()
-p[2, 1]
```

scalar(x)

Returns the standard scalar product of self and x.

Note that the power-sum symmetric functions are orthogonal under this scalar product. The value of $\langle p_\lambda, p_\lambda \rangle$ is given by the size of the centralizer in S_n of a permutation of cycle type λ.

EXAMPLES:

```
sage: p = SFAPower(QQ)
sage: p4 = Partitions(4)
sage: matrix([ [p(a).scalar(p(b)) for a in p4] for b in p4])
[ 4  0  0  0  0]
[ 0  3  0  0  0]
[ 0  0  8  0  0]
[ 0  0  0  4  0]
[ 0  0  0  0 24]
```

class **SymmetricFunctionAlgebra_power**(R)

17.33.9 Generic dual bases symmetric functions

class **SymmetricFunctionAlgebraElement_dual**(A, *dictionary=None*, *dual=None*)

dual()

Returns self in the dual basis.

EXAMPLES:

```
sage: m = SFAMonomial(QQ)
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(scalar=zee)
sage: a = h([2,1])
sage: a.dual()
3*m[1, 1, 1] + 2*m[2, 1] + m[3]
```

expand(*n, alphabet='x'*)

EXAMPLES:

```
sage: m = SFAMonomial(QQ)
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(zee)
sage: a = h([2,1])+h([3])
sage: a.expand(2)
2*x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + 2*x1^3
sage: a.dual().expand(2)
2*x0^3 + 3*x0^2*x1 + 3*x0*x1^2 + 2*x1^3
sage: a.expand(2,alphabet='y')
2*y0^3 + 3*y0^2*y1 + 3*y0*y1^2 + 2*y1^3
sage: a.expand(2,alphabet='x,y')
2*x^3 + 3*x^2*y + 3*x*y^2 + 2*y^3
```

omega()

Returns the image of self under the Frobenius / omega automorphism.

EXAMPLES:

```
sage: m = SFAMonomial(QQ)
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(zee)
sage: hh = SFAHomogeneous(QQ)
sage: hh([2,1]).omega()
h[1, 1, 1] - h[2, 1]
sage: h([2,1]).omega()
d_m[1, 1, 1] - d_m[2, 1]
```

scalar(*x*)

Returns the standard scalar product of self and x.

EXAMPLES:

```
sage: m = SFAMonomial(QQ)
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(scalar=zee)
sage: a = h([2,1])
sage: a.scalar(a)
2
```

scalar_hl(*x*)

Returns the Hall-Littlewood scalar product of self and x.

EXAMPLES:

```
sage: m = SFAMonomial(QQ)
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(scalar=zee)
sage: a = h([2,1])
sage: a.scalar_hl(a)
(t + 2)/(-t^4 + 2*t^3 - 2*t + 1)
```

class SymmetricFunctionAlgebra_dual(*dual_basis, scalar, scalar_name='', prefix=None*)

dual_basis(*scalar=None, scalar_name='', prefix=None*)

Return the dual basis to self. If a the scalar option is not passed, then it returns the dual basis with respect to the scalar product used to define self.

EXAMPLES:

```
sage: m = SFAMonomial(QQ)
sage: zee = sage.combinat.sf.sfa.zee
sage: h = m.dual_basis(scalar=zee)
sage: h.dual_basis()
Symmetric Function Algebra over Rational Field, Monomial symmetric functions as basis
sage: m2 = h.dual_basis(zee, prefix='m2')
sage: m([2])^2
2*m[2, 2] + m[4]
sage: m2([2])^2
2*m2[2, 2] + m2[4]
```

transition_matrix(*basis, n*)

Returns the transition matrix between the n^{th} homogeneous component of self and basis.

EXAMPLES:

```
sage: s = SFASchur(QQ)
sage: e = SFAElementary(QQ)
sage: f = e.dual_basis()
sage: f.transition_matrix(s, 5)
[ 1 -1  0  1  0 -1  1]
[-2  1  1 -1 -1  1  0]
[-2  2 -1 -1  1  0  0]
[ 3 -1 -1  1  0  0  0]
[ 3 -2  1  0  0  0  0]
[-4  1  0  0  0  0  0]
[ 1  0  0  0  0  0  0]
sage: e.transition_matrix(s, 5).inverse().transpose()
[ 1 -1  0  1  0 -1  1]
[-2  1  1 -1 -1  1  0]
[-2  2 -1 -1  1  0  0]
[ 3 -1 -1  1  0  0  0]
[ 3 -2  1  0  0  0  0]
[-4  1  0  0  0  0  0]
[ 1  0  0  0  0  0  0]
```

17.33.10 Symmetric functions defined by orthogonality and triangularity.

One characterization of Schur functions is that they are upper triangularly related to the monomial symmetric functions and orthogonal with respect to the Hall scalar product. We can use the class SymmetricFunctionAlgebra_orthotriang to obtain the Schur functions from this definition.

```
sage: from sage.combinat.sf.sfa import zee
sage: from sage.combinat.sf.orthotriang import SymmetricFunctionAlgebra_orthotriang
sage: m = SFAMonomial(QQ)
sage: s =  SymmetricFunctionAlgebra_orthotriang(QQ, m, zee, 's', 'Schur functions')
sage: s([2,1])^2
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2]


sage: s2 = SFASchur(QQ)
sage: s2([2,1])^2
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2]
```

class SymmetricFunctionAlgebraElement_orthotriang(*A, x*)

class SymmetricFunctionAlgebra_orthotriang(*R, base, scalar, prefix, name, leading_coeff=None*)

17.33.11 Kostka-Foulkes Polynomials

Based on the algorithms in John Stembridge's SF package for Maple which can be found at http://www.math.lsa.umich.edu/~jrs/maple.html .

KostkaFoulkesPolynomial(*mu, nu, t=None*)

Returns the Kostka-Foulkes polynomial $K_{\mu,\nu}(t)$.

EXAMPLES:

```
sage: KostkaFoulkesPolynomial([2,2],[2,2])
1
sage: KostkaFoulkesPolynomial([2,2],[4])
0
sage: KostkaFoulkesPolynomial([2,2],[1,1,1,1])
t^4 + t^2
sage: KostkaFoulkesPolynomial([2,2],[2,1,1])
t
sage: q = PolynomialRing(QQ,'q').gen()
sage: KostkaFoulkesPolynomial([2,2],[2,1,1],q)
q
```

compat(*n, mu, nu*)

Generate all possible partitions of n that can precede μ, ν in a rigging sequence.

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: compat(4, [1], [2,1])
[[1, 1, 1, 1], [2, 1, 1], [2, 2], [3, 1], [4]]
sage: compat(3, [1], [2,1])
[[1, 1, 1], [2, 1], [3]]
sage: compat(2, [1], [])
[[2]]
sage: compat(3, [1], [])
[[2, 1], [3]]
sage: compat(3, [2], [1])
[[3]]
sage: compat(4, [1,1], [])
[[2, 2], [3, 1], [4]]
sage: compat(4, [2], [])
[[4]]
```

dom(*mu, snu*)

Returns True if `sum(mu[:i+1]) >= snu[i]` for all 0 <= i < len(snu); otherwise, it returns False.

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: dom([3,2,1],[2,4,5])
True
sage: dom([3,2,1],[2,4,7])
False
sage: dom([3,2,1],[2,6,5])
False
sage: dom([3,2,1],[4,4,4])
False
```

kfpoly(*mu, nu, t=None*)

Returns the Kostka-Foulkes polynomial $K_{\mu,\nu}(t)$ by generating all rigging sequences for the shape μ, and then selecting those of content ν.

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import kfpoly
sage: kfpoly([2,2], [2,1,1])
t
sage: kfpoly([4], [2,1,1])
t^3
sage: kfpoly([4], [2,2])
t^2
sage: kfpoly([1,1,1,1], [2,2])
0
```

q_bin(*a, b, t=None*)

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: t = PolynomialRing(ZZ, 't').gen()
sage: q_bin(4,2, t)
t^8 + t^7 + 2*t^6 + 2*t^5 + 3*t^4 + 2*t^3 + 2*t^2 + t + 1
sage: q_bin(4,3, t)
t^12 + t^11 + 2*t^10 + 3*t^9 + 4*t^8 + 4*t^7 + 5*t^6 + 4*t^5 + 4*t^4 + 3*t^3 + 2*t^2 + t + 1
```

riggings(*part*)

Generate all possible rigging sequences for a fixed sage.combinat.partition.

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: riggings([3])
[[[1, 1, 1]], [[2, 1]], [[3]]]
sage: riggings([2,1])
[[[2, 1], [1]], [[3], [1]]]
sage: riggings([1,1,1])
[[[3], [2], [1]]]
sage: riggings([2,2])
[[[2, 2], [1, 1]], [[3, 1], [1, 1]], [[4], [1, 1]], [[4], [2]]]
sage: riggings([2,2,2])
[[[3, 3], [2, 2], [1, 1]],
 [[4, 2], [2, 2], [1, 1]],
 [[5, 1], [2, 2], [1, 1]],
 [[6], [2, 2], [1, 1]],
 [[5, 1], [3, 1], [1, 1]],
 [[6], [3, 1], [1, 1]],
 [[6], [4], [2]]]
```

schur_to_hl(*mu, t=None*)

Returns a dictionary corresponding to s_μ in Hall-Littlewood P basis.

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: schur_to_hl([1,1,1])
{[1, 1, 1]: 1}
sage: a = schur_to_hl([2,1])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1] t^2 + t
```

```
[2, 1] 1
sage: a = schur_to_hl([3])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1] t^3
[2, 1] t
[3] 1
sage: a = schur_to_hl([4])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1] t^6
[2, 1, 1] t^3
[2, 2] t^2
[3, 1] t
[4] 1
sage: a = schur_to_hl([3,1])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1] t^5 + t^4 + t^3
[2, 1, 1] t^2 + t
[2, 2] t
[3, 1] 1
sage: a = schur_to_hl([2,2])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1] t^4 + t^2
[2, 1, 1] t
[2, 2] 1
sage: a = schur_to_hl([2,1,1])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1] t^3 + t^2 + t
[2, 1, 1] 1
sage: a = schur_to_hl([1,1,1,1])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1] 1
sage: a = schur_to_hl([2,2,2])
sage: for m,c in sorted(a.iteritems()): print m, c
[1, 1, 1, 1, 1, 1] t^9 + t^7 + t^6 + t^5 + t^3
[2, 1, 1, 1, 1] t^4 + t^2
[2, 2, 1, 1] t
[2, 2, 2] 1
```

weight(*rg*, *t=None*)

Returns the weight of a rigging.

EXAMPLES:

```
sage: from sage.combinat.sf.kfpoly import *
sage: t = PolynomialRing(ZZ, 't').gen()
sage: weight([[2,1], [1]], t)
1
sage: weight([[3],[1]], t)
t^2 + t
sage: weight([[2,1],[3]], t)
t^4
sage: weight([[2, 2], [1, 1]],t)
1
sage: weight([[3, 1], [1, 1]],t)
t
sage: weight([[4], [1, 1]],t)
t^4
sage: weight([[4], [2]],t)
```

```
t^2
```

17.33.12 Hall-Littlewood Polynomials

class HallLittlewoodElement_generic(*A*, *x*)

expand(*n*, *alphabet='x'*)

Expands the symmetric function as a symmetric polynomial in n variables.

EXAMPLES:

```
sage: HLP  = HallLittlewoodP(QQ)
sage: HLQ  = HallLittlewoodQ(QQ)
sage: HLQp = HallLittlewoodQp(QQ)
sage: HLP([2]).expand(2)
x0^2 + (-t + 1)*x0*x1 + x1^2
sage: HLQ([2]).expand(2)
(-t + 1)*x0^2 + (t^2 - 2*t + 1)*x0*x1 + (-t + 1)*x1^2
sage: HLQp([2]).expand(2)
x0^2 + x0*x1 + x1^2
```

scalar(*x*)

Returns standard scalar product between self and s.

This is the default implementation that converts both self and x into Schur functions and performs the scalar product that basis.

EXAMPLES:

```
sage: HLP  = HallLittlewoodP(QQ)
sage: HLQ  = HallLittlewoodQ(QQ)
sage: HLQp = HallLittlewoodQp(QQ)
sage: HLP([2]).scalar(HLQp([2]))
1
sage: HLP([2]).scalar(HLQp([1,1]))
0
```

scalar_hl(*x*, *t=None*)

Returns the standard Hall-Littlewood scalar product of self and x.

EXAMPLES:

```
sage: HLP  = HallLittlewoodP(QQ)
sage: HLQ  = HallLittlewoodQ(QQ)
sage: HLQp = HallLittlewoodQp(QQ)
sage: HLP([2]).scalar_hl(HLQ([2]))
1
sage: HLP([2]).scalar_hl(HLQ([1,1]))
0
```

class HallLittlewoodElement_p(*A*, *x*)

class HallLittlewoodElement_q(*A*, *x*)

class HallLittlewoodElement_qp(*A*, *x*)

HallLittlewoodP(*R*, *t=None*)

Returns the algebra of symmetric functions in Hall-Littlewood P basis. This is the same as the HL basis in John Stembridge's SF examples file.

If t is not specified, then the base ring will be obtained by making the univariate polynomial ring over R with the variable t and taking its fraction field.

EXAMPLES:

```
sage: HallLittlewoodP(QQ)
Hall-Littlewood polynomials in the P basis over Fraction Field of Univariate Polynomial Ring in
sage: HallLittlewoodP(QQ, t=-1)
Hall-Littlewood polynomials in the P basis with t=-1 over Rational Field
sage: HLP = HallLittlewoodP(QQ)
sage: s = SFASchur(HLP.base_ring())
sage: s(HLP([2,1]))
(-t^2-t)*s[1, 1, 1] + s[2, 1]
```

The Hall-Littlewood polynomials in the P basis at $t = 0$ are the Schur functions.

```
sage: HLP = HallLittlewoodP(QQ,t=0)
sage: s = SFASchur(HLP.base_ring())
sage: s(HLP([2,1])) == s([2,1])
True
```

The Hall-Littlewood polynomials in the P basis at $t = 1$ are the monomial symmetric functions.

```
sage: HLP = HallLittlewoodP(QQ,t=1)
sage: m = SFAMonomial(HLP.base_ring())
sage: m(HLP([2,2,1])) == m([2,2,1])
True
```

HallLittlewoodQ(*R, t=None*)

Returns the algebra of symmetric functions in Hall-Littlewood Q basis. This is the same as the Q basis in John Stembridge's SF examples file.

If t is not specified, then the base ring will be obtained by making the univariate polynomial ring over R with the variable t and taking its fraction field.

EXAMPLES:

```
sage: HallLittlewoodQ(QQ)
Hall-Littlewood polynomials in the Q basis over Fraction Field of Univariate Polynomial Ring in
sage: HallLittlewoodQ(QQ, t=-1)
Hall-Littlewood polynomials in the Q basis with t=-1 over Rational Field
```

HallLittlewoodQp(*R, t=None*)

Returns the algebra of symmetric functions in Hall-Littlewood Q' (Qp) basis. This is dual to the Hall-Littlewood P basis with respect to the standard scalar product.

If t is not specified, then the base ring will be obtained by making the univariate polynomial ring over R with the variable t and taking its fraction field.

EXAMPLES:

```
sage: HallLittlewoodQp(QQ)
Hall-Littlewood polynomials in the Qp basis over Fraction Field of Univariate Polynomial Ring in
sage: HallLittlewoodQp(QQ, t=-1)
Hall-Littlewood polynomials in the Qp basis with t=-1 over Rational Field
```

class **HallLittlewood_generic**(*R, t=None*)

transition_matrix(*basis, n*)

Returns the transitions matrix between self and basis for the homogenous component of degree n.

EXAMPLES:

```
sage: HLP = HallLittlewoodP(QQ)
sage: s   = SFASchur(HLP.base_ring())
sage: HLP.transition_matrix(s, 4)
[             1             -t              0            t^2           -t^3]
[             0              1             -t             -t      t^3 + t^2]
[             0              0              1             -t             t^3]
[             0              0              0              1 -t^3 - t^2 - t]
[             0              0              0              0              1]
sage: HLQ = HallLittlewoodQ(QQ)
sage: HLQ.transition_matrix(s,3)
[                         -t + 1                       t^2 - t                      -t^3 + t^
[                              0             t^2 - 2*t + 1           -t^4 + t^3 + t^2 -
[                              0                        0 -t^6 + t^5 + t^4 - t^2 - t +
sage: HLQp = HallLittlewoodQp(QQ)
sage: HLQp.transition_matrix(s,3)
[      1       0       0]
[      t       1       0]
[    t^3 t^2 + t       1]
```

class HallLittlewood_p(*R*, *t=None*)

class HallLittlewood_q(*R*, *t=None*)

class HallLittlewood_qp(*R*, *t=None*)

17.33.13 Jack Polynomials

class JackPolynomial_generic(*A*, *x*)

scalar_jack(*x*)
EXAMPLES:

```
sage: P = JackPolynomialsP(QQ)
sage: Q = JackPolynomialsQ(QQ)
sage: p = Partitions(3).list()
sage: matrix([[P(a).scalar_jack(Q(b)) for a in p] for b in p])
[1 0 0]
[0 1 0]
[0 0 1]
```

class JackPolynomial_j(*A*, *x*)

class JackPolynomial_p(*A*, *x*)

scalar_jack(*x*)
EXAMPLES:

```
sage: P = JackPolynomialsP(QQ)
sage: l = [P(p) for p in Partitions(3)]
sage: matrix([[a.scalar_jack(b) for a in l] for b in l])
[  6*t^3/(2*t^2 + 3*t + 1)                          0                          0]
[                       0     (2*t^3 + t^2)/(t + 2)                          0]
[                       0                          0 1/6*t^3 + 1/2*t^2 + 1/3*t]
```

class JackPolynomial_q(*A*, *x*)

JackPolynomialsJ(*R*, *t=None*)
Returns the algebra of Jack polynomials in the J basis.

If t is not specified, then the base ring will be obtained by making the univariate polynomial ring over R with the variable t and taking its fraction field.

EXAMPLES:

```
sage: JackPolynomialsJ(QQ)
Jack polynomials in the J basis over Fraction Field of Univariate Polynomial Ring in t over Rati
sage: JackPolynomialsJ(QQ,t=-1)
Jack polynomials in the J basis with t=-1 over Rational Field
```

At t = 1, the Jack polynomials in the J basis are scalar multiples of the Schur functions with the scalar given by a Partition's hook_product method at 1.

```
sage: J = JackPolynomialsJ(QQ, t=1)
sage: s = SFASchur(J.base_ring())
sage: p = Partition([3,2,1,1])
sage: s(J(p)) == p.hook_product(1)*s(p)
True
```

At t = 2, the Jack polynomials on the J basis are scalar multiples of the zonal polynomials with the scalar given by a Partition's hook_product method at 1.

```
sage: t = 2
sage: J = JackPolynomialsJ(QQ,t=t)
sage: Z = ZonalPolynomials(J.base_ring())
sage: p = Partition([2,2,1])
sage: Z(J(p)) == p.hook_product(t)*Z(p)
True
```

JackPolynomialsP(*R, t=None*)

Returns the algebra of Jack polynomials in the P basis.

If t is not specified, then the base ring will be obtained by making the univariate polynomial ring over R with the variable t and taking its fraction field.

EXAMPLES:

```
sage: JackPolynomialsP(QQ)
Jack polynomials in the P basis over Fraction Field of Univariate Polynomial Ring in t over Rati
sage: JackPolynomialsP(QQ,t=-1)
Jack polynomials in the P basis with t=-1 over Rational Field
```

At t = 1, the Jack polynomials on the P basis are the Schur symmetric functions.

```
sage: P = JackPolynomialsP(QQ,1)
sage: s = SFASchur(QQ)
sage: P([2,1])^2
JackP[2, 2, 1, 1] + JackP[2, 2, 2] + JackP[3, 1, 1, 1] + 2*JackP[3, 2, 1] + JackP[3, 3] + JackP[
sage: s([2,1])^2
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + 2*s[3, 2, 1] + s[3, 3] + s[4, 1, 1] + s[4, 2]
```

At t = 2, the Jack polynomials on the P basis are the zonal polynomials.

```
sage: P = JackPolynomialsP(QQ,2)
sage: Z = ZonalPolynomials(QQ)
sage: P([2])^2
64/45*JackP[2, 2] + 16/21*JackP[3, 1] + JackP[4]
sage: Z([2])^2
64/45*Z[2, 2] + 16/21*Z[3, 1] + Z[4]
sage: Z(P([2,1]))
```

```
Z[2, 1]
sage: P(Z([2,1]))
JackP[2, 1]
```

JackPolynomialsQ(*R*, *t=None*)

Returns the algebra of Jack polynomials in the Q basis.

If t is not specified, then the base ring will be obtained by making the univariate polynomial ring over R with the variable t and taking its fraction field.

EXAMPLES:

```
sage: JackPolynomialsQ(QQ)
Jack polynomials in the Q basis over Fraction Field of Univariate Polynomial Ring in t over Rati
sage: JackPolynomialsQ(QQ,t=-1)
Jack polynomials in the Q basis with t=-1 over Rational Field
```

JackPolynomialsQp(*R*, *t=None*)

Returns the algebra of Jack polynomials in the Qp, which is dual to the P basis with respect to the standard scalar product.

If t is not specified, then the base ring will be obtained by making the univariate polynomial ring over R with the variable t and taking its fraction field.

EXAMPLES:

```
sage: Qp = JackPolynomialsQp(QQ)
sage: P = JackPolynomialsP(QQ)
sage: a = Qp([2])
sage: a.scalar(P([2]))
1
sage: a.scalar(P([1,1]))
0
sage: P(Qp([2]))
((t-1)/(t+1))*JackP[1, 1] + JackP[2]
```

class JackPolynomials_generic(*R*, *t=None*)

class JackPolynomials_j(*R*, *t=None*)

class JackPolynomials_p(*R*, *t=None*)

class JackPolynomials_q(*R*, *t=None*)

ZonalPolynomials(*R*)

Returns the algebra of zonal polynomials.

EXAMPLES:

```
sage: Z = ZonalPolynomials(QQ)
sage: a = Z([2])
sage: Z([2])^2
64/45*Z[2, 2] + 16/21*Z[3, 1] + Z[4]
```

c1(*part*, *t*)

EXAMPLES:

```
sage: from sage.combinat.sf.jack import c1
sage: t = QQ['t'].gen()
sage: [c1(p,t) for p in Partitions(3)]
[2*t^2 + 3*t + 1, t + 2, 6]
```

c2 (*part, t*)

EXAMPLES:

```
sage: from sage.combinat.sf.jack import c2
sage: t = QQ['t'].gen()
sage: [c2(p,t) for p in Partitions(3)]
[6*t^3, 2*t^3 + t^2, t^3 + 3*t^2 + 2*t]
```

scalar_jack (*part1, part2, t*)

Returns the Jack scalar product between p(part1) and p(part2) where p is the power-sum basis.

EXAMPLES:

```
sage: Q.<t> = QQ[]
sage: from sage.combinat.sf.jack import scalar_jack
sage: matrix([[scalar_jack(p1,p2,t) for p1 in Partitions(4)] for p2 in Partitions(4)])
[   4*t       0       0       0       0]
[     0  3*t^2       0       0       0]
[     0       0  8*t^2       0       0]
[     0       0       0  4*t^3       0]
[     0       0       0       0 24*t^4]
```

17.33.14 k-Schur Functions

class kSchurFunction_generic (*A, x*)

class kSchurFunction_t (*A, x*)

kSchurFunctions (*R, k, t=None*)

Returns the k-Schur functions. See the examples below for caveats on their use.

EXAMPLES:

```
sage: ks3 = kSchurFunctions(QQ, 3); ks3
k-Schur Functions at level 3 over Univariate Polynomial Ring in t over Rational Field
sage: s = SFASchur(ks3.base_ring()) # Allow 't' coefficients for the Schurs.
sage: t = ks3.t # Allow 't' as input
sage: s(ks3([3,2,1]))
s[3, 2, 1] + t*s[4, 1, 1] + t*s[4, 2] + t^2*s[5, 1]


sage: ks3(s([3, 2, 1]) + t*s([4, 1, 1]) + t*s([4, 2]) + t^2*s([5, 1]))
ks3[3, 2, 1]


sage: ks3([4,3,2,1])  # k-Schurs are indexed by partitions with first part \le k.
0
```

Attempting to convert a function that is not in the linear span of the k-Schur's raises an error.

```
sage: ks3(s([4]))
...
ValueError: s[4] is not in the space spanned by k-Schur Functions at level 3 over Univariate Pol
```

Note that the product of k-Schurs is not guaranteed to be in the space spanned by the k-Schurs. In general, we only have that a k-Schur times a j-Schur is a (k+j)-Schur. This fact is not currently incorporated into the Sage design, so the multiplication of k-Schur functions may return an error. This example shows how to get around this 'manually'.

```
sage: ks2 = kSchurFunctions(QQ, 2)
sage: ks2([2,1])^2
...
ValueError: s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + (2*t+2)*s[3, 2, 1] + (t^2+1)*s[3,
3] + (2*t+1)*s[4, 1, 1] + (t^2+2*t+1)*s[4, 2] + (t^2+2*t)*s[5, 1] + t^2*s[6] is not in the
space spanned by k-Schur Functions at level 2 over Univariate Polynomial Ring in t over
Rational Field.
```

```
sage: f = s(ks2([2,1]))^2; f # Convert to Schur functions first and multiply there.
s[2, 2, 1, 1] + s[2, 2, 2] + s[3, 1, 1, 1] + (2*t+2)*s[3, 2, 1] + (t^2+1)*s[3,
3] + (2*t+1)*s[4, 1, 1] + (t^2+2*t+1)*s[4, 2] + (t^2+2*t)*s[5, 1] + t^2*s[6]
sage: ks4 = kSchurFunctions(QQ, 4)
sage: ks4(f) # The product of two 'ks2's is a 'ks4'.
ks4[2, 2, 1, 1] + ks4[2, 2, 2] + ks4[3, 1, 1, 1] + (t+2)*ks4[3, 2, 1] + (t^2+1)*ks4[3, 3] + (t+1
```

However, at t=1, the product of k-Schurs is in the span of the k-Schurs. Below are some examples at t=1.

```
sage: ks3 = kSchurFunctions(QQ, 3, 1); ks3
k-Schur Functions at level 3 with t=1 over Rational Field
sage: s = SFASchur(ks3.base_ring())
sage: ks3(s([3]))
ks3[3]
sage: s(ks3([3,2,1]))
s[3, 2, 1] + s[4, 1, 1] + s[4, 2] + s[5, 1]
sage: ks3([2,1])^2
ks3[2, 2, 1, 1] + ks3[2, 2, 2] + ks3[3, 1, 1, 1]
```

class kSchurFunctions_generic(*R, element_class=None*)

class kSchurFunctions_t(*R, k, t=None*)

17.33.15 LLT Polynomials

LLT(*R, k, t=None*)

Returns a class for working with LLT polynomials.

EXAMPLES:

```
sage: L3 = LLT(QQ,3); L3
LLT polynomials at level 3 over Fraction Field of Univariate Polynomial Ring in t over Rational
sage: L3.cospin([3,2,1])
(t+1)*m[1, 1] + m[2]
sage: L3.hcospin()
LLT polynomials in the HCosp basis at level 3 with t=t over Fraction Field of Univariate Polynom
```

class LLTElement_cospin(*A, x*)

class LLTElement_generic(*A, x*)

class LLTElement_spin(*A, x*)

LLTHCospin(*R, level, t=None*)

Returns the LLT polynomials in the HCospin basis at level level.

EXAMPLES:

```
sage: HCosp3 = LLTHCospin(QQ,3)
sage: HCosp3([1])^2
1/t*HCosp3[1, 1] + ((t-1)/t)*HCosp3[2]
```

LLTHSpin(*R, level, t=None*)

Returns the LLT polynomials in the HSpin basis at level level.

EXAMPLES:

```
sage: HSp3 = LLTHSpin(QQ,3)
sage: HSp3([1])^2
HSp[1, 1] + (-t+1)*HSp[2]
```

class **LLT_class**(*R, k, t=None*)

base_ring()

Returns the base ring of self.

EXAMPLES:

```
sage: LLT(QQ,3).base_ring()
Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

cospin(*skp*)

EXAMPLES:

```
sage: L3 = LLT(QQ,3)
sage: L3.cospin([2,1])
m[1]
sage: L3.cospin([3,2,1])
(t+1)*m[1, 1] + m[2]
sage: s = SFASchur(L3.base_ring())
sage: s(L3.cospin([[2],[1],[2]]))
t^4*s[2, 2, 1] + t^3*s[3, 1, 1] + (t^3+t^2)*s[3, 2] + (t^2+t)*s[4, 1] + s[5]
```

hcospin()

Returns the HCopsin basis.

EXAMPLES:

```
sage: LLT(QQ,3).hcospin()
LLT polynomials in the HCosp basis at level 3 with t=t over Fraction Field of Univariate Pol
```

hspin()

Returns the HSpin basis.

EXAMPLES:

```
sage: LLT(QQ,3).hspin()
LLT polynomials in the HSp basis at level 3 with t=t over Fraction Field of Univariate Polyn
```

level()

Returns the level of self.

EXAMPLES:

```
sage: LLT(QQ,3).level()
3
```

spin_square(*skp*)

Returns the spin polynomial associated with skp with the substitution $t \to t^2$ made.

EXAMPLES:

```
sage: L3 = LLT(QQ,3)
sage: L3.spin_square([2,1])
t*m[1]
sage: L3.spin_square([3,2,1])
```

```
(t^3+t)*m[1, 1] + t^3*m[2]
sage: L3.spin_square([[1],[1],[1]])
(t^6+2*t^4+2*t^2+1)*m[1, 1, 1] + (t^6+t^4+t^2)*m[2, 1] + t^6*m[3]
```

class **LLT_cospin**(*R*, *level*, *t=None*)

class **LLT_generic**(*R*, *level*, *t=None*)

level()

Returns the level of self.

EXAMPLES:

```
sage: from sage.combinat.sf.llt import *
sage: HSp3 = LLT_spin(QQ, 3)
sage: HSp3.level()
3
```

class **LLT_spin**(*R*, *level*, *t=None*)

17.33.16 Macdonald Polynomials

class **MacdonaldPolynomial_generic**(*A*, *x*)

nabla()

Returns the value of the nabla operator applied to self. The eigenvectors of the nabla operator are the Macdonald polynomials in the Ht basis.

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import *
sage: P = MacdonaldPolynomialsP(QQ)
sage: P([1,1]).nabla()
((q^2*t+q*t^2-2*t)/(q*t-1))*McdP[1, 1] + McdP[2]
```

omega_qt()

Returns the image of self under the ω_{qt} automorphism.

EXAMPLES:

```
sage: H = MacdonaldPolynomialsH(QQ)
sage: H([1,1]).omega_qt()
((2*q^2-2*q*t-2*q+2*t)/(t^3-t^2-t+1))*McdH[1, 1] + ((q-1)/(t-1))*McdH[2]
```

scalar_qt(*x*)

Returns the qt-Hall scalar product of self and x by converting both to the power-sum basis.

EXAMPLES:

```
sage: H = MacdonaldPolynomialsH(QQ)
sage: H([1]).scalar_qt(H([1]))
(-q + 1)/(-t + 1)
```

class **MacdonaldPolynomial_h**(*A*, *x*)

class **MacdonaldPolynomial_ht**(*A*, *x*)

nabla()

Returns the value of the nabla operator applied to self. The eigenvectors of the nabla operator are the Macdonald polynomials in the Ht basis.

EXAMPLES:

```
sage: Ht = MacdonaldPolynomialsHt(QQ)
sage: t = Ht.t; q = Ht.q;
sage: a = sum(Ht(p) for p in Partitions(3))
sage: a.nabla() == t^3*Ht([1,1,1])+q*t*Ht([2,1]) + q^3*Ht([3])
True
```

class MacdonaldPolynomial_j(*A*, *x*)

scalar_qt(*x*)

Returns the qt-Hall scalar product of self and x. If x is in the Macdonald J basis, then specialized code is used; otherwise, both are converted to the power-sums and the scalar product is carried out there.

EXAMPLES:

```
sage: J = MacdonaldPolynomialsJ(QQ)
sage: J([1,1]).scalar_qt(J([1,1]))
q^2*t^4 - q^2*t^3 - q*t^4 - q^2*t^2 + q^2*t + 2*q*t^2 + t^3 - t^2 - q - t + 1
sage: J([1,1]).scalar_qt(J([2]))
0
sage: J([2]).scalar_qt(J([2]))
q^4*t^2 - q^4*t - q^3*t^2 - q^2*t^2 + q^3 + 2*q^2*t + q*t^2 - q^2 - q - t + 1
```

class MacdonaldPolynomial_p(*A*, *x*)

scalar_qt(*x*)

Returns the qt-Hall scalar product of self and x. If x is in the Macdonald P or Q basis, then specialized code is used; otherwise, both are converted to the power-sums and the scalar product is carried out there.

EXAMPLES:

```
sage: Q = MacdonaldPolynomialsQ(QQ)
sage: P = MacdonaldPolynomialsP(QQ)
sage: a = P([2])
sage: b = Q([2])
sage: a.scalar_qt(a)
(q^3 - q^2 - q + 1)/(q*t^2 - q*t - t + 1)
sage: a.scalar_qt(b)
1
```

class MacdonaldPolynomial_q(*A*, *x*)

scalar_qt(*x*)

Returns the qt-Hall scalar product of self and x. If x is in the Macdonald P basis, then specialized code is used; otherwise, both are converted to the power-sums and the scalar product is carried out there.

EXAMPLES:

```
sage: Q = MacdonaldPolynomialsQ(QQ)
sage: H = MacdonaldPolynomialsH(QQ)
sage: a = Q([2])
sage: a.scalar_qt(a)
(-q*t^2 + q*t + t - 1)/(-q^3 + q^2 + q - 1)
sage: a.scalar_qt(H([1,1]))
t
```

class MacdonaldPolynomial_s(*A*, *x*)

creation(*k*)

EXAMPLES:

```
sage: S = MacdonaldPolynomialsS(QQ)
sage: a = S(1)
sage: a.creation(1)
(-q+1)*McdS[1]
sage: a.creation(2)
(q^2*t-q*t-q+1)*McdS[1, 1] + (q^2-q*t-q+t)*McdS[2]
```

omega_qt()

Returns the image of self under the Frobenius / omega automorphism.

EXAMPLES:

```
sage: S = MacdonaldPolynomialsS(QQ)
sage: S([1,1]).omega_qt()
(t/(t^3-t^2-t+1))*McdS[1, 1] + ((-1)/(-t^3+t^2+t-1))*McdS[2]
```

MacdonaldPolynomialsH(*R, q=None, t=None*)

Returns the Macdonald polynomials on the H basis. When the H basis is expanded on the Schur basis, the coefficients are the qt-Kostka numbers.

EXAMPLES:

```
sage: H = MacdonaldPolynomialsH(QQ)
sage: s = SFASchur(H.base_ring())
sage: s(H([2]))
q*s[1, 1] + s[2]
sage: s(H([1,1]))
s[1, 1] + t*s[2]
```

MacdonaldPolynomialsHt(*R, q=None, t=None*)

Returns the Macdonald polynomials on the Ht basis. The elements of the Ht basis are eigenvectors of the nabla operator. When expanded on the Schur basis, the coefficients are the modified qt-Kostka numbers.

EXAMPLES:

```
sage: Ht = MacdonaldPolynomialsHt(QQ)
sage: [Ht(p).nabla() for p in Partitions(3)]
[q^3*McdHt[3], q*t*McdHt[2, 1], t^3*McdHt[1, 1, 1]]


sage: s = SFASchur(Ht.base_ring())
sage: from sage.combinat.sf.macdonald import qt_kostka
sage: q,t = Ht.base_ring().gens()
sage: s(Ht([2,1]))
q*t*s[1, 1, 1] + (q+t)*s[2, 1] + s[3]
sage: qt_kostka([1,1,1],[2,1]).subs(t=1/t)*t^Partition([2,1]).weighted_size()
q*t
sage: qt_kostka([2,1],[2,1]).subs(t=1/t)*t^Partition([2,1]).weighted_size()
q + t
sage: qt_kostka([3],[2,1]).subs(t=1/t)*t^Partition([2,1]).weighted_size()
1
```

MacdonaldPolynomialsJ(*R, q=None, t=None*)

Returns the Macdonald polynomials on the J basis also known as the integral form of the Macdonald polynomials. These are scalar multiples of both the P and Q bases. When expressed in the P or Q basis, the scaling coefficients are polynomials in q and t rather than rational functions.

EXAMPLES:

```
sage: J = MacdonaldPolynomialsJ(QQ); J
Macdonald polynomials in the J basis over Fraction Field of Multivariate Polynomial Ring in q, t
sage: P = MacdonaldPolynomialsP(QQ)
sage: Q = MacdonaldPolynomialsQ(QQ)
sage: P(J([2]))
(q*t^2-q*t-t+1)*McdP[2]
sage: P(J([1,1]))
(t^3-t^2-t+1)*McdP[1, 1]
sage: Q(J([2]))
(q^3-q^2-q+1)*McdQ[2]
sage: Q(J([1,1]))
(q^2*t-q*t-q+1)*McdQ[1, 1]
```

MacdonaldPolynomialsP(*R, q=None, t=None*)

Returns the Macdonald polynomials on the P basis. These are upper triangularly related to the monomial symmetric functions and are orthogonal with respect to the qt-Hall scalar product.

EXAMPLES:

```
sage: P = MacdonaldPolynomialsP(QQ); P
Macdonald polynomials in the P basis over Fraction Field of Multivariate Polynomial Ring in q, t
sage: m = SFAMonomial(P.base_ring())
sage: P.transition_matrix(m,2)
[                          1 (q*t - q + t - 1)/(q*t - 1)]
[                          0                         1]
sage: P([1,1]).scalar_qt(P([2]))
0
sage: P([2]).scalar_qt(P([2]))
(q^3 - q^2 - q + 1)/(q*t^2 - q*t - t + 1)
sage: P([1,1]).scalar_qt(P([1,1]))
(q^2*t - q*t - q + 1)/(t^3 - t^2 - t + 1)
```

When q = 0, the Macdonald polynomials on the P basis are the same as the Hall-Littlewood polynomials on the P basis.

```
sage: P = MacdonaldPolynomialsP(QQ,q=0)
sage: P([2])^2
(t+1)*McdP[2, 2] + (-t+1)*McdP[3, 1] + McdP[4]
sage: HLP = HallLittlewoodP(QQ)
sage: HLP([2])^2
(t+1)*P[2, 2] + (-t+1)*P[3, 1] + P[4]
```

MacdonaldPolynomialsQ(*R, q=None, t=None*)

Returns the Macdonald polynomials on the Q basis. These are dual to the Macdonald polynomials on the P basis with respect to the qt-Hall scalar product.

EXAMPLES:

```
sage: Q = MacdonaldPolynomialsQ(QQ); Q
Macdonald polynomials in the Q basis over Fraction Field of Multivariate Polynomial Ring in q, t
sage: P = MacdonaldPolynomialsP(QQ)
sage: Q([2]).scalar_qt(P([2]))
1
sage: Q([2]).scalar_qt(P([1,1]))
0
sage: Q([1,1]).scalar_qt(P([2]))
0
sage: Q([1,1]).scalar_qt(P([1,1]))
```

```
1
sage: Q(P([2]))
((q^3-q^2-q+1)/(q*t^2-q*t-t+1))*McdQ[2]
sage: Q(P([1,1]))
((q^2*t-q*t-q+1)/(t^3-t^2-t+1))*McdQ[1, 1]
```

MacdonaldPolynomialsS(*R, q=None, t=None*)

Returns the modified Schur functions defined by the plethystic substitution $S_\mu = s_\mu[X(1-t)]$. When the Macdonald polynomials in the J basis are expressed in terms of the modified Schur functions, the coefficients are qt-Kostka numbers.

EXAMPLES:

```
sage: S = MacdonaldPolynomialsS(QQ)
sage: J = MacdonaldPolynomialsJ(QQ)
sage: S(J([2]))
q*McdS[1, 1] + ((-1)/(-1))*McdS[2]
sage: S(J([1,1]))
McdS[1, 1] + t*McdS[2]
sage: from sage.combinat.sf.macdonald import qt_kostka
sage: qt_kostka([2],[1,1])
t
sage: qt_kostka([1,1],[2])
q
```

class MacdonaldPolynomials_generic(*R, q=None, t=None*)

class MacdonaldPolynomials_h(*R, q=None, t=None*)

class MacdonaldPolynomials_ht(*R, q=None, t=None*)

class MacdonaldPolynomials_j(*R, q=None, t=None*)

class MacdonaldPolynomials_p(*R, q=None, t=None*)

class MacdonaldPolynomials_q(*R, q=None, t=None*)

class MacdonaldPolynomials_s(*R, q=None, t=None*)

c1(*part, q, t*)

This function returns the qt-Hall scalar product between J(part) and P(part).

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import c1
sage: R.<q,t> = QQ[]
sage: c1(Partition([2,1]),q,t)
-q^4*t + 2*q^3*t - q^2*t + q^2 - 2*q + 1
sage: c1(Partition([1,1]),q,t)
q^2*t - q*t - q + 1
```

c2(*part, q, t*)

This function returns the qt-Hall scalar product between J(part) and Q(part).

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import c2
sage: R.<q,t> = QQ[]
sage: c2(Partition([1,1]),q,t)
t^3 - t^2 - t + 1
sage: c2(Partition([2,1]),q,t)
-q*t^4 + 2*q*t^3 - q*t^2 + t^2 - 2*t + 1
```

qt_kostka(*lam, mu*)

Returns the $K_{\lambda\mu}(q,t)$ by computing the change of basis from the Macdonald H basis to the Schurs.

EXAMPLES:

```
sage: from sage.combinat.sf.macdonald import qt_kostka
sage: qt_kostka([2,1,1],[1,1,1,1])
t^3 + t^2 + t
sage: qt_kostka([1,1,1,1],[2,1,1])
q
sage: qt_kostka([1,1,1,1],[3,1])
q^3
sage: qt_kostka([1,1,1,1],[1,1,1,1])
1
sage: qt_kostka([2,1,1],[2,2])
q^2*t + q*t + q
sage: qt_kostka([2,2],[2,2])
q^2*t^2 + 1
sage: qt_kostka([4],[3,1])
t
sage: qt_kostka([2,2],[3,1])
q^2*t + q
sage: qt_kostka([3,1],[2,1,1])
q*t^3 + t^2 + t
sage: qt_kostka([2,1,1],[2,1,1])
q*t^2 + q*t + 1
sage: qt_kostka([2,1],[1,1,1,1])
0
```

17.33.17 Non-symmetric Macdonald Polynomials

class AugmentedLatticeDiagramFilling(*l, pi=None*)

are_attacking(*i, j, ii, jj*)

Returns True if the boxes (i,j) and (ii,jj) in self are attacking.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: all( a.are_attacking(i,j,ii,jj) for (i,j),(ii,jj) in a.attacking_boxes())
True
sage: a.are_attacking(1,1,3,2)
False
```

attacking_boxes()

Returns a list of pairs of boxes in self that are attacking.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.attacking_boxes()[:5]
[((1, 1), (2, 1)),
 ((1, 1), (3, 1)),
 ((1, 1), (6, 1)),
 ((1, 1), (2, 0)),
 ((1, 1), (3, 0))]
```

boxes()

Returns a list of the coordinates of the boxes of self, including the 'basement row'.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.boxes()
[(1, 1),
 (1, 2),
 (2, 1),
 (3, 1),
 (3, 2),
 (3, 3),
 (6, 1),
 (6, 2),
 (1, 0),
 (2, 0),
 (3, 0),
 (4, 0),
 (5, 0),
 (6, 0)]
```

coeff(*q*, *t*)

Returns the coefficient in front of self in the HHL formula for the expansion of the non-symmetric Macdonald polynomial E(self.shape()).

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: q,t = var('q,t')
sage: a.coeff(q,t)
(t - 1)^4/((q*t^2 - 1)^2*(q^2*t^3 - 1)^2)
```

coeff_integral(*q*, *t*)

Returns the coefficient in front of self in the HHL formula for the expansion of the integral non-symmetric Macdonald polynomial E(self.shape())

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: q,t = var('q,t')
sage: a.coeff_integral(q,t)
(t - 1)^4*(q*t^2 - 1)^2*(q^2*t^3 - 1)^2
```

coinv()

Returns self's co-inversion statistic.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.coinv()
2
```

descents()

Returns a list of the descents of self.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.descents()
[(1, 2), (3, 2)]
```

inv()

Returns self's inversion statistic.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.inv()
15
```

inversions()

Returns a list of the inversions of self.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.inversions()[:5]
[((6, 2), (3, 2)),
 ((1, 2), (6, 1)),
 ((1, 2), (3, 1)),
 ((1, 2), (2, 1)),
 ((6, 1), (3, 1))]
sage: len(a.inversions())
25
```

is_non_attacking()

Returns True if self in non-attacking.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.is_non_attacking()
True
sage: a = AugmentedLatticeDiagramFilling([[1, 1, 1], [2, 3], [3]])
sage: a.is_non_attacking()
False
sage: a = AugmentedLatticeDiagramFilling([[2,2],[1]])
sage: a.is_non_attacking()
False
sage: pi = Permutation([2,1]).to_permutation_group_element()
sage: a = AugmentedLatticeDiagramFilling([[2,2],[1]],pi)
sage: a.is_non_attacking()
True
```

maj()

Returns the major index of self.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.maj()
3
```

permuted_filling(*sigma*)

EXAMPLES:

```
sage: pi=Permutation([2,1,4,3]).to_permutation_group_element()
sage: fill=[[2],[1,2,3],[],[3,1]]
sage: AugmentedLatticeDiagramFilling(fill).permuted_filling(pi)
[[2, 1], [1, 2, 1, 4], [4], [3, 4, 2]]
```

reading_order()

Returns a list of coordinates of the boxes in self, starting from the top right, and reading from right to left. Note that this includes the 'basement row' of self.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.reading_order()
```

```
[(3, 3),
 (6, 2),
 (3, 2),
 (1, 2),
 (6, 1),
 (3, 1),
 (2, 1),
 (1, 1),
 (6, 0),
 (5, 0),
 (4, 0),
 (3, 0),
 (2, 0),
 (1, 0)]
```

reading_word()

Return the reading word of self, obtained by reading the boxes entries of self from right to left, starting in the upper right.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.reading_word()
word: 25465321
```

shape()

Returns the shape of self.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.shape()
[2, 1, 3, 0, 0, 2]
```

weight()

Returns the weight of self.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a.weight()
[1, 2, 1, 1, 2, 1]
```

E(*mu, q=None, t=None, pi=None*)

Returns the non-symmetric Macdonald polynomial in type A corresponding to a shape mu, with basement permuted according to pi.

Note that if both q and t are specified, then they must have the same parent.

REFERENCE:

- 'A combinatorial formula for non-symmetric Macdonald polynomials'. Haiman, Haglund, and Loehr. http://arxiv.org/abs/math/0601693

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import E
sage: E([0,0,0])
1
sage: E([1,0,0])
x0
sage: E([0,1,0])
((-t + 1)/(-q*t^2 + 1))*x0 + x1
```

```
sage: E([0,0,1])
((-t + 1)/(-q*t + 1))*x0 + ((-t + 1)/(-q*t + 1))*x1 + x2
sage: E([1,1,0])
x0*x1
sage: E([1,0,1])
((-t + 1)/(-q*t^2 + 1))*x0*x1 + x0*x2
sage: E([0,1,1])
((-t + 1)/(-q*t + 1))*x0*x1 + ((-t + 1)/(-q*t + 1))*x0*x2 + x1*x2
sage: E([2,0,0])
x0^2 + ((-q*t + q)/(-q*t + 1))*x0*x1 + ((-q*t + q)/(-q*t + 1))*x0*x2
sage: E([0,2,0])
((-t + 1)/(-q^2*t^2 + 1))*x0^2 + ((-q^2*t^3 + q^2*t^2 - q*t^2 + 2*q*t - q + t - 1)/(-q^3*t^3 + q
```

E_integral (*mu, q=None, t=None, pi=None*)

Returns the integral form for the non-symmetric Macdonald polynomial in type A corresponding to a shape mu.

Note that if both q and t are specified, then they must have the same parent.

REFERENCE:

- 'A combinatorial formula for non-symmetric Macdonald polynomials'. Haiman, Haglund, and Loehr. http://arxiv.org/abs/math/0601693

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import E_integral
sage: E_integral([0,0,0])
1
sage: E_integral([1,0,0])
(-t + 1)*x0
sage: E_integral([0,1,0])
(-q*t^2 + 1)*x0 + (-t + 1)*x1
sage: E_integral([0,0,1])
(-q*t + 1)*x0 + (-q*t + 1)*x1 + (-t + 1)*x2
sage: E_integral([1,1,0])
(t^2 - 2*t + 1)*x0*x1
sage: E_integral([1,0,1])
(q*t^3 - q*t^2 - t + 1)*x0*x1 + (t^2 - 2*t + 1)*x0*x2
sage: E_integral([0,1,1])
(q^2*t^3 + q*t^4 - q*t^3 - q*t^2 - q*t - t^2 + t + 1)*x0*x1 + (q*t^2 - q*t - t + 1)*x0*x2 + (t^2
sage: E_integral([2,0,0])
(t^2 - 2*t + 1)*x0^2 + (q^2*t^2 - q^2*t - q*t + q)*x0*x1 + (q^2*t^2 - q^2*t - q*t + q)*x0*x2
sage: E_integral([0,2,0])
(q^2*t^3 - q^2*t^2 - t + 1)*x0^2 + (q^4*t^3 - q^3*t^2 - q^2*t + q*t^2 - q*t + q - t + 1)*x0*x1 +
```

Ht (*mu, q=None, t=None, pi=None*)

Returns the symmetric Macdonald polynomial using the Haiman, Haglund, and Loehr formula.

Note that if both q and t are specified, then they must have the same parent.

REFERENCE:

- 'A combinatorial formula for non-symmetric Macdonald polynomials'. Haiman, Haglund, and Loehr. http://arxiv.org/abs/math/0601693

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import Ht
sage: HHt = MacdonaldPolynomialsHt(QQ)
sage: Ht([0,0,1])
```

```
x0 + x1 + x2
sage: HHt([1]).expand(3)
x0 + x1 + x2
sage: Ht([0,0,2])
x0^2 + (q + 1)*x0*x1 + x1^2 + (q + 1)*x0*x2 + (q + 1)*x1*x2 + x2^2
sage: HHt([2]).expand(3)
x0^2 + (q + 1)*x0*x1 + x1^2 + (q + 1)*x0*x2 + (q + 1)*x1*x2 + x2^2
```

class LatticeDiagram(*l*)

a(*i, j*)

Returns len(self.arm(i,j)).

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.a(5,2)
3
```

arm(*i, j*)

Returns the arm of the box (i,j) in self.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.arm(5,2)
[(1, 2), (3, 2), (8, 1)]
```

arm_left(*i, j*)

Returns the left arm of the box (i,j) in self.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.arm_left(5,2)
[(1, 2), (3, 2)]
```

arm_right(*i, j*)

Returns the right arm of the box (i,j) in self.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.arm_right(5,2)
[(8, 1)]
```

boxes()

EXAMPLES:

```
sage: a = LatticeDiagram([3,0,2])
sage: a.boxes()
[(1, 1), (1, 2), (1, 3), (3, 1), (3, 2)]
sage: a = LatticeDiagram([2, 1, 3, 0, 0, 2])
sage: a.boxes()
[(1, 1), (1, 2), (2, 1), (3, 1), (3, 2), (3, 3), (6, 1), (6, 2)]
```

boxes_same_and_lower_right(*ii, jj*)

Returns a list of the boxes that are in the same row as self, and in the row below self (including the basement) that are strictly to the right of self.

EXAMPLES:

```
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: a = a.shape()
sage: a.boxes_same_and_lower_right(1,1)
[(2, 1), (3, 1), (6, 1), (2, 0), (3, 0), (4, 0), (5, 0), (6, 0)]
sage: a.boxes_same_and_lower_right(1,2)
[(3, 2), (6, 2), (2, 1), (3, 1), (6, 1)]
sage: a.boxes_same_and_lower_right(3,3)
[(6, 2)]
sage: a.boxes_same_and_lower_right(2,3)
[(3, 3), (3, 2), (6, 2)]
```

flip()

Returns the flip of the self where flip is defined as follows. Let r = max(self). Then self.flip()[i] = r - self[i].

EXAMPLES:

```
sage: a = LatticeDiagram([3,0,2])
sage: a.flip()
[0, 3, 1]
```

l(*i, j*)

Returns the self[i] - j.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.l(5,2)
1
```

leg(*i, j*)

Returns the leg of the box (i,j) in self.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.leg(5,2)
[(5, 3)]
```

size()

Returns the number of boxes in self.

EXAMPLES:

```
sage: a = LatticeDiagram([3,1,2,4,3,0,4,2,3])
sage: a.size()
22
```

class **NonattackingBacktracker**(*shape, pi=None*)

get_next_pos(*ii, jj*)

EXAMPLES:

```
sage: from sage.combinat.sf.ns_macdonald import NonattackingBacktracker
sage: a = AugmentedLatticeDiagramFilling([[1,6],[2],[3,4,2],[],[],[5,5]])
sage: n = NonattackingBacktracker(a.shape())
sage: n.get_next_pos(1, 1)
(2, 1)
sage: n.get_next_pos(6, 1)
(1, 2)
sage: n = NonattackingBacktracker(LatticeDiagram([2,2,2]))
sage: n.get_next_pos(3, 1)
(1, 2)
```

NonattackingFillings(*shape, pi=None*)

Returning the combinatorial class of nonattacking fillings of a given shape.

EXAMPLES:

```
sage: NonattackingFillings([0,1,2])
Nonattacking fillings of [0, 1, 2]
sage: NonattackingFillings([0,1,2]).list()
[[[1], [2, 1], [3, 2, 1]],
 [[1], [2, 1], [3, 2, 2]],
 [[1], [2, 1], [3, 2, 3]],
 [[1], [2, 1], [3, 3, 1]],
 [[1], [2, 1], [3, 3, 2]],
 [[1], [2, 1], [3, 3, 3]],
 [[1], [2, 2], [3, 1, 1]],
 [[1], [2, 2], [3, 1, 2]],
 [[1], [2, 2], [3, 1, 3]],
 [[1], [2, 2], [3, 3, 1]],
 [[1], [2, 2], [3, 3, 2]],
 [[1], [2, 2], [3, 3, 3]]]
```

class **NonattackingFillings_shape**(*shape, pi=None*)

flip()

Returns the nonattacking fillings of the the flipped shape.

EXAMPLES:

```
sage: NonattackingFillings([0,1,2]).flip()
Nonattacking fillings of [2, 1, 0]
```

17.34 Root Systems

17.34.1 Cartan types

class **CartanTypeFactory**()

samples(*finite=False, affine=False, crystalographic=False*)

Returns a sample of the implemented cartan types

With finite=True resp. affine=True, one can restrict to finite resp. affine only cartan types

EXAMPLES:

```
sage: CartanType.samples(finite=True)
[['A', 1], ['A', 5], ['B', 5], ['C', 5], ['D', 5], ['E', 6], ['E', 7], ['E', 8], ['F', 4], [

sage: CartanType.samples(affine=True)
[['A', 1, 1], ['A', 5, 1], ['B', 5, 1], ['C', 5, 1], ['D', 5, 1], ['E', 6, 1], ['E', 7, 1],

sage: CartanType.samples()
[['A', 1], ['A', 5], ['B', 5], ['C', 5], ['D', 5], ['E', 6], ['E', 7], ['E', 8], ['F', 4], [
sage: CartanType.samples(crystalographic=True)
[['A', 1], ['A', 5], ['B', 5], ['C', 5], ['D', 5], ['E', 6], ['E', 7], ['E', 8], ['F', 4], [
```

class **CartanType_abstract**()

Abstract class for cartan types

Subclasses should implement:

- type()
- type_string()
- dynkin_diagram()
- cartan_matrix()
- is_finite()
- is_affine()
- is_irreducible()

dual()
Returns the dual cartan type, possibly just as a formal dual.
EXAMPLES:

```
sage: CartanType(['F',4]).dual()
['F', 4]^*
```

index_set()
Returns the index set for self.
EXAMPLES:

```
sage: CartanType(['A', 3, 1]).index_set()
[0, 1, 2, 3]
sage: CartanType(['D', 4]).index_set()
[1, 2, 3, 4]
```

is_affine()
Returns whether self is affine.
EXAMPLES:

```
sage: CartanType(['A', 3]).is_affine()
False
sage: CartanType(['A', 3, 1]).is_affine()
True
```

is_crystalographic()
Returns whether this Cartan type is simple laced
EXAMPLES:

```
sage: [ [t, t.is_crystalographic() ] for t in CartanType.samples(finite=True) ]
[[['A', 1], True], [['A', 5], True],
[['B', 5], True], [['C', 5], True], [['D', 5], True],
[['E', 6], True], [['E', 7], True], [['E', 8], True],
[['F', 4], True], [['G', 2], True],
[['I', 5], False], [['H', 3], False], [['H', 4], False]]
```

TESTS:

```
sage: all(t.is_crystalographic() for t in CartanType.samples(affine=True))
True
```

is_finite()
Returns whether this Cartan type is finite. This should be overridden in any subclass.
EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_type import CartanType_abstract
sage: C = CartanType_abstract()
sage: C.is_irreducible()
...
NotImplementedError
```

```
sage: CartanType(['A',4]).is_finite()
True
sage: CartanType(['A',4, 1]).is_finite()
False
```

is_irreducible()

Report whether this Cartan type is irreducible (i.e. simple). This should be overridden in any subclass.

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_type import CartanType_abstract
sage: C = CartanType_abstract()
sage: C.is_irreducible()
...
NotImplementedError
```

is_reducible()

Report whether the root system is reducible (i.e. not simple), that is whether it can be factored as a product of root systems.

EXAMPLES:

```
sage: CartanType("A2xB3").is_reducible()
True
sage: CartanType(['A',2]).is_reducible()
False
```

is_simple_laced()

Returns whether this Cartan type is simple laced

EXAMPLES:

```
sage: [ [t, t.is_simply_laced() ] for t in CartanType.samples() ]
[[['A', 1], True], [['A', 5], True],
[['B', 5], False], [['C', 5], False], [['D', 5], True],
[['E', 6], True], [['E', 7], True], [['E', 8], True],
[['F', 4], False], [['G', 2], False], [['I', 5], False], [['H', 3], False], [['H', 4], False
[['A', 1, 1], False], [['A', 5, 1], True],
[['B', 5, 1], False], [['C', 5, 1], False], [['D', 5, 1], True],
[['E', 6, 1], True], [['E', 7, 1], True], [['E', 8, 1], True],
[['F', 4, 1], False], [['G', 2, 1], False],
[['A', 2, 2], False], [['A', 10, 2], False], [['A', 9, 2], False], [['D', 5, 2], False], [['
```

rank()

Returns the rank of self.

EXAMPLES:

```
sage: CartanType(['A', 4]).rank()
4
sage: CartanType(['A', 7, 2]).rank()
4
sage: CartanType(['I', 8]).rank()
2
```

root_system()

Returns the root system associated to self.

EXAMPLES:

```
sage: CartanType(['A',4]).root_system()
Root system of type ['A', 4]
```

type()

Returns the type of self, or None if unknown. This method should be overridden in any subclass.

EXAMPLES:

```
sage: from sage.combinat.root_system.cartan_type import CartanType_abstract
sage: C = CartanType_abstract()
sage: C.type() is None
True
```

class CartanType_simple()

cartan_matrix()
Returns the Cartan matrix associated with self.
EXAMPLES:

```
sage: CartanType(['A',4]).cartan_matrix()
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
```

dual()
EXAMPLES:

```
sage: CartanType(["A", 3]).dual()
['A', 3]
sage: CartanType(["B", 3]).dual()
['C', 3]
```

dynkin_diagram()
Returns the Dynkin diagram associated with self.
EXAMPLES:

```
sage: CartanType(['A',4]).dynkin_diagram()
O---O---O---O
1   2   3   4
A4
```

is_irreducible()
EXAMPLES:

```
sage: CartanType(['A', 3]).is_irreducible()
True
```

type()
Returns the type of self.
EXAMPLES:

```
sage: CartanType(['A', 4]).type()
'A'
sage: CartanType(['A', 4, 1]).type()
'A'
```

class CartanType_simple_affine(*t*)
A class for affine simple Cartan types

classical()
Returns the classical Cartan type associated with self (which should be affine)
Caveat: only implemented for untwisted
EXAMPLES:

```
sage: CartanType(['A', 3, 1]).classical()
['A', 3]
sage: CartanType(['B', 3, 1]).classical()
['B', 3]
```

is_affine()
EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_affine()
True
```

is_crystalographic()
EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_crystalographic()
True
```

is_finite()
EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_finite()
False
```

is_simply_laced()
EXAMPLES:

```
sage: CartanType(['A', 3, 1]).is_simply_laced()
True
sage: CartanType(['D', 4, 3]).is_simply_laced()
False
sage: CartanType(['D', 4, 1]).is_simply_laced()
True
sage: CartanType(['B', 4, 1]).is_simply_laced()
False
```

rank()
EXAMPLES:

```
sage: CartanType(['D', 4, 3]).rank()
3
sage: CartanType(['B', 4, 1]).rank()
4
```

class **CartanType_simple_finite**(*t*)
A class for finite simple Cartan types

affine()
Returns the corresponding untwisted affine Cartan type
EXAMPLES:

```
sage: CartanType(['A',3]).affine()
['A', 3, 1]
```

dual()
EXAMPLES:

```
sage: CartanType(['A',3]).dual()
['A', 3]
sage: CartanType(['D',4]).dual()
['D', 4]
sage: CartanType(['E',8]).dual()
```

```
['E', 8]
sage: CartanType(['B',3]).dual()
['C', 3]
sage: CartanType(['C',2]).dual()
['B', 2]
```

is_affine()
EXAMPLES:

```
sage: CartanType(["A", 3]).is_affine()
False
```

is_crystalographic()
EXAMPLES:

```
sage: CartanType(["A", 3]).is_crystalographic()
True
sage: CartanType(["I", 2]).is_crystalographic()
False
```

is_finite()
EXAMPLES:

```
sage: CartanType(["A", 3]).is_finite()
True
```

is_simply_laced()
EXAMPLES:

```
sage: CartanType(['A',3]).is_simply_laced()
True
sage: CartanType(['B',3]).is_simply_laced()
False
```

rank()
EXAMPLES:

```
sage: CartanType(["A", 3]).rank()
3
```

17.34.2 Dynkin diagrams

DynkinDiagram(**args*)
INPUT:

- `ct` - A Cartan Type Returns a Dynkin diagram for type ct.

The edge multiplicities are encoded as edge labels. This uses the convention in Kac / Fulton Harris Representation theory wikipedia http://en.wikipedia.org/wiki/Dynkin_diagram, that is for i != j:

```
j --k--> i <==> a_ij = -k
           <==> -scalar(coroot[i], root[j]) = k
           <==> multiple arrows point from the longer root
                to the shorter one
```

TODO: say something about the node labelling conventions.

EXAMPLES:

```
sage: DynkinDiagram(['A', 4])
 O---O---O---O
 1   2   3   4
 A4
sage: DynkinDiagram(['A',1],['A',1])
 O
 1
 O
 2
 A1xA1
sage: R = RootSystem("A2xB2xF4")
sage: DynkinDiagram(R)
 O---O
 1   2
 O=>=O
 3   4
 O---O=>=O---O
 5   6   7   8
 A2xB2xF4
```

class DynkinDiagram_class(*t*)

add_edge(*i, j, label=1*)

EXAMPLES:

```
sage: from sage.combinat.root_system.dynkin_diagram import DynkinDiagram_class
sage: d = DynkinDiagram_class(CartanType(['A',3]))
sage: list(sorted(d.edges()))
[]
sage: d.add_edge(2, 3)
sage: list(sorted(d.edges()))
[(2, 3, 1), (3, 2, 1)]
```

cartan_matrix()

returns the Cartan matrix for this Dynkin diagram

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).cartan_matrix()
[ 2 -1  0]
[-1  2 -2]
[ 0 -1  2]
```

cartan_type()

EXAMPLES:

```
sage: DynkinDiagram("A2","B2","F4").cartan_type()
A2xB2xF4
```

column(*j*)

Returns the j^{th} column $(a_{i,j})_i$ of the Cartan matrix corresponding to this Dynkin diagram, as a container (or iterator) of tuples $(i, a_{i,j})$

EXAMPLES:

```
sage: g = DynkinDiagram(["B",4])
sage: [ (i,a) for (i,a) in g.column(3) ]
[(3, 2), (2, -1), (4, -2)]
```

dual()

Returns the dual Dynkin diagram, obtained by reversing all edges.

EXAMPLES:

```
sage: D = DynkinDiagram(['C',3])
sage: D.edges()
[(1, 2, 1), (2, 1, 1), (2, 3, 1), (3, 2, 2)]
sage: D.dual()
O---O=>=O
1   2   3
B3
sage: D.dual().edges()
[(1, 2, 1), (2, 1, 1), (2, 3, 2), (3, 2, 1)]
sage: D.dual() == DynkinDiagram(['B',3])
True
```

TESTS:

```
sage: D = DynkinDiagram(['A',0]); D
Dynkin diagram of type ['A', 0]
sage: D.edges()
[]
sage: D.dual()
Dynkin diagram of type ['A', 0]
sage: D.dual().edges()
[]
sage: D = DynkinDiagram(['A',1])
sage: D.edges()
[]
sage: D.dual()
O
1
A1
sage: D.dual().edges()
[]
```

dynkin_diagram()

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).dynkin_diagram()
O---O=<=O
1   2   3
C3
```

index_set()

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).index_set()
[1, 2, 3]
sage: DynkinDiagram("A2","B2","F4").index_set()
[1, 2, 3, 4, 5, 6, 7, 8]
```

rank()

Returns the index set for this Dynkin diagram

EXAMPLES:

```
sage: DynkinDiagram(['C',3]).rank()
3
sage: DynkinDiagram("A2","B2","F4").rank()
8
```

row(i)

Returns the i^{th} row $(a_{i,j})_j$ of the Cartan matrix corresponding to this Dynkin diagram, as a container (or

iterator) of tuples $(j, a_{i,j})$

EXAMPLES:

```
sage: g = DynkinDiagram(["C",4])
sage: [ (i,a) for (i,a) in g.row(3) ]
[(3, 2), (2, -1), (4, -2)]
```

dynkin_diagram(*t*)

Returns the Dynkin diagram of type t.

Note that this function is deprecated, and that you should use DynkinDiagram instead as this will be disappearing in the near future.

EXAMPLES:

```
sage: dynkin_diagram(["A", 3])
doctest:1: DeprecationWarning: dynkin_diagram is deprecated, use DynkinDiagram instead!
O---O---O
1   2   3
A3
```

dynkin_diagram_ascii_art(*t, shift=0, nolabel=False*)

Returns ascii art representing a Dynkin diagram of a finite Cartan type. Accessed through the __repr__ method of DynkinDiagram.

The optional parameter shift causes the indices to be shifted by that amount, for use in Dynkin diagrams of reducible types. The parameter nolabel, if set, inhibits the printing of the Cartan type name as part of the ascii art.

EXAMPLES:

```
sage: DynkinDiagram("E6")
        O 2
        |
        |
 O---O---O---O---O
 1   3   4   5   6
 E6
```

extended_dynkin_diagram_ascii_art(*t*)

Returns ascii art representing the extended Dynkin diagram of a finite Cartan type. This is also the Dynkin diagram of the associated untwisted affine root system.

EXAMPLES:

```
sage: DynkinDiagram(['E',6,1])
        O 0
        |
        |
        O 2
        |
        |
O---O---O---O---O
1   3   4   5   6
E6~
```

precheck(*t, letter=None, length=None, affine=None, n_ge=None, n=None*)

EXAMPLES:

```
sage: from sage.combinat.root_system.dynkin_diagram import precheck
sage: ct = CartanType(['A',4])
sage: precheck(ct, letter='C')
...
ValueError: t[0] must be = 'C'
sage: precheck(ct, affine=1)
...
ValueError: t[2] must be = 1
sage: precheck(ct, length=3)
...
ValueError: len(t) must be = 3
sage: precheck(ct, n=3)
...
ValueError: t[1] must be = 3
sage: precheck(ct, n_ge=5)
...
ValueError: t[1] must be >= 5
```

17.34.3 Cartan matrices

cartan_matrix(*t*)

Returns the Cartan matrix corresponding to type t.

EXAMPLES:

```
sage: cartan_matrix(['A', 4])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -1]
[ 0  0 -1  2]
sage: cartan_matrix(['B', 6])
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2 -1  0  0]
[ 0  0 -1  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -2  2]
sage: cartan_matrix(['C', 4])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -1  2 -2]
[ 0  0 -1  2]
sage: cartan_matrix(['D', 6])
[ 2 -1  0  0  0  0]
[-1  2 -1  0  0  0]
[ 0 -1  2 -1  0  0]
[ 0  0 -1  2 -1 -1]
[ 0  0  0 -1  2  0]
[ 0  0  0 -1  0  2]
sage: cartan_matrix(['E',6])
[ 2  0 -1  0  0  0]
[ 0  2  0 -1  0  0]
[-1  0  2 -1  0  0]
[ 0 -1 -1  2 -1  0]
[ 0  0  0 -1  2 -1]
[ 0  0  0  0 -1  2]
sage: cartan_matrix(['E',7])
```

```
[ 2  0 -1  0  0  0  0]
[ 0  2  0 -1  0  0  0]
[-1  0  2 -1  0  0  0]
[ 0 -1 -1  2 -1  0  0]
[ 0  0  0 -1  2 -1  0]
[ 0  0  0  0 -1  2 -1]
[ 0  0  0  0  0 -1  2]
sage: cartan_matrix(['E', 8])
[ 2  0 -1  0  0  0  0  0]
[ 0  2  0 -1  0  0  0  0]
[-1  0  2 -1  0  0  0  0]
[ 0 -1 -1  2 -1  0  0  0]
[ 0  0  0 -1  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
sage: cartan_matrix(['F', 4])
[ 2 -1  0  0]
[-1  2 -1  0]
[ 0 -2  2 -1]
[ 0  0 -1  2]
```

This is different from MuPAD-Combinat, due to different node convention?

```
sage: cartan_matrix(['G', 2])
[ 2 -3]
[-1  2]
sage: cartan_matrix(['A', 3, 1])
[ 2 -1  0 -1]
[-1  2 -1  0]
[ 0 -1  2 -1]
[-1  0 -1  2]
sage: cartan_matrix(['B', 3, 1])
[ 2  0 -1  0]
[ 0  2 -1  0]
[-1 -1  2 -1]
[ 0  0 -2  2]
sage: cartan_matrix(['C', 3, 1])
[ 2 -1  0  0]
[-2  2 -1  0]
[ 0 -1  2 -2]
[ 0  0 -1  2]
sage: cartan_matrix(['D', 4, 1])
[ 2  0 -1  0  0]
[ 0  2 -1  0  0]
[-1 -1  2 -1 -1]
[ 0  0 -1  2  0]
[ 0  0 -1  0  2]
sage: cartan_matrix(['E', 6, 1])
[ 2  0 -1  0  0  0  0]
[ 0  2  0 -1  0  0  0]
[-1  0  2  0 -1  0  0]
[ 0 -1  0  2 -1  0  0]
[ 0  0 -1 -1  2 -1  0]
[ 0  0  0  0 -1  2 -1]
[ 0  0  0  0  0 -1  2]
sage: cartan_matrix(['E', 7, 1])
[ 2 -1  0  0  0  0  0  0]
```

```
[-1  2  0 -1  0  0  0  0]
[ 0  0  2  0 -1  0  0  0]
[ 0 -1  0  2 -1  0  0  0]
[ 0  0 -1 -1  2 -1  0  0]
[ 0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0 -1  2 -1]
[ 0  0  0  0  0  0 -1  2]
sage: cartan_matrix(['E', 8, 1])
[ 2  0  0  0  0  0  0  0 -1]
[ 0  2  0 -1  0  0  0  0  0]
[ 0  0  2  0 -1  0  0  0  0]
[ 0 -1  0  2 -1  0  0  0  0]
[ 0  0 -1 -1  2 -1  0  0  0]
[ 0  0  0  0 -1  2 -1  0  0]
[ 0  0  0  0  0 -1  2 -1  0]
[ 0  0  0  0  0  0 -1  2 -1]
[-1  0  0  0  0  0  0 -1  2]
sage: cartan_matrix(['F', 4, 1])
[ 2 -1  0  0  0]
[-1  2 -1  0  0]
[ 0 -1  2 -1  0]
[ 0  0 -2  2 -1]
[ 0  0  0 -1  2]
sage: cartan_matrix(['G', 2, 1])
[ 2  0 -1]
[ 0  2 -3]
[-1 -1  2]
```

17.34.4 Coxeter matrices

coxeter_matrix(*t*)

Returns the Coxeter matrix of type t.

EXAMPLES:

```
sage: coxeter_matrix(['A', 4])
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
sage: coxeter_matrix(['B', 4])
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
sage: coxeter_matrix(['C', 4])
[1 3 2 2]
[3 1 3 2]
[2 3 1 4]
[2 2 4 1]
sage: coxeter_matrix(['D', 4])
[1 3 2 2]
[3 1 3 3]
[2 3 1 2]
[2 3 2 1]
```

```
sage: coxeter_matrix(['E', 6])
[1 2 3 2 2 2]
[2 1 2 3 2 2]
[3 2 1 3 2 2]
[2 3 3 1 3 2]
[2 2 2 3 1 3]
[2 2 2 2 3 1]

sage: coxeter_matrix(['F', 4])
[1 3 2 2]
[3 1 4 2]
[2 4 1 3]
[2 2 3 1]

sage: coxeter_matrix(['G', 2])
[1 6]
[6 1]
```

coxeter_matrix_as_function(*t*)

Returns the coxeter matrix associated to the Cartan type t.

EXAMPLES:

```
sage: from sage.combinat.root_system.coxeter_matrix import coxeter_matrix_as_function
sage: f = coxeter_matrix_as_function(['A',4])
sage: matrix([[f(i,j) for j in range(1,5)] for i in range(1,5)])
[1 3 2 2]
[3 1 3 2]
[2 3 1 3]
[2 2 3 1]
```

17.34.5 Root systems

class RootSystem(*cartan_type, as_dual_of=None*)

Returns the root system associated to the Cartan type t.

EXAMPLES: We construct the root system for type B_3

```
sage: R=RootSystem(['B',3]); R
Root system of type ['B', 3]
```

R models the root system abstractly. It comes equipped with various realizations of the root and weight lattices, where all computation take place. Let us play first with the root lattice.

```
sage: space = R.root_lattice()
sage: space
Root lattice of the Root system of type ['B', 3]
```

It is the free $\mathbf{Z}$-module $\bigoplus_i \mathbf{Z}.\alpha_i$ spanned by the simple roots:

```
sage: space.base_ring()
Integer Ring
sage: list(space.basis())
[alpha[1], alpha[2], alpha[3]]
```

Let us do some computations with the simple roots:

```
sage: alpha = space.simple_roots()
sage: alpha[1] + alpha[2]
alpha[1] + alpha[2]
```

There is a canonical pairing between the root lattice and the coroot lattice:

```
sage: R.coroot_lattice()
Coroot lattice of the Root system of type ['B', 3]
```

We construct the simple coroots, and do some computations (see comments about duality below for some caveat).

```
sage: alphacheck = space.simple_coroots()
sage: list(alphacheck)
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

We can carry over the same computations in any of the other realizations of the root lattice, like the root space $\bigoplus_i \mathbf{Q}.\alpha_i$, the weight lattice $\bigoplus_i \mathbf{Z}.\Lambda_i$, the weight space $\bigoplus_i \mathbf{Q}.\Lambda_i$. For example:

```
sage: space = R.weight_space()
sage: space
Weight space over the Rational Field of the Root system of type ['B', 3]


sage: space.base_ring()
Rational Field
sage: list(space.basis())
[Lambda[1], Lambda[2], Lambda[3]]


sage: alpha = space.simple_roots()
sage: alpha[1] + alpha[2]
Lambda[1] + Lambda[2] - 2*Lambda[3]
```

The fundamental weights are the dual basis of the coroots:

```
sage: Lambda = space.fundamental_weights()
sage: Lambda[1]
Lambda[1]


sage: alphacheck = space.simple_coroots()
sage: list(alphacheck)
[alphacheck[1], alphacheck[2], alphacheck[3]]


sage: [Lambda[i].scalar(alphacheck[1]) for i in space.index_set()]
[1, 0, 0]
sage: [Lambda[i].scalar(alphacheck[2]) for i in space.index_set()]
[0, 1, 0]
sage: [Lambda[i].scalar(alphacheck[3]) for i in space.index_set()]
[0, 0, 1]
```

Let us use the simple reflections. In the weight space, they work as in the *number game*: firing the node i on an element x adds c times the simple root α_i, where c is the coefficient of i in x:

```
sage: s = space.simple_reflections()
sage: Lambda[1].simple_reflection(1)
-Lambda[1] + Lambda[2]
sage: Lambda[2].simple_reflection(1)
Lambda[2]
sage: Lambda[3].simple_reflection(1)
Lambda[3]
sage: (-2*Lambda[1] + Lambda[2] + Lambda[3]).simple_reflection(1)
2*Lambda[1] - Lambda[2] + Lambda[3]
```

It can be convenient to manipulate the simple reflections themselves:

```
sage: s = space.simple_reflections()
sage: s[1](Lambda[1])
-Lambda[1] + Lambda[2]
sage: s[1](Lambda[2])
Lambda[2]
sage: s[1](Lambda[3])
Lambda[3]
```

The root system may also come equipped with an ambient space, that is a simultaneous realization of the weight lattice and the coroot lattice in a Euclidean vector space. This is implemented on a type by type basis, and is not always available. When the coefficients permit it, this is also available as an ambient lattice.

TODO: Demo: signed permutations realization of type B

The root system is aware of its dual root system:

```
sage: R.dual
Dual of root system of type ['B', 3]
```

R.dual is really the root system of type C_3:

```
sage: R.dual.cartan_type()
['C', 3]
```

And the coroot lattice that we have been manipulating before is really implemented as the root lattice of the dual root system:

```
sage: R.dual.root_lattice()
Coroot lattice of the Root system of type ['B', 3]
```

In particular, the coroots for the root lattice are in fact the roots of the coroot lattice:

```
sage: list(R.root_lattice().simple_coroots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
sage: list(R.coroot_lattice().simple_roots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
sage: list(R.dual.root_lattice().simple_roots())
[alphacheck[1], alphacheck[2], alphacheck[3]]
```

The coweight lattice and space are defined similarly. Note that, to limit confusion, all the output have been tweaked appropriately.

TESTS:

```
sage: R = RootSystem(['C',3])
sage: R == loads(dumps(R))
True
sage: L = R.ambient_space()
sage: s = L.simple_reflections()
sage: s = L.simple_projections() # todo: not implemented
sage: L == loads(dumps(L))
True
sage: L = R.root_space()
sage: s = L.simple_reflections()
sage: L == loads(dumps(L))
True


sage: all(RootSystem(T).check() for T in CartanType.samples(finite=True,crystalographic=True))
True
```

ambient_lattice()

Returns the usual ambient lattice for this root_system, if it exists and is implemented, and None otherwise. This is a $\mathbf{Z}$-module, endowed with its canonical euclidean scalar product, which embeds simultaneously the root lattice and the coroot lattice (what about the weight lattice?)

EXAMPLES:

```
sage: RootSystem(['A',4]).ambient_lattice()
Ambient lattice of the Root system of type ['A', 4]

sage: RootSystem(['B',4]).ambient_lattice()
sage: RootSystem(['C',4]).ambient_lattice()
sage: RootSystem(['D',4]).ambient_lattice()
sage: RootSystem(['E',6]).ambient_lattice()
sage: RootSystem(['F',4]).ambient_lattice()
sage: RootSystem(['G',2]).ambient_lattice()
```

ambient_space()

Returns the usual ambient space for this root_system, if it is implemented, and None otherwise. This is a $\mathbf{Q}$-module, endowed with its canonical euclidean scalar product, which embeds simultaneously the root lattice and the coroot lattice (what about the weight lattice?). An alternative base ring can be provided as an option; it must contain the smallest ring over which the ambient space can be defined ($\mathbf{Z}$ or $\mathbf{Q}$, depending on the type).

EXAMPLES:

```
sage: RootSystem(['A',4]).ambient_space()
Ambient space of the Root system of type ['A', 4]

sage: RootSystem(['B',4]).ambient_space()
Ambient space of the Root system of type ['B', 4]

sage: RootSystem(['C',4]).ambient_space()
Ambient space of the Root system of type ['C', 4]

sage: RootSystem(['D',4]).ambient_space()
Ambient space of the Root system of type ['D', 4]

sage: RootSystem(['E',6]).ambient_space()
Ambient space of the Root system of type ['E', 6]

sage: RootSystem(['F',4]).ambient_space()
Ambient space of the Root system of type ['F', 4]
```

```
sage: RootSystem(['G',2]).ambient_space()
Ambient space of the Root system of type ['G', 2]
```

cartan_matrix()

EXAMPLES:

```
sage: RootSystem(['A',3]).cartan_matrix()
[ 2 -1  0]
[-1  2 -1]
[ 0 -1  2]
```

cartan_type()

Returns the Cartan type of the root system.

EXAMPLES:

```
sage: R = RootSystem(['A',3])
sage: R.cartan_type()
['A', 3]
```

check()

Runs the checks on the root system.

EXAMPLES:

```
sage: RootSystem(["A",3]).check()
True
```

coroot_lattice()

Returns the coroot lattice associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).coroot_lattice()
Coroot lattice of the Root system of type ['A', 3]
```

coroot_space(*base_ring=Rational Field*)

Returns the coroot space associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).coroot_space()
Coroot space over the Rational Field of the Root system of type ['A', 3]
```

coweight_lattice()

Returns the coweight lattice associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).coweight_lattice()
Coweight lattice of the Root system of type ['A', 3]
```

coweight_space(*base_ring=Rational Field*)

Returns the weight space associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).coweight_space()
Coweight space over the Rational Field of the Root system of type ['A', 3]
```

dynkin_diagram()

Returns the Dynkin diagram of the root system.

EXAMPLES:

```
sage: R = RootSystem(['A',3])
sage: R.dynkin_diagram()
O---O---O
1   2   3
A3
```

index_set()

EXAMPLES:

```
sage: RootSystem(['A',3]).index_set()
[1, 2, 3]
```

is_finite()

Returns True if self is a finite root system.

EXAMPLES:

```
sage: RootSystem(["A",3]).is_finite()
True
sage: RootSystem(["A",3,1]).is_finite()
False
```

is_irreducible()

Returns True if self is an irreducible root system.

EXAMPLES:

```
sage: RootSystem(['A', 3]).is_irreducible()
True
sage: RootSystem("A2xB2").is_irreducible()
False
```

root_lattice()

Returns the root lattice associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).root_lattice()
Root lattice of the Root system of type ['A', 3]
```

root_space()

Returns the root space associated to self.

EXAMPLES

```
sage: RootSystem(['A',3]).root_space()
Root space over the Rational Field of the Root system of type ['A', 3]
```

weight_lattice()

Returns the weight lattice associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).weight_lattice()
Weight lattice of the Root system of type ['A', 3]
```

weight_space()

Returns the weight space associated to self.

EXAMPLES:

```
sage: RootSystem(['A',3]).weight_space()
Weight space over the Rational Field of the Root system of type ['A', 3]
```

WeylDim(*ct, coeffs*)

The Weyl Dimension Formula.

INPUT:

- `type` - a Cartan type
- `coeffs` - a list of nonnegative integers

The length of the list must equal the rank type[1]. A dominant weight hwv is constructed by summing the fundamental weights with coefficients from this list. The dimension of the irreducible representation of the semisimple complex Lie algebra with highest weight vector hwv is returned.

EXAMPLES:

For $SO(7)$, the Cartan type is B_3, so:

```
sage: WeylDim(['B',3],[1,0,0]) # standard representation of SO(7)
7
sage: WeylDim(['B',3],[0,1,0]) # exterior square
21
sage: WeylDim(['B',3],[0,0,1]) # spin representation of spin(7)
8
sage: WeylDim(['B',3],[1,0,1]) # sum of the first and third fundamental weights
48
sage: [WeylDim(['F',4],x) for x in [1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,1]]
[52, 1274, 273, 26]
sage: [WeylDim(['E', 6], x) for x in [0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0], [0, 0, 0, 0, 0, 1],
[1, 78, 27, 351, 351, 351, 27, 650, 351]
```

17.34.6 Weyl Groups

WeylGroup(*x*)

Returns the Weyl group of type type.

INPUT:

- `ct` - a Cartan Type.

EXAMPLES: The following constructions yield the same result, namely a weight lattice and its corresponding Weyl group:

```
sage: G = WeylGroup(['F',4])
sage: L = G.lattice()
```

or alternatively and equivalently:

```
sage: L = RootSystem(['F',4]).ambient_space()
sage: G = L.weyl_group()
```

Either produces a weight lattice, with access to its roots and weights.

```
sage: G = WeylGroup(['F',4])
sage: G.order()
1152
sage: [s1,s2,s3,s4] = G.simple_reflections()
sage: w = s1*s2*s3*s4; w
[ 1/2  1/2  1/2  1/2]
[-1/2  1/2  1/2 -1/2]
[ 1/2  1/2 -1/2 -1/2]
[ 1/2 -1/2  1/2 -1/2]
sage: type(w)
<class 'sage.combinat.root_system.weyl_group.WeylGroupElement'>
```

```
sage: w.order()
12
sage: w.length() # length function on Weyl group
4

sage: L = G.lattice()
sage: fw = L.fundamental_weights(); fw
Finite family {1: (1, 1, 0, 0), 2: (2, 1, 1, 0), 3: (3/2, 1/2, 1/2, 1/2), 4: (1, 0, 0, 0)}
sage: rho = sum(fw); rho
(11/2, 5/2, 3/2, 1/2)
sage: w.action(rho) # action of G on weight lattice
(5, -1, 3, 2)
```

class **WeylGroupElement**(*g*, *parent*)

Class for a Weyl Group elements

action(*v*)

Returns the action of self on the vector v.

EXAMPLES:

```
sage: w = WeylGroup(['A',2])
sage: s1 = w.simple_reflection(1)
sage: v = w.lattice()([1,0,0])
sage: s1.action(v)
(0, 1, 0)
```

lattice()

Returns the ambient lattice associated with self.

EXAMPLES:

```
sage: w = WeylGroup(['A',2])
sage: s1 = w.simple_reflection(1)
sage: s1.lattice()
Ambient space of the Root system of type ['A', 2]
```

length()

Returns the length of self, which is the smallest number of simple reflections into which the element can be decomposed.

EXAMPLES:

```
sage: w = WeylGroup(['A',3])
sage: s1 = w.simple_reflection(1)
sage: s2 = w.simple_reflection(2)
sage: s1.length()
1
sage: (s1*s2).length()
2
```

matrix()

Returns self as a matrix.

EXAMPLES:

```
sage: w = WeylGroup(['A',2])
sage: s1 = w.simple_reflection(1)
sage: m = s1.matrix(); m
[0 1 0]
[1 0 0]
[0 0 1]
```

```
sage: m.parent()
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
```

parent()

Returns self's parent.

EXAMPLES:

```
sage: w = WeylGroup(['A',2])
sage: s1 = w.simple_reflection(1)
sage: s1.parent()
Weyl Group of type ['A', 2] (as a matrix group acting on the ambient space)
```

class **WeylGroup_gens**(*gens, L*)

Class for a Weyl Group with generators (simple reflections)

cartan_type()

Returns the CartanType associated to self.

EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: G.cartan_type()
['F', 4]
```

character_table()

Returns the GAP character table as a string. For larger tables you may preface this with a command such as gap.eval("SizeScreen([120,40])") in order to widen the screen.

EXAMPLES:

```
sage: print WeylGroup(['A',3]).character_table()
CT1
<BLANKLINE>
     2  3  2  2  .  3
     3  1  .  .  1  .
<BLANKLINE>
       1a 4a 2a 3a 2b
<BLANKLINE>
X.1     1 -1 -1  1  1
X.2     3  1 -1  . -1
X.3     2  .  . -1  2
X.4     3 -1  1  . -1
X.5     1  1  1  1  1
```

gens()

Returns the generators of self, i.e. the simple reflections.

EXAMPLES:

```
sage: G = WeylGroup(['A',3])
sage: G.gens()
[[0 1 0 0]
 [1 0 0 0]
 [0 0 1 0]
 [0 0 0 1],
 [1 0 0 0]
 [0 0 1 0]
 [0 1 0 0]
 [0 0 0 1],
 [1 0 0 0]
 [0 1 0 0]
```

```
 [0 0 0 1]
 [0 0 1 0]]
```

lattice()
Returns the ambient lattice of self's type.
EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: G.lattice()
Ambient space of the Root system of type ['F', 4]
```

list()
Returns a list of the elements of self.
EXAMPLES:

```
sage: G = WeylGroup(['A',1])
sage: G.list()
[[0 1]
 [1 0], [1 0]
 [0 1]]
```

long_element()
Returns the long Weyl group element.
EXAMPLES:

```
sage: [WeylGroup(t).long_element().length() for t in ['A',5],['B',3],['C',3],['D',4],['G',2]
[15, 9, 9, 12, 6, 24, 36]
```

simple_reflection(i)
Returns the i^{th} simple reflection.
EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: G.simple_reflection(1)
[1 0 0 0]
[0 0 1 0]
[0 1 0 0]
[0 0 0 1]
```

simple_reflections()
Returns the list of the simple reflections of self.
EXAMPLES:

```
sage: G = WeylGroup(['F',4])
sage: [s1,s2,s3,s4] = G.simple_reflections()
sage: w = s1*s2*s3*s4; w
[ 1/2  1/2  1/2  1/2]
[-1/2  1/2  1/2 -1/2]
[ 1/2  1/2 -1/2 -1/2]
[ 1/2 -1/2  1/2 -1/2]
sage: s4^2==G.unit()
True
sage: type(w)
<class 'sage.combinat.root_system.weyl_group.WeylGroupElement'>
```

unit()
Returns the unit element of the Weyl group
EXAMPLES:

```
sage: e = WeylGroup(['A',3]).unit(); e
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: type(e)
<class 'sage.combinat.root_system.weyl_group.WeylGroupElement'>
```

17.34.7 Weyl Characters

class WeightRing (*A, prefix=None*)

base_ring()
Returns the base ring.
EXAMPLES:

```
sage: R = WeylCharacterRing(['A',3], base_ring = CC); R.base_ring()
Complex Field with 53 bits of precision
```

cartan_type()
Returns the Cartan Type.
EXAMPLES:

```
sage: A2 = WeylCharacterRing("A2")
sage: WeightRing(A2).cartan_type()
['A', 2]
```

lattice()
Returns the weight lattice realization associated to self.
EXAMPLES:

```
sage: E8 = WeylCharacterRing(['E',8])
sage: e8 = WeightRing(E8)
sage: e8.lattice()
Ambient space of the Root system of type ['E', 8]
```

wt_repr(*wt*)
Returns a string representing the irreducible character with highest weight vectr wt.
EXAMPLES:

```
sage: G2 = WeylCharacterRing(['G',2])
sage: g2 = WeightRing(G2)
sage: g2.wt_repr([1,0,0])
'g2(1,0,0)'
```

class WeightRingElement (*A, mdict*)
A class for weights, and linear combinations of weights. See WeightRing? for more information.

cartan_type()
Returns the Cartan Type.
EXAMPLES:

```
sage: A2=WeylCharacterRing("A2")
sage: a2 = WeightRing(A2)
sage: a2([0,1,0]).cartan_type()
['A', 2]
```

character()

Assuming that self is invariant under the Weyl group, this will express it as a linear combination of characters. If self is not Weyl group invariant, this method will not terminate.

EXAMPLES:

```
sage: A2 = WeylCharacterRing(['A',2])
sage: a2 = WeightRing(A2)
sage: W = a2.lattice().weyl_group()
sage: mu = a2(2,1,0)
sage: nu = sum(mu.weyl_group_action(w) for w in W)
sage: nu
a2(0,1,2) + a2(0,2,1) + a2(1,0,2) + a2(1,2,0) + a2(2,0,1) + a2(2,1,0)
sage: nu.character()
-2*A2(1,1,1) + A2(2,1,0)
```

mlist()

Returns a list of weights in self with their multiplicities.

EXAMPLES:

```
sage: G2 = WeylCharacterRing(['G',2])
sage: g2 = WeightRing(G2)
sage: pr = sum(g2(a) for a in g2.lattice().positive_roots())
sage: sorted(pr.mlist())
[[(1, -2, 1), 1],  [(1, -1, 0), 1],  [(1, 1, -2), 1],  [(1, 0, -1), 1],  [(2, -1, -1), 1],
```

weyl_group_action(*w*)

Returns the actionof the Weyl group element w on self.

EXAMPLES:

```
sage: G2 = WeylCharacterRing(['G',2])
sage: g2 = WeightRing(G2)
sage: L = g2.lattice()
sage: [fw1, fw2] = L.fundamental_weights()
sage: sum(g2(fw2).weyl_group_action(w) for w in L.weyl_group())
2*g2(-2,1,1) + 2*g2(-1,-1,2) + 2*g2(-1,2,-1) + 2*g2(1,-2,1) + 2*g2(1,1,-2) + 2*g2(2,-1,-1)
```

class WeylCharacter(*A, hdict, mdict*)

A class for Weyl Characters. Let K be a compact Lie group, which we assume is semisimple and simply-connected. Its complexified Lie algebra L is the Lie algebra of a complex analytic Lie group G. The following three categories are equivalent: representations of K; representations of L; and analytic representations of G. In every case, there is a parametrization of the irreducible representations by their highest weight vectors. For this theory of Weyl, see (for example) J. F. Adams, Lectures on Lie groups; Broecker and Tom Dieck, Representations of Compact Lie groups; Bump, Lie Groups, Part I; Fulton and Harris, Representation Theory, Part IV; Goodman and Wallach, Representations and Invariants of the Classical Groups, Chapter 5; Hall, Lie Groups, Lie Algebras and Representations; Humphreys, Introduction to Lie Algebras and their representations; Procesi, Lie Groups; Samelson, Notes on Lie Algebras; Varadarajan, Lie groups, Lie algebras, and their representations; or Zhelobenko, Compact Lie Groups and their Representations.

There is associated with K, L or G as above a lattice, the weight lattice, whose elements (called weights) are characters of a Cartan subgroup or subalgebra. There is an action of the Weyl group W on the lattice, and elements of a fixed fundamental domain for W, the positive Weyl chamber, are called dominant. There is for each representation a unique highest dominant weight that occurs with nonzero multiplicity with respect to a certain partial order, and it is called the highest weight vector.

EXAMPLES:

```
sage: L = RootSystem(['A',2]).ambient_space()
sage: [fw1,fw2] = L.fundamental_weights()
sage: R = WeylCharacterRing(['A',2], prefix="R")
```

```
sage: [R(1),R(fw1),R(fw2)]
[R(0,0,0), R(1,0,0), R(1,1,0)]
```

Here R(1), R(fw1) and R(fw2) are irreducible representations with highest weight vectors 0, and the first two fundamental weights.

A Weyl character is a character (not necessarily irreducible) of a reductive Lie group or algebra. It is represented by a pair of dictionaries. The mdict is a dictionary of weight multiplicities. The hdict is a dictionary of highest weight vectors of the irreducible characters that occur in its decomposition into irreducibles, with the multiplicities in this decomposition.

For type A (also G_2, F_4, E_6 and E_7) we will take as the weight lattice not the weight lattice of the semisimple group, but for a larger one. For type A, this means we are concerned with the representation theory of K=U(n) or G=GL(n,CC) rather than SU(n) or SU(n,CC). This is useful since the representation theory of GL(n) is ubiquitous, and also since we may then represent the fundamental weights (in root_system.py) by vectors with integer entries. If you are only interested in SL(3), say, use WeylCharacterRing(['A',2]) as above but be aware that R([a,b,c]) and R([a+1,b+1,c+1]) represent the same character of SL(3) since R([1,1,1]) is the determinant.

branch(*S, rule='default'*)

Returns the restriction of the character to the subalgebra. If no rule is specified, we will try to specify one.

INPUT:

- `S` - a Weyl character ring for a Lie subgroup or subalgebra
- `rule` - a branching rule.

See branch_weyl_character? for more information about branching rules.

EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: A2 = WeylCharacterRing(['A',2])
sage: [B3(w).branch(A2,rule="levi") for w in B3.lattice().fundamental_weights()]
[A2(0,0,-1) + A2(0,0,0) + A2(1,0,0),
 A2(0,-1,-1) + A2(0,0,-1) + A2(0,0,0) + A2(1,0,-1) + A2(1,0,0) + A2(1,1,0),
 A2(-1/2,-1/2,-1/2) + A2(1/2,-1/2,-1/2) + A2(1/2,1/2,-1/2) + A2(1/2,1/2,1/2)]
```

cartan_type()

Returns the Cartan Type.

EXAMPLES:

```
sage: A2=WeylCharacterRing("A2")
sage: A2([1,0,0]).cartan_type()
['A', 2]
```

check(*verbose=False*)

To check the correctness of an element, we compare the theoretical dimension computed Weyl character formula with the actual one obtained by adding up the multiplicities.

EXAMPLES:

```
sage: F4 = WeylCharacterRing(['F',4])
sage: [F4(x).check(verbose = true) for x in F4.lattice().fundamental_weights()]
[[52, 52], [1274, 1274], [273, 273], [26, 26]]
```

degree()

The degree of the character, that is, the dimension of module.

EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: [B3(x).degree() for x in B3.lattice().fundamental_weights()]
[7, 21, 8]
```

hlist()
Returns a list of highest weight vectors and multiplicities of the irreducible characters in self.
EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: B3(1/2,1/2,1/2).hlist()
[[(1/2, 1/2, 1/2), 1]]
```

mlist()
Returns a list of weights in self with their multiplicities.
EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: sorted(B3(1/2,1/2,1/2).mlist())
[[(-1/2, -1/2, -1/2), 1],  [(-1/2, -1/2, 1/2), 1],  [(-1/2, 1/2, -1/2), 1],  [(-1/2, 1/2, 1/
```

parent()
EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: B3(2).parent()
The Weyl Character Ring of Type [B,3] with Integer Ring coefficients
```

WeylCharacterRing(*ct, base_ring=Integer Ring, prefix=None, cache=False*)
A class for rings of Weyl characters. The Weyl character is a character of a semisimple (or reductive) Lie group or algebra. They form a ring, in which the addition and multiplication correspond to direct sum and tensor product of representations.
INPUT:

- `ct` - The Cartan Type

OPTIONAL ARGUMENTS:

- `base_ring` - (default: **Z**)
- `prefix` (default an automatically generated prefix based on Cartan type)
- `cache` - (default False) setting cache = True is a substantial speedup at the expense of some memory.

If no prefix specified, one is generated based on the Cartan type. It is good to name the ring after the prefix, since then it can parse its own output.
EXAMPLES:

```
sage: G2 = WeylCharacterRing(['G',2])
sage: [fw1,fw2] = G2.lattice().fundamental_weights()
sage: 2*G2(2*fw1+fw2)
2*G2(4,-1,-3)
sage: 2*G2(4,-1,-3)
2*G2(4,-1,-3)
sage: G2(4,-1,-3).degree()
189
```

Note that since the ring was named G_2 after its default prefix, it was able to parse its own output. You do not have to use the default prefix. Thus:
EXAMPLES:

```
sage: R = WeylCharacterRing(['B',3], prefix='R')
sage: chi = R(R.lattice().fundamental_weights()[3]); chi
R(1/2,1/2,1/2)
sage: R(1/2,1/2,1/2) == chi
True
```

The multiplication in R corresponds to the product of characters, which you can use to determine the decomposition of tensor products into irreducibles. For example, let us compute the tensor product of the standard and spin representations of Spin(7).

EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: [fw1,fw2,fw3]=B3.lattice().fundamental_weights()
sage: [B3(fw1).degree(),B3(fw3).degree()]
[7, 8]
sage: B3(fw1)*B3(fw3)
B3(1/2,1/2,1/2) + B3(3/2,1/2,1/2)
```

The name of the irreducible representation encodes the highest weight vector.

TESTS:

```
sage: F4 = WeylCharacterRing(['F',4], cache = True)
sage: [fw1,fw2,fw3,fw4] = F4.lattice().fundamental_weights()
sage: chi = F4(fw4); chi, chi.degree()
(F4(1,0,0,0), 26)
sage: chi^2
F4(0,0,0,0) + F4(1,0,0,0) + F4(1,1,0,0) + F4(3/2,1/2,1/2,1/2) + F4(2,0,0,0)
sage: [x.degree() for x in [F4(0,0,0,0), F4(1,0,0,0), F4(1,1,0,0), F4(3/2,1/2,1/2,1/2), F4(2,0,0
[1, 26, 52, 273, 324]
```

You can produce a list of the irreducible elements of an irreducible character.

EXAMPLES:

```
sage: R = WeylCharacterRing(['A',2], prefix = R)
sage: sorted(R([2,1,0]).mlist())
[[(1, 1, 1), 2],  [(1, 2, 0), 1],  [(1, 0, 2), 1],  [(2, 1, 0), 1],  [(2, 0, 1), 1],  [(0, 1, 2)
```

class **WeylCharacterRing_class**(*ct, base_ring, prefix, cache*)

base_ring()

Returns the base ring.

EXAMPLES:

```
sage: R = WeylCharacterRing(['A',3], base_ring = CC); R.base_ring()
Complex Field with 53 bits of precision
```

cartan_type()

Returns the Cartan Type.

EXAMPLES:

```
sage: WeylCharacterRing("A2").cartan_type()
['A', 2]
```

char_from_weights(*mdict*)

Return the hdict given just a dictionary of weight multiplicities.

This will not terminate unless the dictionary of weight multiplicities is Weyl group invariant.

EXAMPLES:

```
sage: B3 = WeylCharacterRing(['B',3])
sage: m = B3(1,0,0)._mdict
sage: B3.char_from_weights(m)
{(1, 0, 0): 1}
```

irr_repr(*hwv*)

Return a string representing the irreducible character with highest weight vectr hwv.

EXAMPLES:

```
sage: A2 = WeylCharacterRing(['A',2])
sage: A2.irr_repr([2,1,0])
'A2(2,1,0)'
```

lattice()

Returns the weight lattice associated to self.

EXAMPLES:

```
sage: WeylCharacterRing(['E',8]).lattice()
Ambient space of the Root system of type ['E', 8]
```

branch_weyl_character(*chi, R, S, rule='default'*)

A Branching rule describes the restriction of representations from a Lie group or algebra G to a smaller one. See for example, R. C. King, Branching rules for classical Lie groups using tensor and spinor methods. J. Phys. A 8 (1975), 429-449 or Howe, Tan and Willenbring, Stable branching rules for classical symmetric pairs, Trans. Amer. Math. Soc. 357 (2005), no. 4, 1601-1626.

INPUT:

- `chi` - a character of G
- `R` - the Weyl Character Ring of G
- `S` - the Weyl Character Ring of H
- `rule` - a set of r dominant weights in H where r is the rank of G.

You may use a predefined rule by specifying rule = one of"levi", "automorphic", "symmetric", "extended", "triality" or "miscellaneous". The use of these rules will be explained next. After the examples we will explain how to write your own branching rules for cases that we have omitted. (Rules for the exceptional groups have not yet been coded.)

To explain the predefined rules we survey the most important branching rules. These may be classified into several cases, and once this is understood, the detailed classification can be read off from the Dynkin diagrams. Dynkin classified the maximal subgroups of Lie groups in Mat. Sbornik N.S. 30(72):349-462 (1952).

We will list these for the cases where the Dynkin diagram of S is connected. This excludes branching rules such as A3 -> A1 x A1, which are not yet implemented.

LEVI TYPE. These can be read off from the Dynkin diagram. If removing a node from the Dynkin diagram produces another Dynkin diagram, there is a branching rule. Currently we require that the smaller diagram be connected. For these rules use the option rule="levi":

```
['A',r] -> ['A',r-1]
['B',r] -> ['A',r-1]
['B',r] -> ['B',r-1]
['C',r] -> ['A',r-1]
['C',r] -> ['C',r-1]
['D',r] -> ['A',r-1]
['D',r] -> ['D',r-1]
['E',r] -> ['A',r-1] r = 6,7,8 (not implemented yet)
['E',r] -> ['D',r-1] r = 6,7,8 (not implemented yet)
['E',r] -> ['E',r-1] r = 6,7 (not implemented yet)
['F',4] -> ['B',3] (not implemented yet)
['F',4] -> ['C',3] (not implemented yet)
['G',2] -> ['A',1] (short root) (not implemented yet)
```

The other Levi branching rule from G_2 -> A_1 corresponding to the long root is available by first branching G_2 -> A_2 then branching to A1.

AUTOMORPHIC TYPE. If the Dynkin diagram has a symmetry, then there is an automorphism that is a special case of a branching rule. There is also an exotic"triality" automorphism of D4 having order 3. Use rule="automorphic" or rule="triality"

['A',r] - ['A',r] ['D',r] - ['D',r] ['E',6] - ['E',6] (not implemented yet)

SYMMETRIC TYPE. Related to the automorphic type, when the Dynkin diagram has a symmetry there is a branching rule to the subalgebra (or subgroup) of invariants under the automorphism. Use rule="symmetric".

['A',2r+1] - ['B',r] ['A',2r] - ['C',r] ['D',r] - ['B',r-1] ['E',6] - ['F',4] (not implemented yet) ['D',4] - ['G',2] (not implemented yet)

EXTENDED TYPE. If removing a node from the extended Dynkin diagram results in a Dynkin diagram, then there is a branching rule. Use rule="extended" for these.

['G',2] - ['A',2] (not implemented yet) ['B',r] - ['D',r] ['F',4] - ['B',4] (not implemented yet) ['E',7] - ['A',7] (not implemented yet) ['E',8] - ['A',8] (not implemented yet)

MISCELLANEOUS: Use rule="miscellaneous" for the following rule, which does not fit into the above framework.

['B',3] - ['G',2] (Not implemented yet.)

ISOMORPHIC TYPE: Although not usually referred to as a branching rule, the effects of the accidental isomorphisms may be handled using rule="isomorphic"

['B',2] - ['C',2] ['C',2] - ['B',2] ['A',3] - ['D',3] ['D',3] - ['A',3]

EXAMPLES: (Levi type)

```
sage: A2 = WeylCharacterRing(['A',2])
sage: B2 = WeylCharacterRing(['B',2])
sage: C2 = WeylCharacterRing(['C',2])
sage: A3 = WeylCharacterRing(['A',3])
sage: B3 = WeylCharacterRing(['B',3])
sage: C3 = WeylCharacterRing(['C',3])
sage: D3 = WeylCharacterRing(['D',3])
sage: A4 = WeylCharacterRing(['A',4])
sage: D4 = WeylCharacterRing(['D',4])
sage: A5 = WeylCharacterRing(['A',5])
sage: D5 = WeylCharacterRing(['D',5])
sage: [B3(w).branch(A2,rule="levi") for w in B3.lattice().fundamental_weights()]
[A2(0,0,-1) + A2(0,0,0) + A2(1,0,0),
 A2(0,-1,-1) + A2(0,0,-1) + A2(0,0,0) + A2(1,0,-1) + A2(1,0,0) + A2(1,1,0),
 A2(-1/2,-1/2,-1/2) + A2(1/2,-1/2,-1/2) + A2(1/2,1/2,-1/2) + A2(1/2,1/2,1/2)]
```

The last example must be understood as follows. The representation of B3 being branched is spin, which is not a representation of SO(7) but of its double cover spin(7). The group A2 is really GL(3) and the double cover of SO(7) induces a cover of GL(3) that is trivial over SL(3) but not over the center of GL(3). The weight lattice for this GL(3) consists of triples (a,b,c) of half integers such that a-b and b-c are in **Z**, and this is reflected in the last decomposition.

```
sage: [C3(w).branch(A2,rule="levi") for w in C3.lattice().fundamental_weights()]
[A2(0,0,-1) + A2(1,0,0),
 A2(0,-1,-1) + A2(1,0,-1) + A2(1,1,0),
 A2(-1,-1,-1) + A2(1,-1,-1) + A2(1,1,-1) + A2(1,1,1)]
sage: [D4(w).branch(A3,rule="levi") for w in D4.lattice().fundamental_weights()]
[A3(0,0,0,-1) + A3(1,0,0,0),
 A3(0,0,-1,-1) + A3(0,0,0,0) + A3(1,0,0,-1) + A3(1,1,0,0),
 A3(1/2,-1/2,-1/2,-1/2) + A3(1/2,1/2,1/2,-1/2),
 A3(-1/2,-1/2,-1/2,-1/2) + A3(1/2,1/2,-1/2,-1/2) + A3(1/2,1/2,1/2,1/2)]
```

```
sage: [B3(w).branch(B2,rule="levi") for w in B3.lattice().fundamental_weights()]
[2*B2(0,0) + B2(1,0), B2(0,0) + 2*B2(1,0) + B2(1,1), 2*B2(1/2,1/2)]
sage: C3 = WeylCharacterRing(['C',3])
sage: [C3(w).branch(C2,rule="levi") for w in C3.lattice().fundamental_weights()]
[2*C2(0,0) + C2(1,0),
 C2(0,0) + 2*C2(1,0) + C2(1,1),
 C2(1,0) + 2*C2(1,1)]
sage: [D5(w).branch(D4,rule="levi") for w in D5.lattice().fundamental_weights()]
[2*D4(0,0,0,0) + D4(1,0,0,0),
 D4(0,0,0,0) + 2*D4(1,0,0,0) + D4(1,1,0,0),
 D4(1,0,0,0) + 2*D4(1,1,0,0) + D4(1,1,1,0),
 D4(1/2,1/2,1/2,-1/2) + D4(1/2,1/2,1/2,1/2),
 D4(1/2,1/2,1/2,-1/2) + D4(1/2,1/2,1/2,1/2)]
```

EXAMPLES: (Automorphic type, including D4 triality)

```
sage: [A3(chi).branch(A3,rule="automorphic") for chi in A3.lattice().fundamental_weights()]
[A3(0,0,0,-1), A3(0,0,-1,-1), A3(0,-1,-1,-1)]
sage: [D4(chi).branch(D4,rule="automorphic") for chi in D4.lattice().fundamental_weights()]
[D4(1,0,0,0), D4(1,1,0,0), D4(1/2,1/2,1/2,1/2), D4(1/2,1/2,1/2,-1/2)]
sage: [D4(chi).branch(D4,rule="triality") for chi in D4.lattice().fundamental_weights()]
[D4(1/2,1/2,1/2,-1/2), D4(1,1,0,0), D4(1/2,1/2,1/2,1/2), D4(1,0,0,0)]
```

EXAMPLES: (Symmetric type)

```
sage: [w.branch(B2,rule="symmetric") for w in [A4(1,0,0,0,0),A4(1,1,0,0,0),A4(1,1,1,0,0),A4(2,0,
[B2(1,0), B2(1,1), B2(1,1), B2(0,0) + B2(2,0)]
sage: [A5(w).branch(C3,rule="symmetric") for w in A5.lattice().fundamental_weights()]
[C3(1,0,0),
 C3(0,0,0) + C3(1,1,0),
 C3(1,0,0) + C3(1,1,1),
 C3(0,0,0) + C3(1,1,0),
 C3(1,0,0)]
sage: [A5(w).branch(D3,rule="symmetric") for w in A5.lattice().fundamental_weights()]
[D3(1,0,0), D3(1,1,0), D3(1,1,-1) + D3(1,1,1), D3(1,1,0), D3(1,0,0)]
sage: [D4(x).branch(B3,rule="symmetric") for x in D4.lattice().fundamental_weights()]
[B3(0,0,0) + B3(1,0,0),
 B3(1,0,0) + B3(1,1,0),
 B3(1/2,1/2,1/2),
 B3(1/2,1/2,1/2)]
```

EXAMPLES: (Extended type)

```
sage: [B3(x).branch(D3,rule="extended") for x in B3.lattice().fundamental_weights()]
[D3(0,0,0) + D3(1,0,0),
 D3(1,0,0) + D3(1,1,0),
 D3(1/2,1/2,-1/2) + D3(1/2,1/2,1/2)]
```

EXAMPLES: (Isomorphic type)

```
sage: [B2(x).branch(C2, rule="isomorphic") for x in B2.lattice().fundamental_weights()]
[C2(1,1), C2(1,0)]
sage: [C2(x).branch(B2, rule="isomorphic") for x in C2.lattice().fundamental_weights()]
[B2(1/2,1/2), B2(1,0)]
sage: [A3(x).branch(D3,rule="isomorphic") for x in A3.lattice().fundamental_weights()]
[D3(1/2,1/2,1/2), D3(1,0,0), D3(1/2,1/2,-1/2)]
sage: [D3(x).branch(A3,rule="isomorphic") for x in D3.lattice().fundamental_weights()]
[A3(1/2,1/2,-1/2,-1/2), A3(1/4,1/4,1/4,-3/4), A3(3/4,-1/4,-1/4,-1/4)]
```

Here A3(x,y,z,w) can be understood as a representation of SL(4). The weights x,y,z,w and x+t,y+t,z+t,w+t represent the same representation of SL(4) - though not of GL(4) - since A3(x+t,y+t,z+t,w+t) is the same as A3(x,y,z,w) tensored with det^t. So as a representation of SL(4), A3(1/4,1/4,1/4,-3/4) is the same as A3(1,1,1,0). The exterior square representation SL(4) -> GL(6) admits an invariant symmetric bilinear form, so is a representation SL(4) -> SO(6) that lifts to an isomorphism SL(4) -> Spin(6). Conversely, there are two isomorphisms SO(6) -> SL(4), of which we've selected one.

You may also write your own rules. We may arrange a Cartan subalgebra U of H to be contained in a Cartan subalgebra T of G. This embedding must be chosen in a particular way - the restriction of the positive Weyl chamber in G.lattice() must be contained in the positive Weyl chamber in H.lattice(). The RULE is the adjoint of this embedding U - T, which is a mapping from the dual space of T, which is the weight space of G, to the weight space of H.

EXAMPLES:

```
sage: A3 = WeylCharacterRing(['A',3])
sage: C2 = WeylCharacterRing(['C',2])
sage: rule = lambda x : [x[0]-x[3],x[1]-x[2]]
sage: branch_weyl_character(A3([1,1,0,0]),A3,C2,rule)
C2(0,0) + C2(1,1)
sage: A3(1,1,0,0).branch(C2, rule) == C2(0,0) + C2(1,1)
True
```

irreducible_character_freudenthal(*hwv, L, debug=False*)

Returns the dictionary of multiplicities for the irreducible character with highest weight lamb. The weight multiplicities are computed by the Freudenthal multiplicity formula. The algorithm is based on recursion relation that is stated, for example, in Humphrey's book on Lie Algebras. The multiplicities are invariant under the Weyl group, so to compute them it would be sufficient to compute them for the weights in the positive Weyl chamber. However after some testing it was found to be faster to compute every weight using the recursion, since the use of the Weyl group is expensive in its current implementation.

INPUT:

- `hwv` - a dominant weight in a weight lattice.
- `L` - the ambient lattice

17.35 Crystals

17.35.1 Crystals

Let T be a CartanType with index set I, and W be a realization of the type T weight lattice.

A type T crystal C is a colored oriented graph equipped with a weight function from the nodes to some realization of the type T weight lattice such that:

- Each edge is colored with a label in $i \in I$.
- For each $i \in I$, each node x has:
  - at most one i-successor $f_i(x)$;
  - at most one i-predecessor $e_i(x)$.

  Furthermore, when they exist,
  - $f_i(x)$.weight() = x.weight() - α_i;
  - $e_i(x)$.weight() = x.weight() + α_i.

This crystal actually models a representation of a Lie algebra if it satisfies some further local conditions due to Stembridge, see J. Stembridge, *A local characterization of simply-laced crystals*, Trans. Amer. Math. Soc. 355 (2003), no. 12, 4807-4823.

EXAMPLES:

We construct the type A_5 crystal on letters (or in representation theoretic terms, the highest weight crystal of type A_5 corresponding to the highest weight Λ_1)

```
sage: C = CrystalOfLetters(['A',5]); C
The crystal of letters for type ['A', 5]
```

It has a single highest weight element:

```
sage: C.highest_weight_vectors()
[1]
```

A crystal is a CombinatorialClass; and we can count and list its elements in the usual way:

```
sage: C.cardinality()
6
sage: C.list()
[1, 2, 3, 4, 5, 6]
```

as well as use it in for loops

```
sage: [x for x in C]
[1, 2, 3, 4, 5, 6]
```

Here are some more elaborate crystals (see their respective documentations):

```
sage: Tens = TensorProductOfCrystals(C, C)
sage: Spin = CrystalOfSpins(['B', 3])
sage: Tab  = CrystalOfTableaux(['A', 3], shape = [2,1,1])
sage: Fast = FastCrystal(['B', 2], shape = [3/2, 1/2])
```

One can get (currently) crude plotting via:

```
sage: Tab.plot()
```

For rank two crystals, there is an alternative method of getting metapost pictures. For more information see C.metapost?

Caveat: this crystal library, although relatively featureful for classical crystals, is still in an early development stage, and the syntax details may be subject to changes.

TODO:

- Vocabulary and conventions:
  - elements or vectors of a crystal?
  - For a classical crystal: connected / highest weight / irreducible
  - ...
- More introductory doc explaining the mathematical background
- Layout instructions for plot() for rank 2 types

- Streamlining the latex output
- Littelmann paths and/or alcove paths (this would give us the exceptional types)
- RestrictionOfCrystal / DirectSumOfCrystals
- Crystal.crystal_morphism
- Affine crystals
- Kirillov-Reshetikhin crystals

Most of the above features (except Littelmann/alcove paths) are in MuPAD-Combinat (see lib/COMBINAT/crystals.mu), which could provide inspiration.

class ClassicalCrystal()

The abstract class of classical crystals

cardinality()

Returns the number of elements of the crystal, using Weyl's dimension formula on each connected component

EXAMPLES:

```
sage: from sage.combinat.crystals.crystals import ClassicalCrystal
sage: C = CrystalOfLetters(['A', 5])
sage: ClassicalCrystal.cardinality(C)
6
```

highest_weight_vector()

Returns the highest weight vector if there is a single one; otherwise, raises an error.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: C.highest_weight_vector()
1
```

highest_weight_vectors()

Returns a list of the highest weight vectors of self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: C.highest_weight_vectors()
[1]

sage: C = CrystalOfLetters(['A',2])
sage: T = TensorProductOfCrystals(C,C,C,generators=[[C(2),C(1),C(1)],[C(1),C(2),C(1)]])
sage: T.highest_weight_vectors()
[[2, 1, 1], [1, 2, 1]]
```

list()

The default implementation of list which builds the list from the iterator.

EXAMPLES:

```
sage: class C(CombinatorialClass):
...       def __iter__(self):
...           return iter([1,2,3])
...
sage: C().list() #indirect doctest
[1, 2, 3]
```

class Crystal()

The abstract class of crystals

Derived subclasses should implement the following:

- either a method cartan_type or an attribute _cartan_type
- module_generators: a list (or container) of distinct elements which generate the crystal using f_i

Lambda()

Returns the fundamentals weights in the weight lattice realization for the root system associated to self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A', 5])
sage: C.Lambda()
Finite family {1: (1, 0, 0, 0, 0, 0), 2: (1, 1, 0, 0, 0, 0), 3: (1, 1, 1, 0, 0, 0), 4: (1, 1
```

cartan_type()

Returns the Cartan type of the associated crystal

EXAMPLES:: sage: C = CrystalOfLetters(['A',2]) sage: C.cartan_type() ['A', 2]

character(*R*)

INPUT: R, a WeylCharacterRing. Produces the character of the crystal.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',2])
sage: T = TensorProductOfCrystals(C, C)
sage: A2 = WeylCharacterRing(C.cartan_type()); A2
The Weyl Character Ring of Type [A,2] with Integer Ring coefficients
sage: chi = T.character(A2); chi
A2(1,1,0) + A2(2,0,0)
sage: chi.check(verbose = true)
[9, 9]
```

check()

Runs sanity checks on the crystal:

- Checks that count, list, and __iter__ are consistent. For a ClassicalCrystal, this in particular checks that the number of elements returned by the brute force listing and the iterator __iter__ are consistent with the Weyl dimension formula.
- Should check Stembridge's rules, etc.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A', 5])
sage: C.check()
True
```

digraph()

Returns the DiGraph associated to self.

EXAMPLES:

```
sage: from sage.combinat.crystals.crystals import Crystal
sage: C = CrystalOfLetters(['A', 5])
sage: Crystal.digraph(C)
Digraph on 6 vertices
```

dot_tex()

Returns a dot_tex version of self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',2])
sage: C.dot_tex()
'digraph G { \n  node [ shape=plaintext ];\n  N_0 [ label = " ", texlbl = "$\\text{1}$" ];\n
```

index_set()

Returns the index set of the Dynkin diagram underlying the given crystal

EXAMPLES: sage: C = CrystalOfLetters(['A', 5]) sage: C.index_set() [1, 2, 3, 4, 5]

latex()

Returns the crystal graph as a bit of latex. This can be exported to a file with self.latex_file('filename').

This requires dot2tex to be installed in sage-python.

Here some tips for installation:

- Install graphviz = 2.14
- Download pyparsing-1.4.11.tar.gz pydot-0.9.10.tar.gz dot2tex-2.7.0.tar.gz (see the dot2tex web page for download links) (note that the most recent version of pydot may not work. Be sure to install the 0.9.10 version.) Install each of them using the standard python install, but using sage-python:

```
# FIX ACCORDING TO YOUR Sage INSTALL
export sagedir=/opt/sage/
export sagepython=$sagedir/local/bin/sage-python

# Use downloaded version nums
for package in pyparsing-1.4.11 pydot-0.9.10 dot2tex-2.7.0; do\
        tar zxvf $package.tar.gz;\
        cd $package;\
        sudo $sagepython setup.py install;\
        cd ..;\
    done
```

- Install pgf-2.00 inside your latex tree In short:
  - untaring in /usr/share/texmf/tex/generic
  - clean out remaining pgf files from older version
  - run texhash

You should be done! To test, go to the dot2tex-2.7.0/examples directory, and type:

```
$sagedir//local/bin/dot2tex balls.dot > balls.tex
pdflatex balls.tex
open balls.pdf \#your favorite viewer here
```

EXAMPLES:

```
sage: C = CrystalOfLetters(['A', 5])
sage: C.latex() #optional requires dot2tex
...
```

latex_file(*filename*)

Exports a file, suitable for pdflatex, to 'filename'. This requires a proper installation of dot2tex in sage-python. For more information see the documentation for self.latex().

EXAMPLES:

```
sage: C = CrystalOfLetters(['A', 5])
sage: C.latex_file('/tmp/test.tex') #optional requires dot2tex
```

list()

Returns a list of the elements of self obtained by continually apply the f_i operators to the module generators of self.

EXAMPLES:

```
sage: from sage.combinat.crystals.crystals import Crystal
sage: C = CrystalOfLetters(['A', 5])
sage: l = Crystal.list(C)
sage: l.sort(); l
[1, 2, 3, 4, 5, 6]
```

metapost (*filename, thicklines=False, labels=True, scaling_factor=1.0, tallness=1.0*)

Use C.metapost("filename.mp",[options]) where options can be:

thicklines = True (for thicker edges) labels = False (to suppress labeling of the vertices) scaling_factor=value, where value is a floating point number, 1.0 by default. Increasing or decreasing the scaling factor changes the size of the image. tallness=1.0. Increasing makes the image taller without increasing the width.

Root operators e(1) or f(1) move along red lines, e(2) or f(2) along green. The highest weight is in the lower left. Vertices with the same weight are kept close together. The concise labels on the nodes are strings introduced by Berenstein and Zelevinsky and Littelmann; see Littelmann's paper Cones, Crystals, Patterns, sections 5 and 6.

For Cartan types B2 or C2, the pattern has the form

a2 a3 a4 a1

where c*a2 = a3 = 2*a4 =0 and a1=0, with c=2 for B2, c=1 for C2. Applying e(2) a1 times, e(1) a2 times, e(2) a3 times, e(1) a4 times returns to the highest weight. (Observe that Littelmann writes the roots in opposite of the usual order, so our e(1) is his e(2) for these Cartan types.) For type A2, the pattern has the form

a3 a2 a1

where applying e(1) a1 times, e(2) a2 times then e(3) a1 times returns to the highest weight. These data determine the vertex and may be translated into a Gelfand-Tsetlin pattern or tableau.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A', 2])
sage: C.metapost('/tmp/test.mp') #optional

sage: C = CrystalOfLetters(['A', 5])
sage: C.metapost('/tmp/test.mp')
...
NotImplementedError
```

plot (***options*)

Returns the plot of self as a directed graph.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A', 5])
sage: show_default(False) #do not show the plot by default
sage: C.plot()
Graphics object consisting of 17 graphics primitives
```

weight_lattice_realization()

Returns the weight lattice realization for the root system associated to self. This default implementation uses the ambient space of the root system.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A', 5])
sage: C.weight_lattice_realization()
Ambient space of the Root system of type ['A', 5]
```

class **CrystalBacktracker** (*crystal*)

class **CrystalElement**()

The abstract class of crystal elements

Sub classes should implement at least:

- x.e(i) (returning $e_i(x)$)
- x.f(i) (returning $f_i(x)$)
- x.weight()

Epsilon()

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: C(0).Epsilon()
(0, 0, 0, 0, 0, 0)
sage: C(1).Epsilon()
(0, 0, 0, 0, 0, 0)
sage: C(2).Epsilon()
(1, 0, 0, 0, 0, 0)
```

Phi()

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: C(0).Phi()
(0, 0, 0, 0, 0, 0)
sage: C(1).Phi()
(1, 0, 0, 0, 0, 0)
sage: C(2).Phi()
(1, 1, 0, 0, 0, 0)
```

cartan_type()

Returns the cartan type associated to self

EXAMPLES:: sage: C = CrystalOfLetters(['A', 5]) sage: C(1).cartan_type() ['A', 5]

e(i)

Returns $e_i(x)$ if it exists or None otherwise. This is to be implemented by subclasses of CrystalElement.

TESTS:

```
sage: from sage.combinat.crystals.crystals import CrystalElement
sage: C = CrystalOfLetters(['A',5])
sage: CrystalElement.e(C(1), 1)
...
NotImplementedError
```

epsilon(i)

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: C(1).epsilon(1)
0
sage: C(2).epsilon(1)
1
```

f(i)

Returns $f_i(x)$ if it exists or None otherwise. This is to be implemented by subclasses of CrystalElement.

TESTS:

```
sage: from sage.combinat.crystals.crystals import CrystalElement
sage: C = CrystalOfLetters(['A',5])
sage: CrystalElement.f(C(1), 1)
...
NotImplementedError
```

index_set()
EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: C(1).index_set()
[1, 2, 3, 4, 5]
```

is_highest_weight()
Returns True if self is a highest weight.
EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: C(1).is_highest_weight()
True
sage: C(2).is_highest_weight()
False
```

phi(i)
EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: C(1).phi(1)
1
sage: C(2).phi(1)
0
```

s(i)
Returns the reflection of self along its i-string
EXAMPLES:

```
sage: C = CrystalOfTableaux(['A',2], shape=[2,1])
sage: b=C(rows=[[1,1],[3]])
sage: b.s(1)
[[2, 2], [3]]
sage: b=C(rows=[[1,2],[3]])
sage: b.s(2)
[[1, 2], [3]]
sage: T=CrystalOfTableaux(['A',2],shape=[4])
sage: t=T(rows=[[1,2,2,2]])
sage: t.s(1)
[[1, 1, 1, 2]]
```

weight()
EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: C(1).weight()
(1, 0, 0, 0, 0, 0)
```

17.35.2 Crystals of letters

class ClassicalCrystalOfLetters(*cartan_type, element_class, element_print_style=None, dual=None*)
A generic class for classical crystals of letters.

All classical crystals of letters should be instances of this class or of subclasses. To define a new crystal of letters, one only needs to implement a class for the elements (which subclasses Letter and CrystalElement), with appropriate e and f operations. If the module generator is not 1, one also needs to define the subclass ClassicalCrystalOfLetters for the crystal itself.

The basic assumption is that crystals of letters are small, but used intensively as building blocks. Therefore, we explicitly build in memory the list of all elements, the crystal graph and its transitive closure, so as to make the following operations constant time: list, cmp, (todo: phi, epsilon, e, f with caching)

digraph()

Returns the directed graph associated to self.

EXAMPLES:

```
sage: CrystalOfLetters(['A',5]).digraph()
Digraph on 6 vertices
```

list()

Returns a list of the elements of self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: C.list()
[1, 2, 3, 4, 5, 6]
```

lt_elements(*x, y*)

Returns True if and only if there is a path from x to y in the crystal graph, when x is not equal to y.

Because the crystal graph is classical, it is a directed acyclic graph which can be interpreted as a poset. This function implements the comparison function of this poset.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A', 5])
sage: x = C(1)
sage: y = C(2)
sage: C.lt_elements(x,y)
True
sage: C.lt_elements(y,x)
False
sage: C.lt_elements(x,x)
False
sage: C = CrystalOfLetters(['D', 4])
sage: C.lt_elements(C(4),C(-4))
False
sage: C.lt_elements(C(-4),C(4))
False
```

CrystalOfLetters(*cartan_type, element_print_style=None, dual=None*)

Returns the crystal of letters of the given type.

For classical types, this is a combinatorial model for the crystal with highest weight Lambda_1 (the first fundamental weight).

Any irreducible classical crystal appears as the irreducible component of the tensor product of several copies of this crystal (plus possibly one copy of the spin crystal, see CrystalOfSpins). See M. Kashiwara, T. Nakashima, *Crystal graphs for representations of the 'q'-analogue of classical Lie algebras*, J. Algebra **165** (1994), no. 2, 295-345. Elements of this irreducible component have a fixed shape, and can be fit inside a tableau shape. Otherwise said, any irreducible classical crystal is isomorphic to a crystal of tableaux with cells filled by elements of the crystal of letters (possibly tensored with the crystal of spins).

INPUT:

- `T` - A CartanType

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: C.list()
[1, 2, 3, 4, 5, 6]
sage: C.cartan_type()
['A', 5]
```

For type E6, one can also specify how elements are printed. This option is usually set to None and the default representation is used. If one chooses the option 'compact', the elements are printed in the more compact convention with 27 letters +abcdefghijklmnopqrstuvwxyz and the 27 letters -ABCDEFGHIJKLMNOPQRSTUVWXYZ for the dual crystal.

EXAMPLES:

```
sage: C = CrystalOfLetters(['E',6], element_print_style = 'compact')
sage: C.list()
[+, a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z]
sage: C = CrystalOfLetters(['E',6], element_print_style = 'compact', dual = True)
sage: C.list()
[-, A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V, W, X, Y, Z]
```

class **Crystal_of_letters_type_A_element** (*parent, value*)

Type A crystal of letters elements

TESTS:

```
sage: C = CrystalOfLetters (['A',3])
sage: C.list()
[1, 2, 3, 4]
sage: [ [x < y for y in C] for x in C ]
[[False, True, True, True],
 [False, False, True, True],
 [False, False, False, True],
 [False, False, False, False]]


sage: C = CrystalOfLetters(['A',5])
sage: C(1) < C(1), C(1) < C(2), C(1) < C(3), C(2) < C(1)
(False, True, True, False)


sage: C.check()
True
```

e (*i*)

Returns the action of e_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',4])
sage: [[c,i,c.e(i)] for i in C.index_set() for c in C if c.e(i) is not None]
[[2, 1, 1], [3, 2, 2], [4, 3, 3], [5, 4, 4]]
```

f (*i*)

Returns the action of f_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',4])
sage: [[c,i,c.f(i)] for i in C.index_set() for c in C if c.f(i) is not None]
[[1, 1, 2], [2, 2, 3], [3, 3, 4], [4, 4, 5]]
```

weight()
Returns the weight of self.
EXAMPLES:

```
sage: [v.weight() for v in CrystalOfLetters(['A',3])]
[(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)]
```

class **Crystal_of_letters_type_B_element**(*parent, value*)
Type B crystal of letters elements
TESTS:

```
sage: C = CrystalOfLetters (['B',3])
sage: C.check()
True
```

e(*i*)
Returns the action of e_i on self.
EXAMPLES:

```
sage: C = CrystalOfLetters(['B',4])
sage: [[c,i,c.e(i)] for i in C.index_set() for c in C if c.e(i) is not None]
[[2, 1, 1],
 [-1, 1, -2],
 [3, 2, 2],
 [-2, 2, -3],
 [4, 3, 3],
 [-3, 3, -4],
 [0, 4, 4],
 [-4, 4, 0]]
```

f(*i*)
Returns the actions of f_i on self.
EXAMPLES:

```
sage: C = CrystalOfLetters(['B',4])
sage: [[c,i,c.f(i)] for i in C.index_set() for c in C if c.f(i) is not None]
[[1, 1, 2],
 [-2, 1, -1],
 [2, 2, 3],
 [-3, 2, -2],
 [3, 3, 4],
 [-4, 3, -3],
 [4, 4, 0],
 [0, 4, -4]]
```

weight()
Returns the weight of self.
EXAMPLES:

```
sage: [v.weight() for v in CrystalOfLetters(['B',3])]
[(1, 0, 0),
 (0, 1, 0),
 (0, 0, 1),
 (0, 0, 0),
 (0, 0, -1),
 (0, -1, 0),
 (-1, 0, 0)]
```

class **Crystal_of_letters_type_C_element** (*parent, value*)

Type C crystal of letters elements

TESTS:

```
sage: C = CrystalOfLetters (['C',3])
sage: C.list()
[1, 2, 3, -3, -2, -1]
sage: [ [x < y for y in C] for x in C ]
[[False, True, True, True, True, True],
 [False, False, True, True, True, True],
 [False, False, False, True, True, True],
 [False, False, False, False, True, True],
 [False, False, False, False, False, True],
 [False, False, False, False, False, False]]
sage: C.check()
True
```

e (*i*)

Returns the action of e_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['C',4])
sage: [[c,i,c.e(i)] for i in C.index_set() for c in C if c.e(i) is not None]
[[2, 1, 1],
 [-1, 1, -2],
 [3, 2, 2],
 [-2, 2, -3],
 [4, 3, 3],
 [-3, 3, -4],
 [-4, 4, 4]]
```

f (*i*)

Retursn the action of f_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['C',4])
sage: [[c,i,c.f(i)] for i in C.index_set() for c in C if c.f(i) is not None]
[[1, 1, 2],
 [-2, 1, -1],
 [2, 2, 3],
 [-3, 2, -2],
 [3, 3, 4],
 [-4, 3, -3],
 [4, 4, -4]]
```

weight ()

Returns the weight of self.

EXAMPLES:

```
sage: [v.weight() for v in CrystalOfLetters(['C',3])]
[(1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 0, -1), (0, -1, 0), (-1, 0, 0)]
```

class **Crystal_of_letters_type_D_element** (*parent, value*)

Type D crystal of letters elements

TESTS:

```
sage: C = CrystalOfLetters(['D',4])
sage: C.list()
```

```
[1, 2, 3, 4, -4, -3, -2, -1]
sage: C.check()
True
```

e(*i*)

Returns the action of e_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['D',5])
sage: [[c,i,c.e(i)] for i in C.index_set() for c in C if c.e(i) is not None]
[[2, 1, 1],
 [-1, 1, -2],
 [3, 2, 2],
 [-2, 2, -3],
 [4, 3, 3],
 [-3, 3, -4],
 [5, 4, 4],
 [-4, 4, -5],
 [-5, 5, 4],
 [-4, 5, 5]]
```

f(*i*)

Returns the action of f_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['D',5])
sage: [[c,i,c.f(i)] for i in C.index_set() for c in C if c.f(i) is not None]
[[1, 1, 2],
 [-2, 1, -1],
 [2, 2, 3],
 [-3, 2, -2],
 [3, 3, 4],
 [-4, 3, -3],
 [4, 4, 5],
 [-5, 4, -4],
 [4, 5, -5],
 [5, 5, -4]]
```

weight()

Returns the weight of self.

EXAMPLES:

```
sage: [v.weight() for v in CrystalOfLetters(['D',4])]
[(1, 0, 0, 0),
 (0, 1, 0, 0),
 (0, 0, 1, 0),
 (0, 0, 0, 1),
 (0, 0, 0, -1),
 (0, 0, -1, 0),
 (0, -1, 0, 0),
 (-1, 0, 0, 0)]
```

class **Crystal_of_letters_type_E6_element**(*parent, value*)

Type E_6 crystal of letters elements. This crystal corresponds to the highest weight crystal $B(\Lambda_1)$.

TESTS:

```
sage: C = CrystalOfLetters(['E',6])
sage: C.module_generators
```

```
[[1]]
sage: C.list()
[[1], [-1, 3], [-3, 4], [-4, 2, 5], [-2, 5], [-5, 2, 6], [-2, -5, 4, 6],
[-4, 3, 6], [-3, 1, 6], [-1, 6], [-6, 2], [-2, -6, 4], [-4, -6, 3, 5],
[-3, -6, 1, 5], [-1, -6, 5], [-5, 3], [-3, -5, 1, 4], [-1, -5, 4], [-4, 1, 2],
[-1, -4, 2, 3], [-3, 2], [-2, -3, 4], [-4, 5], [-5, 6], [-6], [-2, 1], [-1, -2, 3]]
sage: C.check()
True
sage: all(b.f(i).e(i) == b for i in C.index_set() for b in C if b.f(i) is not None)
True
sage: all(b.e(i).f(i) == b for i in C.index_set() for b in C if b.e(i) is not None)
True
sage: G = C.digraph()
sage: G.show(edge_labels=true, figsize=12, vertex_size=1)
```

e(*i*)

Returns the action of e_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['E',6])
sage: C([-1,3]).e(1)
[1]
sage: C([-2,-3,4]).e(2)
[-3, 2]
sage: C([1]).e(1)
```

f(*i*)

Returns the action of f_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['E',6])
sage: C([1]).f(1)
[-1, 3]
sage: C([-6]).f(1)
```

weight()

Returns the weight of self.

EXAMPLES:

```
sage: [v.weight() for v in CrystalOfLetters(['E',6])]
[(0, 0, 0, 0, 0, -2/3, -2/3, 2/3),
(-1/2, 1/2, 1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
(1/2, -1/2, 1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
(1/2, 1/2, -1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
(-1/2, -1/2, -1/2, 1/2, 1/2, -1/6, -1/6, 1/6),
(1/2, 1/2, 1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
(-1/2, -1/2, 1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
(-1/2, 1/2, -1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
(1/2, -1/2, -1/2, -1/2, 1/2, -1/6, -1/6, 1/6),
(0, 0, 0, 0, 1, 1/3, 1/3, -1/3),
(1/2, 1/2, 1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
(-1/2, -1/2, 1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
(-1/2, 1/2, -1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
(1/2, -1/2, -1/2, 1/2, -1/2, -1/6, -1/6, 1/6),
(0, 0, 0, 1, 0, 1/3, 1/3, -1/3),
(-1/2, 1/2, 1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
(1/2, -1/2, 1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
(0, 0, 1, 0, 0, 1/3, 1/3, -1/3),
```

```
(1/2, 1/2, -1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
(0, 1, 0, 0, 0, 1/3, 1/3, -1/3),
(1, 0, 0, 0, 0, 1/3, 1/3, -1/3),
(0, -1, 0, 0, 0, 1/3, 1/3, -1/3),
(0, 0, -1, 0, 0, 1/3, 1/3, -1/3),
(0, 0, 0, -1, 0, 1/3, 1/3, -1/3),
(0, 0, 0, 0, -1, 1/3, 1/3, -1/3),
(-1/2, -1/2, -1/2, -1/2, -1/2, -1/6, -1/6, 1/6),
(-1, 0, 0, 0, 0, 1/3, 1/3, -1/3)]
```

class **Crystal_of_letters_type_E6_element_dual** (*parent, value*)

Type E_6 crystal of letters elements. This crystal corresponds to the highest weight crystal $B(\Lambda_6)$. This crystal is dual to $B(\Lambda_1)$ of type E_6.

TESTS:

```
sage: C = CrystalOfLetters(['E',6], dual = True)
sage: C.module_generators
[[6]]
sage: all(b==b.retract(b.lift()) for b in C)
True
sage: C.list()
[[6], [5, -6], [4, -5], [2, 3, -4], [3, -2], [1, 2, -3], [2, -1], [1, 4, -2, -3],
[4, -1, -2], [1, 5, -4], [3, 5, -1, -4], [5, -3], [1, 6, -5], [3, 6, -1, -5], [4, 6, -3, -5],
[2, 6, -4], [6, -2], [1, -6], [3, -1, -6], [4, -3, -6], [2, 5, -4, -6], [5, -2, -6], [2, -5],
[4, -2, -5], [3, -4], [1, -3], [-1]]
sage: C.check()
True
sage: all(b.f(i).e(i) == b for i in C.index_set() for b in C if b.f(i) is not None)
True
sage: all(b.e(i).f(i) == b for i in C.index_set() for b in C if b.e(i) is not None)
True
sage: G = C.digraph()
sage: G.show(edge_labels=true, figsize=12, vertex_size=1)
```

e (*i*)

Returns the action of e_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['E',6], dual = True)
sage: C([-1]).e(1)
[1, -3]
```

f (*i*)

Returns the action of f_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['E',6], dual = True)
sage: C([6]).f(6)
[5, -6]
sage: C([6]).f(1)
```

lift ()

Lifts an element of self to the crystal of letters CrystalOfLetters(['E',6]) by taking its inverse weight.

EXAMPLES:

```
sage: C = CrystalOfLetters(['E',6], dual = True)
sage: b=C.module_generators[0]
```

```
sage: b.lift()
[-6]
```

retract(*p*)

Retracts element p, which is an element in CrystalOfLetters(['E',6]) to an element in CrystalOfLetters(['E',6], dual = True) by taking its inverse weight.

EXAMPLES:

```
sage: C = CrystalOfLetters(['E',6])
sage: Cd = CrystalOfLetters(['E',6], dual = True)
sage: b = Cd.module_generators[0]
sage: p = C([-1,3])
sage: b.retract(p)
[1, -3]
sage: b.retract(None)
```

weight()

Returns the weight of self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['E',6], dual = True)
sage: b=C.module_generators[0]
sage: b.weight()
(0, 0, 0, 0, 1, -1/3, -1/3, 1/3)
sage: [v.weight() for v in C]
[(0, 0, 0, 0, 1, -1/3, -1/3, 1/3),
(0, 0, 0, 1, 0, -1/3, -1/3, 1/3),
(0, 0, 1, 0, 0, -1/3, -1/3, 1/3),
(0, 1, 0, 0, 0, -1/3, -1/3, 1/3),
(-1, 0, 0, 0, 0, -1/3, -1/3, 1/3),
(1, 0, 0, 0, 0, -1/3, -1/3, 1/3),
(1/2, 1/2, 1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, -1, 0, 0, 0, -1/3, -1/3, 1/3),
(-1/2, -1/2, 1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, -1, 0, 0, -1/3, -1/3, 1/3),
(-1/2, 1/2, -1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, -1/2, 1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, 0, -1, 0, -1/3, -1/3, 1/3),
(-1/2, 1/2, 1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, 1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, -1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, -1/2, -1/2, 1/2, 1/6, 1/6, -1/6),
(0, 0, 0, 0, -1, -1/3, -1/3, 1/3),
(-1/2, 1/2, 1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, 1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, -1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, -1/2, 1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, 1/2, 1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, -1/2, 1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(-1/2, 1/2, -1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(1/2, -1/2, -1/2, -1/2, -1/2, 1/6, 1/6, -1/6),
(0, 0, 0, 0, 0, 2/3, 2/3, -2/3)]
```

class **Crystal_of_letters_type_G_element**(*parent, value*)

Type G2 crystal of letters elements

TESTS:

```
sage: C = CrystalOfLetters(['G',2])
sage: C.list()
[1, 2, 3, 0, -3, -2, -1]
sage: C.check()
True
```

e(*i*)

Returns the action of e_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['G',2])
sage: [[c,i,c.e(i)] for i in C.index_set() for c in C if c.e(i) is not None]
[[2, 1, 1],
 [0, 1, 3],
 [-3, 1, 0],
 [-1, 1, -2],
 [3, 2, 2],
 [-2, 2, -3]]
```

f(*i*)

Returns the action of f_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['G',2])
sage: [[c,i,c.f(i)] for i in C.index_set() for c in C if c.f(i) is not None]
[[1, 1, 2],
 [3, 1, 0],
 [0, 1, -3],
 [-2, 1, -1],
 [2, 2, 3],
 [-3, 2, -2]]
```

weight()

Returns the weight of self.

EXAMPLES:

```
sage: [v.weight() for v in CrystalOfLetters(['G',2])]
[(1, 0, -1), (1, -1, 0), (0, 1, -1), (0, 0, 0), (0, -1, 1), (-1, 1, 0), (-1, 0, 1)]
```

class Letter(*parent, value*)

A class for letters

parent()

Returns the parent of self.

EXAMPLES:

```
sage: from sage.combinat.crystals.letters import Letter
sage: Letter(ZZ, 1).parent()
Integer Ring
```

17.35.3 Spin Crystals

These are the crystals associated with the three spin representations: the spin representations of odd orthogonal groups (or rather their double covers); and the + and - spin representations of the even orthogonal groups.

We follow Kashiwara and Nakashima (Journal of Algebra 165, 1994) in representing the elements of the spin Crystal by sequences of signs +/-. Two other representations are available as attributes internal_repn and signature of the crystal element.

- A numerical internal representation, an integer N such that if N-1 is written in binary and the 1's are replaced by −, the 0's by +
- The signature, which is a list in which + is replaced by +1 and − by -1.

CrystalOfSpins(*ct*)

Return the spin crystal of the given type B.

This is a combinatorial model for the crystal with highest weight $Lambda_n$ (the n-th fundamental weight). It has 2^n elements, here called Spins. See also CrystalOfLetters, CrystalOfSpinsPlus and CrystalOfSpinsMinus.

INPUT:

- `['B',n]` - A CartanType of type B.

EXAMPLES:

```
sage: C = CrystalOfSpins(['B',3])
sage: C.list()
[[1, 1, 1],
 [1, 1, -1],
 [1, -1, 1],
 [-1, 1, 1],
 [1, -1, -1],
 [-1, 1, -1],
 [-1, -1, 1],
 [-1, -1, -1]]
 sage: C.cartan_type()
 ['B', 3]


sage: [x.signature() for x in C]
['+++', '++-', '+-+', '-++', '+--', '-+-', '--+', '---']
```

TESTS:

```
sage: TensorProductOfCrystals(C,C,generators=[[C.list()[0],C.list()[0]]]).cardinality()
35
```

CrystalOfSpinsMinus(*ct*)

Return the minus spin crystal of the given type D.

This is the crystal with highest weight $Lambda_{n-1}$ (the (n-1)-st fundamental weight).

INPUT:

- `['D',n]` - A CartanType of type D.

EXAMPLES:

```
sage: E = CrystalOfSpinsMinus(['D',4])
sage: E.list()
 [[1, 1, 1, -1],
  [1, 1, -1, 1],
  [1, -1, 1, 1],
  [-1, 1, 1, 1],
  [1, -1, -1, -1],
  [-1, 1, -1, -1],
  [-1, -1, 1, -1],
  [-1, -1, -1, 1]]
sage: [x.signature() for x in E]
['+++-', '++-+', '+-++', '-+++', '+---', '-+--', '--+-', '---+']
```

TESTS:

```
sage: len(TensorProductOfCrystals(E,E,generators=[[E[0],E[0]]]).list())
35
sage: D = CrystalOfSpinsPlus(['D',4])
sage: len(TensorProductOfCrystals(D,E,generators=[[D.list()[0],E.list()[0]]]).list())
56
```

CrystalOfSpinsPlus(*ct*)

Return the plus spin crystal of the given type D.

This is the crystal with highest weight $Lambda_n$ (the n-th fundamental weight).

INPUT:

- `['D',n]` - A CartanType of type D.

EXAMPLES:

```
sage: D = CrystalOfSpinsPlus(['D',4])
sage: D.list()
[[1, 1, 1, 1],
 [1, 1, -1, -1],
 [1, -1, 1, -1],
 [-1, 1, 1, -1],
 [1, -1, -1, 1],
 [-1, 1, -1, 1],
 [-1, -1, 1, 1],
 [-1, -1, -1, -1]]


sage: [x.signature() for x in D]
['++++', '++--', '+-+-', '-++-', '+--+', '-+-+', '--++', '----']
```

TESTS:

```
sage: D.check()
True
```

class GenericCrystalOfSpins(*ct, element_class, case*)

cmp_elements(*x, y*)

Returns True if and only if there is a path from x to y in the crystal graph.

Because the crystal graph is classical, it is a directed acyclic graph which can be interpreted as a poset. This function implements the comparison function of this poset.

EXAMPLES:

```
sage: C = CrystalOfSpins(['B',3])
sage: x = C([1,1,1])
sage: y = C([-1,-1,-1])
sage: C.cmp_elements(x,y)
-1
sage: C.cmp_elements(y,x)
1
sage: C.cmp_elements(x,x)
0
```

digraph()

Returns the directed graph associated to self.

EXAMPLES:

```
sage: CrystalOfSpins(['B',3]).digraph()
Digraph on 8 vertices
```

list()
Returns a list of the elements of self.
EXAMPLES:

```
sage: CrystalOfSpins(['B',3]).list()
[[1, 1, 1],
 [1, 1, -1],
 [1, -1, 1],
 [-1, 1, 1],
 [1, -1, -1],
 [-1, 1, -1],
 [-1, -1, 1],
 [-1, -1, -1]]
```

class Spin(*parent, value*)

parent()
Returns the parent of self.
EXAMPLES:

```
sage: C = CrystalOfSpins(['B',3])
sage: C([1,1,1]).parent()
The crystal of spins for type ['B', 3]
```

signature()
Returns the signature of self.
EXAMPLES:

```
sage: C = CrystalOfSpins(['B',3])
sage: C([1,1,1]).signature()
'+++'
sage: C([1,1,-1]).signature()
'++-'
```

class Spin_crystal_type_B_element(*parent, value*)
Type B spin representation crystal element

e(*i*)
Returns the action of e_i on self.
EXAMPLES:

```
sage: C = CrystalOfSpins(['B',3])
sage: [[C[m].e(i) for i in range(1,4)] for m in range(8)]
[[None, None, None],
 [None, None, [1, 1, 1]],
 [None, [1, 1, -1], None],
 [[1, -1, 1], None, None],
 [None, None, [1, -1, 1]],
 [[1, -1, -1], None, [-1, 1, 1]],
 [None, [-1, 1, -1], None],
 [None, None, [-1, -1, 1]]]
```

f(*i*)
Returns the action of f_i on self.
EXAMPLES:

```
sage: C = CrystalOfSpins(['B',3])
sage: [[C[m].f(i) for i in range(1,4)] for m in range(8)]
[[None, None, [1, 1, -1]],
 [None, [1, -1, 1], None],
 [[-1, 1, 1], None, [1, -1, -1]],
 [None, None, [-1, 1, -1]],
 [[-1, 1, -1], None, None],
 [None, [-1, -1, 1], None],
 [None, None, [-1, -1, -1]],
 [None, None, None]]
```

class **Spin_crystal_type_D_element**(*parent, value*)

Type D spin representation crystal element

e(*i*)

Returns the action of e_i on self.

EXAMPLES:

```
sage: D = CrystalOfSpinsPlus(['D',4])
sage: [[D.list()[m].e(i) for i in range(1,4)] for m in range(8)]
[[None, None, None],
 [None, None, None],
 [None, [1, 1, -1, -1], None],
 [[1, -1, 1, -1], None, None],
 [None, None, [1, -1, 1, -1]],
 [[1, -1, -1, 1], None, [-1, 1, 1, -1]],
 [None, [-1, 1, -1, 1], None],
 [None, None, None]]

sage: E = CrystalOfSpinsMinus(['D',4])
sage: [[E[m].e(i) for i in range(1,4)] for m in range(8)]
[[None, None, None],
 [None, None, [1, 1, 1, -1]],
 [None, [1, 1, -1, 1], None],
 [[1, -1, 1, 1], None, None],
 [None, None, None],
 [[1, -1, -1, -1], None, None],
 [None, [-1, 1, -1, -1], None],
 [None, None, [-1, -1, 1, -1]]]
```

f(*i*)

Returns the action of f_i on self.

EXAMPLES:

```
sage: D = CrystalOfSpinsPlus(['D',4])
sage: [[D.list()[m].f(i) for i in range(1,4)] for m in range(8)]
[[None, None, None],
 [None, [1, -1, 1, -1], None],
 [[-1, 1, 1, -1], None, [1, -1, -1, 1]],
 [None, None, [-1, 1, -1, 1]],
 [[-1, 1, -1, 1], None, None],
 [None, [-1, -1, 1, 1], None],
 [None, None, None],
 [None, None, None]]

sage: E = CrystalOfSpinsMinus(['D',4])
sage: [[E[m].f(i) for i in range(1,4)] for m in range(8)]
[[None, None, [1, 1, -1, 1]],
 [None, [1, -1, 1, 1], None],
```

```
[[-1, 1, 1, 1], None, None],
[None, None, None],
[[-1, 1, -1, -1], None, None],
[None, [-1, -1, 1, -1], None],
[None, None, [-1, -1, -1, 1]],
[None, None, None]]
```

17.35.4 Tensor Products of Crystals

class **CrystalOfTableaux**(*type, shape=None, shapes=None*)

Crystals of tableaux. Input: a Cartan Type type and "shape", a partition of length <= type[1]. Produces a classical crystal with the given Cartan Type and highest weight corresponding to the given shape.

If the type is ['D',r] then the shape is permitted to have a negative value in the r-th position. Thus if shape='[s_1,s_2,...,s_r]' then s_r may be negative but in any case $s1 \geq s2 \geq ...s_{r-1} \geq |s_r|$. This crystal is related to $[s_1, \cdots, |s_r|]$ by the outer automorphism of SO(2r).

Note that crystals of tableaux are constructed using an embedding into tensor products following Kashiwara and Nakashima [Kashiwara, Masaki; Nakashima, Toshiki, *Crystal graphs for representations of the q-analogue of classical Lie algebras*, J. Algebra 165 (1994), no. 2, 295-345.]. Sage's tensor product rule for crystals differs from that of Kashiwara and Nakashima by reversing the order of the tensor factors. Sage produces the same crystals of tableaux as Kashiwara and Nakashima. With Sage's convention, the tensor product of crystals is the same as the monoid operation on tableaux and hence the plactic monoid.

EXAMPLES:

We create the crystal of tableaux for type A_2, with highest weight given by the partition [2,1,1]:

```
sage: Tab = CrystalOfTableaux(['A',3], shape = [2,1,1])
```

Here is the list of its elements:

```
sage: Tab.list()
[[[1, 1], [2], [3]], [[1, 2], [2], [3]], [[1, 3], [2], [3]],
 [[1, 4], [2], [3]], [[1, 4], [2], [4]], [[1, 4], [3], [4]],
 [[2, 4], [3], [4]], [[1, 1], [2], [4]], [[1, 2], [2], [4]],
 [[1, 3], [2], [4]], [[1, 3], [3], [4]], [[2, 3], [3], [4]],
 [[1, 1], [3], [4]], [[1, 2], [3], [4]], [[2, 2], [3], [4]]]
```

One can get (currently) crude plotting via:

```
#         sage: Tab.plot()  # random
```

One can get instead get a LaTeX drawing ready to be copy-pasted into a LaTeX file:

```
#         sage: Tab.latex() # random
```

See sage.combinat.crystals.crystals? for general help on using crystals

Internally, a tableau of a given Cartan type is represented as a tensor product of letters of the same type. The order in which the tensor factors appear is by reading the columns of the tableaux left to right, top to bottom (in French notation). As an example:

```
sage: T = CrystalOfTableaux(['A',2], shape = [3,2])
sage: T.module_generators[0]
[[1, 1, 1], [2, 2]]
sage: T.module_generators[0]._list
[2, 1, 2, 1, 1]
```

To create a tableau, one can use:

```
sage: Tab = CrystalOfTableaux(['A',3], shape = [2,2])
sage: Tab(rows=[[1,2],[3,4]])
[[1, 2], [3, 4]]
sage: Tab(columns=[[3,1],[4,2]])
[[1, 2], [3, 4]]
```

FIXME: do we want to specify the columns increasingly or decreasingly That is, should this be Tab(columns = [[1,3],[2,4]])

TODO: make this fully consistent with Tableau!

TESTS:

Base cases:

```
sage: T = CrystalOfTableaux(['A',2], shape = [])
sage: T.list()
[[]]
sage: T = CrystalOfTableaux(['C',2], shape = [1])
sage: T.check()
True
sage: T.list()
[[[1]], [[2]], [[-2]], [[-1]]]
sage: T = CrystalOfTableaux(['A',2], shapes = [[],[1],[2]])
sage: T.list()
[[], [[1]], [[2]], [[3]], [[1, 1]], [[1, 2]], [[2, 2]], [[1, 3]], [[2, 3]], [[3, 3]]]
sage: T.module_generators
([], [[1]], [[1, 1]])
sage: T = CrystalOfTableaux(['B',2],shape=[3])
sage: T(rows=[[1,1,0]])
[[1, 1, 0]]
```

Input tests:

```
sage: Tab = CrystalOfTableaux(['A',3], shape = [2,2])
sage: C = Tab.letters
sage: Tab(rows    = [[1,2],[3,4]])._list == [C(3),C(1),C(4),C(2)]
True
sage: Tab(columns = [[3,1],[4,2]])._list == [C(3),C(1),C(4),C(2)]
True
```

And for compatibility with TensorProductOfCrystal we should also allow as input the internal list / sequence of elements:

```
sage: Tab(list    = [3,1,4,2])._list     == [C(3),C(1),C(4),C(2)]
True
sage: Tab(3,1,4,2)._list                 == [C(3),C(1),C(4),C(2)]
True
```

Type D, illustrating that the last parameter in the shape can be negative:

```
sage: C = CrystalOfTableaux(['D',4],shape=[1,1,1,-1])
sage: C.cardinality()
35
sage: C.check()
True
```

The next example checks whether a given tableau is in fact a valid type C tableau or not:

sage: T = CrystalOfTableaux(['C',3], shape = [2,2,2]) sage: t = T(rows=[[1,3],[2,-3],[3,-1]]) sage: t in T.list() True sage: t = T(rows=[[2,3],[3,-3],[-3,-2]]) sage: t in T.list() False

cartan_type()

Returns the Cartan type of the associated crystal

EXAMPLES:: sage: T = CrystalOfTableaux(['A',3], shape = [2,2]) sage: T.cartan_type() ['A', 3]

module_generator(*shape*)

This yields the module generator (or highest weight element) of a classical crystal of given shape. The module generator is the unique tableau with equal shape and content.

EXAMPLE: sage: T = CrystalOfTableaux(['D',3], shape = [1,1]) sage: T.module_generator([1,1]) [[1], [2]]

class CrystalOfTableauxElement(*parent, *args, **options*)

to_tableau()

Returns the Tableau object corresponding to self.

EXAMPLES:

```
sage: T = CrystalOfTableaux(['A',3], shape = [2,2])
sage: t = T(rows=[[1,2],[3,4]]).to_tableau(); t
[[1, 2], [3, 4]]
sage: type(t)
<class 'sage.combinat.tableau.Tableau_class'>
sage: type(t[0][0])
<type 'sage.rings.integer.Integer'>
sage: T = CrystalOfTableaux(['D',3], shape = [1,1])
sage: t=T(rows=[[-3],[3]]).to_tableau(); t
[[-3], [3]]
sage: t=T(rows=[[3],[-3]]).to_tableau(); t
[[3], [-3]]
```

class CrystalOfWords()

Auxiliary class to provide a call method to create tensor product elements. This class is shared with several tensor product classes and is also used in CrystalOfTableaux to allow tableaux of different tensor product structures in column-reading (and hence different shapes) to be considered elements in the same crystal.

class FullTensorProductOfClassicalCrystals(**crystals, **options*)

class FullTensorProductOfCrystals(**crystals, **options*)

cardinality()

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',2])
sage: T = TensorProductOfCrystals(C,C)
sage: T.cardinality()
9
```

list()

The default implementation of list which builds the list from the iterator.

EXAMPLES:

```
sage: class C(CombinatorialClass):
...       def __iter__(self):
...           return iter([1,2,3])
...
sage: C().list() #indirect doctest
[1, 2, 3]
```

class ImmutableListWithParent(*parent, list*)

A class for lists having a parent

Specification: any subclass C should implement __init__ which accepts the following form C(parent, list = list)

EXAMPLES: We create an immutable list whose parent is the class list:

```
sage: from sage.combinat.crystals.tensor_product import ImmutableListWithParent
sage: l = ImmutableListWithParent(list, [1,2,3])
sage: l._list
[1, 2, 3]
sage: l.parent()
<type 'list'>
sage: l.sibling([2,1]) == ImmutableListWithParent(list, [2,1])
True
sage: l.reversed()
[3, 2, 1]
sage: l.set_index(1,4)
[1, 4, 3]
```

parent()

EXAMPLES:

```
sage: from sage.combinat.crystals.tensor_product import ImmutableListWithParent
sage: l = ImmutableListWithParent(list, [1,2,3])
sage: l.parent()
<type 'list'>
```

reversed()

Returns the sibling of self which is obtained by reversing the elements of self.

EXAMPLES:

```
sage: from sage.combinat.crystals.tensor_product import ImmutableListWithParent
sage: l = ImmutableListWithParent(list, [1,2,3])
sage: l.reversed()
[3, 2, 1]
```

set_index(*k, value*)

Returns the sibling of self obtained by setting the k^{th} entry of self to value.

EXAMPLES:

```
sage: from sage.combinat.crystals.tensor_product import ImmutableListWithParent
sage: l = ImmutableListWithParent(list, [1,2,3])
sage: l.set_index(0,2)
[2, 2, 3]
sage: l.set_index(1,4)
[1, 4, 3]
sage: _.parent()
<type 'list'>
```

sibling(*l*)

Returns an ImmutableListWithParent object whose list is l and whose parent is the same as self's parent.

Note that the implementation of this function makes an assumption about the constructor for subclasses.

EXAMPLES:

```
sage: from sage.combinat.crystals.tensor_product import ImmutableListWithParent
sage: l = ImmutableListWithParent(list, [1,2,3])
sage: m = l.sibling([2,3,4]); m
[2, 3, 4]
```

```
sage: m.parent()
<type 'list'>
```

class **TensorProductOfClassicalCrystalsWithGenerators**(**crystals, **options*)

TensorProductOfCrystals(**crystals, **options*)

Tensor product of crystals.

Given two crystals B and B' of the same type, one can form the tensor product $B \otimes B'$. As a set $B \otimes B'$ is the Cartesian product $B \times B'$. The crystal operators f_i and e_i act on $b \otimes b' \in B \otimes B'$ as follows:

f_i(b otimes b') = begin{cases} f_i(b) otimes b' & text{if $varepsilon_i(b) ge varphi_i(b')$}\ b otimes f_i(b') & text{otherwise} end{cases}

and

e_i(b otimes b') = begin{cases} b otimes e_i(b') & text{if $varepsilon_i(b) le varphi_i(b')$}\ e_i(b) otimes b' & text{otherwise.} end{cases}

Note that this is the opposite of Kashiwara's convention for tensor products of crystals.

EXAMPLES:

We construct the type A_2-crystal generated by $2 \otimes 1 \otimes 1$:

```
sage: C = CrystalOfLetters(['A',2])
sage: T = TensorProductOfCrystals(C,C,C,generators=[[C(2),C(1),C(1)]])
```

It has 8 elements:

```
sage: T.list()
[[2, 1, 1], [2, 1, 2], [2, 1, 3], [3, 1, 3], [3, 2, 3], [3, 1, 1], [3, 1, 2], [3, 2, 2]]
```

One can also check the Cartan type of the crystal:

```
sage: T.cartan_type()
['A', 2]
```

Other examples include crystals of tableaux (which internally are represented as tensor products obtained by reading the tableaux columnwise):

```
sage: C = CrystalOfTableaux(['A',3], shape=[1,1,0])
sage: D = CrystalOfTableaux(['A',3], shape=[1,0,0])
sage: T = TensorProductOfCrystals(C,D, generators=[[C(rows=[[1], [2]]), D(rows=[[1]])], [C(rows=
sage: T.cardinality()
24
sage: T.check()
True
sage: T.module_generators
[[[[1], [2]], [[1]]], [[[2], [3]], [[1]]]]
sage: [x.weight() for x in T.module_generators]
[(2, 1, 0, 0), (1, 1, 1, 0)]
```

If no module generators are specified, we obtain the full tensor product:

```
sage: C=CrystalOfLetters(['A',2])
sage: T=TensorProductOfCrystals(C,C)
sage: T.list()
[[1, 1], [1, 2], [1, 3], [2, 1], [2, 2], [2, 3], [3, 1], [3, 2], [3, 3]]
sage: T.cardinality()
9
```

For a tensor product of crystals without module generators, the default implementation of module_generators contains all elements in the tensor product of the crystals. If there is a subset of elements in the tensor product that still generates the crystal, this needs to be implemented for the specific crystal separately:

```
sage: T.module_generators.list()
[[1, 1], [1, 2], [1, 3], [2, 1], [2, 2], [2, 3], [3, 1], [3, 2], [3, 3]]
```

For classical highest weight crystals, it is also possible to list all highest weight elements:

```
sage: C = CrystalOfLetters(['A',2])
sage: T = TensorProductOfCrystals(C,C,C,generators=[[C(2),C(1),C(1)],[C(1),C(2),C(1)]])
sage: T.highest_weight_vectors()
[[2, 1, 1], [1, 2, 1]]
```

class **TensorProductOfCrystalsElement** (*parent, list*)

A class for elements of tensor products of crystals

e (*i*)

Returns the action of e_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: T = TensorProductOfCrystals(C,C)
sage: T(C(1),C(2)).e(1) == T(C(1),C(1))
True
sage: T(C(2),C(1)).e(1) == None
True
sage: T(C(2),C(2)).e(1) == T(C(1),C(2))
True
```

epsilon (*i*)

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: T = TensorProductOfCrystals(C,C)
sage: T(C(1),C(1)).epsilon(1)
0
sage: T(C(1),C(2)).epsilon(1)
1
sage: T(C(2),C(1)).epsilon(1)
0
```

f (*i*)

Returns the action of f_i on self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: T = TensorProductOfCrystals(C,C)
sage: T(C(1),C(1)).f(1)
[1, 2]
sage: T(C(1),C(2)).f(1)
[2, 2]
```

```
sage: T(C(2),C(1)).f(1) is None
True
```

phi(*i*)

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: T = TensorProductOfCrystals(C,C)
sage: T(C(1),C(1)).phi(1)
2
sage: T(C(1),C(2)).phi(1)
1
sage: T(C(2),C(1)).phi(1)
0
```

positions_of_unmatched_minus(*i, dual=False, reverse=False*)

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: T = TensorProductOfCrystals(C,C)
sage: T(C(2),C(1)).positions_of_unmatched_minus(1)
[]
sage: T(C(1),C(2)).positions_of_unmatched_minus(1)
[0]
```

positions_of_unmatched_plus(*i*)

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',5])
sage: T = TensorProductOfCrystals(C,C)
sage: T(C(2),C(1)).positions_of_unmatched_plus(1)
[]
sage: T(C(1),C(2)).positions_of_unmatched_plus(1)
[1]
```

weight()

Returns the weight of self.

EXAMPLES:

```
sage: C = CrystalOfLetters(['A',3])
sage: T = TensorProductOfCrystals(C,C)
sage: T(C(1),C(2)).weight()
(1, 1, 0, 0)
```

class **TensorProductOfCrystalsWithGenerators**(**crystals, **options*)

17.35.5 Fast Rank Two Crystals

class **FastCrystal**(*ct, shape, format='string'*)

An alternative implementation of rank 2 crystals. The root operators are implemented in memory by table lookup. This means that in comparison with the Crystals of Tableaux, these crystals are slow to instantiate but faster for computation. Implemented for types A2, B2 and C2.

Input: CartanType and a shape. The CartanType is ['A',2], ['B',2] or ['C',2]. The shape is of the form [l1,l2] where l1 and l2 are either integers or (in type B) half integers such that l1-l2 is integral. It is assumed that l1 >= l2 >= 0. If l1 and l2 are integers, this will produce the a crystal isomorphic to the one obtained by CrystalOfTableaux(type, shape=[l1,l2]). Furthermore FastCrystal(['B', 2], l1+1/2, l2+1/2) produces a crystal isomorphic to the following crystal T:

```
C = CrystalOfTableaux(['B',2], shape=[l1,l2])
D = CrystalOfSpins(['B',2])
T = TensorProductOfCrystals(C,D,C.list()[0],D.list()[0])
```

The representation of elements is in term of the Berenstein-Zelevinsky-Littelmann strings [a1, a2, ...] described under metapost in crystals.py. Alternative representations may be obtained by the options format="dual_string" or format="simple". In the simple format, the element is represented by and integer, and in the dual_string format, it is represented by the Berenstein-Zelevinsky-Littelmann string, but the underlying decomposition of the long Weyl group element into simple reflections is changed.

TESTS:

```
sage: C = FastCrystal(['A',2],shape=[4,1])
sage: C.cardinality()
24
sage: C.cartan_type()
['A', 2]
sage: C.check()
True
sage: C = FastCrystal(['B',2],shape=[4,1])
sage: C.cardinality()
154
sage: C.check()
True
sage: C = FastCrystal(['B',2],shape=[3/2,1/2])
sage: C.cardinality()
16
sage: C.check()
True
sage: C = FastCrystal(['C',2],shape=[2,1])
sage: C.cardinality()
16
sage: C = FastCrystal(['C',2],shape=[3,1])
sage: C.cardinality()
35
sage: C.check()
True
```

cmp_elements(*x*, *y*)

Returns True if and only if there is a path from x to y in the crystal graph.

Because the crystal graph is classical, it is a directed acyclic graph which can be interpreted as a poset. This function implements the comparison function of this poset.

EXAMPLES:

```
sage: C = FastCrystal(['A',2],shape=[2,1])
sage: x = C(0)
sage: y = C(1)
sage: C.cmp_elements(x,y)
-1
sage: C.cmp_elements(y,x)
1
sage: C.cmp_elements(x,x)
0
```

digraph()

Returns the digraph associated to self.

EXAMPLES:

```
sage: C = FastCrystal(['A',2],shape=[2,1])
sage: C.digraph()
Digraph on 8 vertices
```

list()
Returns a list of the elements of self.
EXAMPLES:

```
sage: C = FastCrystal(['A',2],shape=[2,1])
sage: C.list()
[[0, 0, 0],
 [1, 0, 0],
 [0, 1, 1],
 [0, 2, 1],
 [1, 2, 1],
 [0, 1, 0],
 [1, 1, 0],
 [2, 1, 0]]
```

class **FastCrystalElement**(*parent, value, format*)

e(*i*)
Returns the action of e_i on self.
EXAMPLES:

```
sage: C = FastCrystal(['A',2],shape=[2,1])
sage: C(1).e(1)
[0, 0, 0]
sage: C(0).e(1) is None
True
```

f(*i*)
Returns the action of f_i on self.
EXAMPLES:

```
sage: C = FastCrystal(['A',2],shape=[2,1])
sage: C(6).f(1)
[1, 2, 1]
sage: C(7).f(1) is None
True
```

parent()
Returns the parent of self.
EXAMPLES:

```
sage: C = FastCrystal(['A',2],shape=[2,1])
sage: C[0].parent()
The fast crystal for A2 with shape [2,1]
```

weight()
Returns the weight of self.
EXAMPLES:

```
sage: [v.weight() for v in FastCrystal(['A',2], shape=[2,1])]
[(2, 1, 0), (1, 2, 0), (1, 1, 1), (1, 0, 2), (0, 1, 2), (2, 0, 1), (1, 1, 1), (0, 2, 1)]
sage: [v.weight() for v in FastCrystal(['B',2], shape=[1,0])]
[(1, 0), (0, 1), (0, 0), (0, -1), (-1, 0)]
sage: [v.weight() for v in FastCrystal(['B',2], shape=[1/2,1/2])]
```

```
[(1/2, 1/2), (1/2, -1/2), (-1/2, 1/2), (-1/2, -1/2)]
sage: [v.weight() for v in FastCrystal(['C',2], shape=[1,0])]
[(1, 0), (0, 1), (0, -1), (-1, 0)]
sage: [v.weight() for v in FastCrystal(['C',2], shape=[1,1])]
[(1, 1), (1, -1), (0, 0), (-1, 1), (-1, -1)]
```

17.36 Posets

17.36.1 Posets

class FinitePoset (*digraph, elements=None*)

Note: A class that inherits from this class needs to define _element_type. This is the class of the elements that the inheriting class contains. For example, for this class, FinitePoset, _element_type is PosetElement.

antichains()

Returns a list of all antichains of the poset.

An antichain of a poset is a collection of elements of the poset that are pairwise incomparable.

EXAMPLES:

```
sage: Posets.PentagonPoset().antichains()
[[], [0], [1], [2], [3], [4], [1, 2], [1, 3]]
sage: Posets.AntichainPoset(3).antichains()
[[], [2], [1], [0], [1, 0], [2, 1], [2, 0], [2, 1, 0]]
sage: Posets.ChainPoset(3).antichains()
[[], [0], [1], [2]]
```

bottom()

Returns the bottom element of the poset, if it exists.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.bottom() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.bottom()
0
```

closed_interval(*x, y*)

Returns a list of the elements z such that $x \leq z \leq y$. The order is that induced by the ordering in `self.linear_extension()`.

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: P = Poset(dag)
sage: I = set(map(P,[2,5,6,4,7]))
sage: I == set(P.closed_interval(2,7))
True
```

cover_relations(*element=None*)

Returns the list of pairs [u,v] of elements of the poset such that u v is a cover relation (that is, u v and there does not exist z such that u z v).

EXAMPLES:

```
sage: Q = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: Q.cover_relations()
[[1, 2], [0, 2], [2, 3], [3, 4]]
```

cover_relations_iterator()

Returns an iterator for the cover relations of the poset.

EXAMPLES:

```
sage: Q = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: type(Q.cover_relations_iterator())
<type 'generator'>
sage: [z for z in Q.cover_relations_iterator()]
[[1, 2], [0, 2], [2, 3], [3, 4]]
```

covers(*x*, *y*)

Returns True if y covers x and False otherwise.

EXAMPLES:

```
sage: Q = Poset([[1,5],[2,6],[3],[4],[],[6,3],[4]])
sage: Q.covers(Q(1),Q(6))
True
sage: Q.covers(Q(1),Q(4))
False
```

dual()

Returns the dual poset of the given poset.

EXAMPLE:

```
sage: P = Poset([[1,2],[4],[3],[4],[]])
sage: P.dual()
Finite poset containing 5 elements
```

graphviz_string(*graph_string='graph'*, *edge_string='–'*)

Returns a representation in the DOT language, ready to render in graphviz.

REFERENCES:

- http://www.graphviz.org/doc/info/lang.html

EXAMPLES:

```
sage: P = Poset({'a':['b'],'b':['d'],'c':['d'],'d':['f'],'e':['f'],'f':[]})
sage: print P.graphviz_string()
graph {
"f";"d";"b";"a";"c";"e";
"f"--"e";"d"--"c";"b"--"a";"d"--"b";"f"--"d";
}
```

has_bottom()

Returns True if the poset has a unique minimal element.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.has_bottom()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.has_bottom()
True
```

has_top()

Returns True if the poset contains a unique maximal element, and False otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.has_top()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.has_top()
True
```

hasse_diagram()

Returns the Hasse_diagram of the poset as a Sage DiGraph object.

EXAMPLES:

```
sage: Q = Poset({5:[2,3], 1:[3,4], 2:[0], 3:[0], 4:[0]})
sage: Q.hasse_diagram()
Digraph on 6 vertices

sage: P = Poset({'a':['b'],'b':['d'],'c':['d'],'d':['f'],'e':['f'],'f':[]})
sage: H = P.hasse_diagram()
sage: P.cover_relations()
[[e, f], [c, d], [a, b], [b, d], [d, f]]
sage: H.edges()
[(a, b, None), (c, d, None), (b, d, None), (e, f, None), (d, f, None)]
```

interval(*x*, *y*)

Returns a list of the elements z such that $x \leq z \leq y$. The order is that induced by the ordering in `self.linear_extension()`.

INPUT:

- `x` - any element of the poset
- `y` - any element of the poset

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: P = Poset(dag)
sage: I = set(map(P,[2,5,6,4,7]))
sage: I == set(P.interval(2,7))
True

sage: dg = DiGraph({"a":["b","c"], "b":["d"], "c":["d"]})
sage: P = Poset(dg)
sage: P.interval("a","d")
[a, c, b, d]
```

is_bounded()

Returns True if the poset contains a unique maximal element and a unique minimal element, and False otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.is_bounded()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.is_bounded()
True
```

is_chain()

Returns True if the poset is totally ordered, and False otherwise.

EXAMPLES:

```
sage: L = Poset({0:[1],1:[2],2:[3],3:[4]})
sage: L.is_chain()
True
sage: V = Poset({0:[1,2]})
sage: V.is_chain()
False
```

is_gequal(*x, y*)

Returns True if x is greater than or equal to y in the poset, and False otherwise.

EXAMPLES:

```
sage: Q = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = Q(0),Q(1),Q(4)
sage: Q.is_gequal(x,y)
False
sage: Q.is_gequal(y,x)
False
sage: Q.is_gequal(x,z)
False
sage: Q.is_gequal(z,x)
True
sage: Q.is_gequal(z,y)
True
sage: Q.is_gequal(z,z)
True
```

is_graded()

Returns True if the poset is graded, and False otherwise.

A poset is *graded* if it admits a rank function.

EXAMPLES:

```
sage: P = Poset([[1],[2],[3],[4],[]])
sage: P.is_graded()
True
sage: Q = Poset([[1,5],[2,6],[3],[4],[],[6,3],[4]])
sage: Q.is_graded()
False
```

is_greater_than(*x, y*)

Returns True if x is greater than but not equal to y in the poset, and False otherwise.

EXAMPLES:

```
sage: Q = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = Q(0),Q(1),Q(4)
sage: Q.is_greater_than(x,y)
False
sage: Q.is_greater_than(y,x)
False
sage: Q.is_greater_than(x,z)
False
sage: Q.is_greater_than(z,x)
True
sage: Q.is_greater_than(z,y)
True
sage: Q.is_greater_than(z,z)
False
```

is_join_semilattice()

Returns True is the poset has a join operation, and False otherwise.

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: P.is_join_semilattice()
True

sage: P = Poset([[1,2],[3],[3],[]])
sage: P.is_join_semilattice()
True

sage: P = Poset({0:[2,3],1:[2,3]})
sage: P.is_join_semilattice()
False
```

is_lequal(*x*, *y*)

Returns True if x is less than or equal to y in the poset, and False otherwise.

EXAMPLES:

```
sage: Q = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = Q(0),Q(1),Q(4)
sage: Q.is_lequal(x,y)
False
sage: Q.is_lequal(y,x)
False
sage: Q.is_lequal(x,z)
True
sage: Q.is_lequal(y,z)
True
sage: Q.is_lequal(z,z)
True
```

is_less_than(*x*, *y*)

Returns True if x is less than but not equal to y in the poset, and False otherwise.

EXAMPLES:

```
sage: Q = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = Q(0),Q(1),Q(4)
sage: Q.is_less_than(x,y)
False
sage: Q.is_less_than(y,x)
False
sage: Q.is_less_than(x,z)
True
sage: Q.is_less_than(y,z)
True
sage: Q.is_less_than(z,z)
False
```

is_meet_semilattice()

Returns True if self has a meet operation, and False otherwise.

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: P.is_meet_semilattice()
True

sage: P = Poset([[1,2],[3],[3],[]])
sage: P.is_meet_semilattice()
True
```

```
sage: P = Poset({0:[2,3],1:[2,3]})
sage: P.is_meet_semilattice()
False
```

is_ranked()

Returns True if the poset is ranked, and False otherwise.

A poset is {ranked} if it admits a rank function.

EXAMPLES:

```
sage: P = Poset([[1],[2],[3],[4],[]])
sage: P.is_ranked()
True
sage: Q = Poset([[1,5],[2,6],[3],[4],[],[6,3],[4]])
sage: Q.is_ranked()
False
```

join_matrix()

Returns a matrix whose (i,j) entry is k, where self.linear_extension()[k] is the join (least upper bound) of self.linear_extension()[i] and self.linear_extension()[j].

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: J = P.join_matrix(); J
[0 1 2 3 4 5 6 7]
[1 1 3 3 7 7 7 7]
[2 3 2 3 4 6 6 7]
[3 3 3 3 7 7 7 7]
[4 7 4 7 4 7 7 7]
[5 7 6 7 7 5 6 7]
[6 7 6 7 7 6 6 7]
[7 7 7 7 7 7 7 7]
sage: J[P(4).vertex,P(3).vertex] == P(7).vertex
True
sage: J[P(5).vertex,P(2).vertex] == P(5).vertex
True
sage: J[P(5).vertex,P(2).vertex] == P(2).vertex
False
```

lequal_matrix(***kwds*)

Computes the matrix whose [i,j] entry is 1 if self.linear_extension()[i] self.linear_extension()[j] 0 otherwise.

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: LEQM = P.lequal_matrix(); LEQM
[1 1 1 1 1 1 1 1]
[0 1 0 1 0 0 0 1]
[0 0 1 1 1 0 1 1]
[0 0 0 1 0 0 0 1]
[0 0 0 0 1 0 0 1]
[0 0 0 0 0 1 1 1]
[0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 1]
sage: LEQM[1,3]
1
sage: P.linear_extension()[1] < P.linear_extension()[3]
True
sage: LEQM[2,5]
```

```
0
sage: P.linear_extension()[2] < P.linear_extension()[5]
False
```

level_sets()

Returns a list l such that l[i+1] is the set of minimal elements of the poset obtained by removing the elements in l[0], l[1], ..., l[i].

EXAMPLES:

```
sage: P = Poset({0:[1,2],1:[3],2:[3],3:[]})
sage: [len(x) for x in P.level_sets()]
[1, 2, 1]

sage: Q = Poset({0:[1,2], 1:[3], 2:[4], 3:[4]})
sage: [len(x) for x in Q.level_sets()]
[1, 2, 1, 1]
```

linear_extension()

Returns a linear extension of the poset.

EXAMPLES:

```
sage: B = Posets.BooleanLattice(3)
sage: B.linear_extension()
[0, 1, 2, 3, 4, 5, 6, 7]
```

linear_extensions()

Returns a list of all the linear extensions of the poset.

EXAMPLES:

```
sage: D = Poset({ 0:[1,2], 1:[3], 2:[3,4] })
sage: D.linear_extensions()
[[0, 1, 2, 3, 4], [0, 1, 2, 4, 3], [0, 2, 1, 3, 4], [0, 2, 1, 4, 3], [0, 2, 4, 1, 3]]
```

list()

List the elements of the poset. This just returns the result of `linear_extension()`.

EXAMPLES:

```
sage: D = Poset({ 0:[1,2], 1:[3], 2:[3,4] })
sage: D.list()
[0, 1, 2, 3, 4]
sage: type(D.list()[0])
<class 'sage.combinat.posets.elements.PosetElement'>
```

lower_covers(*y*)

Returns a list of lower covers of the element y. An lower cover of y is an element x such that y x is a cover relation.

EXAMPLES:

```
sage: Q = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: map(Q.lower_covers,Q.list())
[[], [], [1, 0], [2], [3]]
```

lower_covers_iterator(*y*)

Returns an iterator for the lower covers of the element y. An lower cover of y is an element x such that y x is a cover relation.

EXAMPLES:

```
sage: Q = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: type(Q.lower_covers_iterator(0))
<type 'generator'>
```

maximal_elements()

Returns a list of the maximal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.maximal_elements()
[4]
```

meet_matrix()

Returns a matrix whose (i,j) entry is k, where self.linear_extension()[k] is the meet (greatest lower bound) of self.linear_extension()[i] and self.linear_extension()[j].

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: M = P.meet_matrix(); M
[0 0 0 0 0 0 0 0]
[0 1 0 1 0 0 0 1]
[0 0 2 2 2 0 2 2]
[0 1 2 3 2 0 2 3]
[0 0 2 2 4 0 2 4]
[0 0 0 0 0 5 5 5]
[0 0 2 2 2 5 6 6]
[0 1 2 3 4 5 6 7]
sage: M[P(4).vertex,P(3).vertex] == P(0).vertex
True
sage: M[P(5).vertex,P(2).vertex] == P(2).vertex
True
sage: M[P(5).vertex,P(2).vertex] == P(5).vertex
False
```

minimal_elements()

Returns a list of the minimal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P(0) in P.minimal_elements()
True
sage: P(1) in P.minimal_elements()
True
sage: P(2) in P.minimal_elements()
True
```

mobius_function(*x*, *y*)

Returns the value of the Mobius function of the poset on the elements x and y.

EXAMPLES:

```
sage: P = Poset([[1,2,3],[4],[4],[4],[]])
sage: P.mobius_function(P(0),P(4))
2
sage: sum([P.mobius_function(P(0),v) for v in P])
0
sage: sum([abs(P.mobius_function(P(0),v)) \
...        for v in P])
6
sage: for u,v in P.cover_relations_iterator():
```

```
...     if P.mobius_function(u,v) != -1:
...         print "Bug in mobius_function!"

sage: Q = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: Q.mobius_function(Q(0),Q(-1))
0
sage: Q.mobius_function(Q(0),Q(5))
0
sage: Q.mobius_function(Q(2),Q(7))
2
sage: Q.mobius_function(Q(3),Q(3))
1
sage: sum([Q.mobius_function(Q(0),v) for v in Q])
0
```

mobius_function_matrix()

Returns a matrix whose (i,j) entry is the value of the Mobius function evaluated at self.linear_extension()[i] and self.linear_extension()[j].

EXAMPLES:

```
sage: P = Poset([[4,2,3],[],[1],[1],[1]])
sage: x,y = (P.linear_extension()[0],P.linear_extension()[1])
sage: P.mobius_function(x,y)
-1
sage: M = P.mobius_function_matrix(); M
[ 1 -1 -1 -1  2]
[ 0  1  0  0 -1]
[ 0  0  1  0 -1]
[ 0  0  0  1 -1]
[ 0  0  0  0  1]
sage: M[0,4]
2
sage: M[0,1]
-1
```

open_interval(x, y)

Returns a list of the elements z such that $x < z < y$. The order is that induced by the ordering in `self.linear_extension()`.

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: P = Poset(dag)
sage: I = set(map(P,[5,6,4]))
sage: I == set(P.open_interval(2,7))
True

sage: dg = DiGraph({"a":["b","c"], "b":["d"], "c":["d"]})
sage: P = Poset(dg)
sage: P.open_interval("a","d")
[c, b]
```

order_complex()

Returns the order complex associated to this poset.

The order complex is the simplical complex with vertices equal to the elements of the poset, and faces given by the chains.

EXAMPLES:

```
sage: P = Posets.BooleanLattice(3)
sage: S = P.order_complex(); S
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5, 6, 7) and 6 facets
sage: S.f_vector()
[1, 8, 19, 18, 6]
sage: S.homology()      # S is contractible
{0: 0, 1: 0, 2: 0, 3: 0}
sage: Q = P.subposet([1,2,3,4,5,6])
sage: Q.order_complex().homology()    # a circle
{0: 0, 1: Z}
```

order_filter(*elements*)

Returns the order filter generated by a list of elements.

I is an order filter if, for any x in I and y such that $y \geq x$, then y is in I.

EXAMPLES:

```
sage: B = Posets.BooleanLattice(4)
sage: B.order_filter([3,8])
[8, 9, 10, 3, 11, 12, 13, 14, 7, 15]
```

order_ideal(*elements*)

Returns the order ideal generated by a list of elements.

I is an order ideal if, for any x in I and y such that $y \leq x$, then y is in I.

EXAMPLES:

```
sage: B = Posets.BooleanLattice(4)
sage: B.order_ideal([7,10])
[0, 8, 1, 2, 10, 3, 4, 5, 6, 7]
```

plot(*label_elements=True, element_labels=None, label_font_size=12, label_font_color='black', vertex_size=300, vertex_colors=None, **kwds*)

Returns a Graphic object corresponding the Hasse diagram of the poset. Optionally, it is labelled.

INPUT:

- `label_elements` - whether to display element labels
- `element_labels` - a dictionary of element labels

EXAMPLES:

```
sage: D = Poset({ 0:[1,2], 1:[3], 2:[3,4] })
sage: D.plot(label_elements=False)
sage: D.plot()
sage: type(D.plot())
<class 'sage.plot.plot.Graphics'>
sage: elm_labs = {0:'a', 1:'b', 2:'c', 3:'d', 4:'e'}
sage: D.plot(element_labels=elm_labs)

sage: P = Poset({})
sage: P.plot()

sage: P = Poset(DiGraph('E@ACA@?'))
sage: P.plot()
```

principal_order_filter(*x*)

Returns the order filter generated by an element `x`.

EXAMPLES:

```
sage: B = Posets.BooleanLattice(4)
sage: B.principal_order_filter(2)
[2, 10, 3, 11, 6, 14, 7, 15]
```

principal_order_ideal(*x*)
Returns the order ideal generated by an element x.
EXAMPLES:

```
sage: B = Posets.BooleanLattice(4)
sage: B.principal_order_ideal(6)
[0, 2, 4, 6]
```

random_subposet(*p*)
Returns a random subposet that contains each element with probability p.
EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: Q = P.random_subposet(.25)
```

rank(*element=None*)
Returns the rank of an element, or the rank of the poset if element is None. (The rank of a poset is the length of the longest chain of elements of the poset.)
EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: P.rank(5)
2
sage: P.rank()
3
sage: Q = Poset([[1,2],[3],[],[]])

sage: P = Posets.SymmetricGroupBruhatOrderPoset(4)

sage: [(v,P.rank(v)) for v in P]
[(1234, 0),
 (2134, 1),
...
 (4231, 5),
 (4321, 6)]
```

rank_function()
Returns a rank function of the poset, if it exists.
A *rank function* of a poset P is a function r from that maps elements of P to integers and satisfies: $r(x) = r(y) + 1$ if x covers y.
EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: P.rank_function() is not None
True
sage: r = P.rank_function()
sage: for u,v in P.cover_relations_iterator():
...     if r(v) != r(u) + 1:
...         print "Bug in rank_function!"

sage: Q = Poset([[1,2],[4],[3],[4],[]])
sage: Q.rank_function() is None
True
```

show(*label_elements=True, element_labels=None, label_font_size=12, label_font_color='black', vertex_size=300, vertex_colors=None, **kwds*)
Shows the Graphics object corresponding the Hasse diagram of the poset. Optionally, it is labelled.
INPUT:

- label_elements - whether to display element labels

- `element_labels` - a dictionary of element labels

EXAMPLES:

```
sage: D = Poset({ 0:[1,2], 1:[3], 2:[3,4] })
sage: D.plot(label_elements=False)
sage: D.show()
sage: elm_labs = {0:'a', 1:'b', 2:'c', 3:'d', 4:'e'}
sage: D.show(element_labels=elm_labs)
```

size()

Returns the number of elements in the poset.

EXAMPLES:

```
sage: Poset([[1,2,3],[4],[4],[4],[]]).size()
5
```

subposet(*elements*)

Returns the poset containing elements with partial order induced by that of self.

EXAMPLES:

```
sage: P = Poset({"a":["c","d"], "b":["d","e"], "c":["f"], "d":["f"], "e":["f"]})
sage: P.subposet(["a","b","f"])
Finite poset containing 3 elements

sage: P = posets.BooleanLattice(2)
sage: above = P.principal_order_filter(0)
sage: Q = P.subposet(above)
sage: above_new = Q.principal_order_filter(Q.list()[0])
sage: Q.subposet(above_new)
Finite poset containing 4 elements
```

top()

Returns the top element of the poset, if it exists.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.top() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.top()
1
```

upper_covers(*y*)

Returns a list of upper covers of the element y. An upper cover of y is an element x such that y x is a cover relation.

EXAMPLES:

```
sage: Q = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: map(Q.upper_covers,Q.list())
[[2], [2], [3], [4], []]
```

upper_covers_iterator(*y*)

Returns an iterator for the upper covers of the element y. An upper cover of y is an element x such that y x is a cover relation.

EXAMPLES:

```
sage: Q = Poset({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: type(Q.upper_covers_iterator(0))
<type 'generator'>
```

class FinitePosets_n(*n*)

The Combinatorial Class of all posets on n vertices.

cardinality(*from_iterator=False*)

Return the cardinality of this object.

Note: By default, this returns pre-computed values obtained from the On-Line Encyclopedia of Integer Sequences (A000112). To override this, use pass the argument `from_iterator=True`.

EXAMPLES:

```
sage: P = Posets(3)
sage: P.cardinality()
5
sage: P.cardinality(from_iterator=True)
5
```

Poset(*data=None, element_labels=None, cover_relations=False*)

Construct a poset from various forms of input data.

INPUT:

1. A two-element list or tuple (E, R), where E is a collection of elements of the poset and R is the set of relations. Elements of R are two-element lists/tuples/iterables. If cover_relations=True, then R is assumed to be the cover relations of the poset. If E is empty, then E is taken to be the set of elements appearing in the relations R.
2. A two-element list or tuple (E, f), where E is the set of elements of the poset and f is a function such that f(x,y) is True if x <= y and False otherwise for all pairs of elements in E. If cover_relations=True, then f(x,y) should be True if and only if x is covered by y, and False otherwise.
3. A dictionary, list or tuple of upper covers: data[x] is an list of the elements that cover the element x in the poset.

   Note: If data is a list or tuple of length 2, then it is handled by the above cases.
4. An acyclic, loop-free and multi-edge free DiGraph. If cover_relations is True, then the edges of the digraph correspond to cover relations in the poset. If cover_relations is False, then the cover relations are computed.
5. A previously constructed poset (the poset itself is returned).

- **element_labels – (default: None) an optional list or** dictionary of objects that label the poset elements.
- `cover_relations` - (default: False) If True, then the data is assumed to describe a directed acyclic graph whose arrows are cover relations. If False, then the cover relations are first computed.

OUTPUT:

FinitePoset – an instance of the FinitePoset class.

EXAMPLES:

1. Elements and cover relations:

```
sage: elms = [1,2,3,4,5,6,7]
sage: rels = [[1,2],[3,4],[4,5],[2,5]]
sage: Poset((elms, rels), cover_relations = True)
Finite poset containing 7 elements
```

Elements and non-cover relations:

```
sage: elms = [1,2,3,4]
sage: rels = [[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]]
sage: P = Poset( [elms,rels] ,cover_relations=False); P
```

```
Finite poset containing 4 elements
sage: P.cover_relations()
[[1, 2], [2, 3], [3, 4]]
```

2.Elements and function: the standard permutations of [1, 2, 3, 4] with the Bruhat order:

```
sage: elms = Permutations(4)
sage: fcn = lambda p,q : p.bruhat_lequal(q)
sage: Poset((elms, fcn))
Finite poset containing 24 elements
```

With a function that identifies the cover relations: the set partitions of {1, 2, 3} ordered by refinement:

```
sage: elms = SetPartitions(3)
sage: def fcn(A, B):
...     if len(A) != len(B)+1:
...         return False
...     for a in A:
...         if not any(set(a).issubset(b) for b in B):
...             return False
...     return True
sage: Poset((elms, fcn), cover_relations=True)
Finite poset containing 5 elements
```

3.A dictionary of upper covers:

```
sage: Poset({'a':['b','c'], 'b':['d'], 'c':['d'], 'd':[]})
Finite poset containing 4 elements
```

A list of upper covers:

```
sage: Poset([[1,2],[4],[3],[4],[]])
Finite poset containing 5 elements
```

A list of upper covers and a dictionary of labels:

```
sage: elm_labs = {0:"a",1:"b",2:"c",3:"d",4:"e"}
sage: P = Poset([[1,2],[4],[3],[4],[]],elm_labs)
sage: P.list()
[a, b, c, d, e]
```

Warning: The special case where the argument data is a list or tuple of length 2 is handled by the above cases. So you cannot use this method to input a 2-element poset.

4.An acyclic DiGraph.

```
sage: dag = DiGraph({0:[2,3], 1:[3,4], 2:[5], 3:[5], 4:[5]})
sage: Poset(dag)
Finite poset containing 6 elements
```

Any directed acyclic graph without loops or multiple edges, as long as cover_relations=False:

```
sage: dig = DiGraph({0:[2,3], 1:[3,4,5], 2:[5], 3:[5], 4:[5]})
sage: dig.allows_multiple_edges()
False
sage: dig.allows_loops()
False
sage: dig.transitive_reduction() == dig
False
sage: Poset(dig, cover_relations=False)
Finite poset containing 6 elements
sage: Poset(dig, cover_relations=True)
```

```
...
ValueError: Hasse diagram is not transitively reduced.
```

class **Posets_all**()
The Combinatorial Class of all posets.

is_poset(*dig*)
Tests whether a directed graph is acyclic and transitively reduced.

EXAMPLES:

```
sage: from sage.combinat.posets.posets import is_poset
sage: dig = DiGraph({0:[2,3], 1:[3,4,5], 2:[5], 3:[5], 4:[5]})
sage: is_poset(dig)
False
sage: is_poset(dig.transitive_reduction())
True
```

17.36.2 Hasse diagrams of posets

class **HasseDiagram**(*data=None, pos=None, loops=None, format=None, boundary=, [], weighted=None, implementation='networkx', sparse=True, vertex_labels=True, **kwds*)
The Hasse diagram of a poset. This is just a transitively-reduced, directed, acyclic graph without loops or multiple edges.

Note: We assume that `range(n)` is a linear extension of the poset. That is, `range(n)` is the vertex set and a topological sort of the digraph.

This should not be called directly, use Poset instead; all type checking happens there.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]}); H
Hasse diagram of a poset containing 4 elements
sage: H == loads(dumps(H))
True
```

antichains()
Returns a list of all antichains of the poset.

An antichain of a poset is a collection of elements of the poset that are pairwise incomparable.

Note: This algorithm is based on Freese-Jezek-Nation p226

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[4],2:[3],3:[4]})
sage: H.antichains()
[[], [0], [1], [2], [3], [4], [1, 2], [1, 3]]

sage: H = HasseDiagram({0:[],1:[],2:[]})
sage: H.antichains()
[[], [0], [1], [2], [1, 2], [0, 1], [0, 2], [0, 1, 2]]

sage: H = HasseDiagram({0:[1],1:[2],2:[3],3:[4]})
sage: H.antichains()
[[], [0], [1], [2], [3], [4]]
```

bottom()

Returns the bottom element of the poset, if it exists.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.bottom() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.bottom()
0
```

closed_interval(*x, y*)

Returns a list of the elements z such that x = z = y. The order is that induced by the ordering in self.linear_extension.

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram(dag)
sage: set([2,5,6,4,7]) == set(H.closed_interval(2,7))
True
```

complements()

Returns a list `l` such that `l[i]` is a complement of `i` in `self`.

A complement of `x` is an element `y` such that the meet of `x` and `y` is the bottom element of `self` and the join of `x` and `y` is the top element of `self`.

EXAMPLES:: sage: from sage.combinat.posets.hasse_diagram import HasseDiagram sage: H = HasseDiagram({0:[1,2,3],1:[4],2:[4],3:[4]}) sage: H.complements() [4, 3, 3, 2, 0]
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[4]}) sage: H.complements() [4, None, None, None, 0]

cover_relations(*element=None*)

TESTS:: sage: from sage.combinat.posets.hasse_diagram import HasseDiagram sage: H = HasseDiagram({0:[2,3], 1:[3,4], 2:[5], 3:[5], 4:[5]}) sage: H.cover_relations() [(0, 2), (0, 3), (1, 3), (1, 4), (2, 5), (3, 5), (4, 5)]

cover_relations_iterator()

TESTS:: sage: from sage.combinat.posets.hasse_diagram import HasseDiagram sage: H = HasseDiagram({0:[2,3], 1:[3,4], 2:[5], 3:[5], 4:[5]}) sage: list(H.cover_relations_iterator()) [(0, 2), (0, 3), (1, 3), (1, 4), (2, 5), (3, 5), (4, 5)]

covers(*x, y*)

Returns True if y covers x and False otherwise.

EXAMPLES:

```
sage: Q = Poset([[1,5],[2,6],[3],[4],[],[6,3],[4]])
sage: Q.covers(Q(1),Q(6))
True
sage: Q.covers(Q(1),Q(4))
False
```

dual()

Returns a poset that is dual to the given poset.

EXAMPLE:

```
sage: P = Poset([[1,2],[4],[3],[4],[]])
sage: P.dual()
Finite poset containing 5 elements
```

has_bottom()

Returns True if the poset has a unique minimal element.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.has_bottom()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.has_bottom()
True
```

has_top()

Returns `True` if the poset contains a unique maximal element, and `False` otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.has_top()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.has_top()
True
```

interval(*x*, *y*)

Returns a list of the elements z such that x <= z <= y. The order is that induced by the ordering in self.linear_extension.

INPUT:

- `x` - any element of the poset
- `y` - any element of the poset

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram(dag)
sage: I = set([2,5,6,4,7])
sage: I == set(H.interval(2,7))
True
```

is_bounded()

Returns True if the poset contains a unique maximal element and a unique minimal element, and False otherwise.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.is_bounded()
False
sage: Q = Poset({0:[1],1:[]})
sage: Q.is_bounded()
True
```

is_chain()

Returns True if the poset is totally ordered, and False otherwise.

EXAMPLES:

```
sage: L = Poset({0:[1],1:[2],2:[3],3:[4]})
sage: L.is_chain()
True
sage: V = Poset({0:[1,2]})
```

```
sage: V.is_chain()
False
```

is_complemented_lattice()

Returns `True` if `self` is the Hasse diagram of a complemented lattice, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: H.is_complemented_lattice()
True

sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[4]})
sage: H.is_complemented_lattice()
False
```

is_distributive_lattice()

Returns `True` if `self` is the Hasse diagram of a distributive lattice, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.is_distributive_lattice()
False
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3]})
sage: H.is_distributive_lattice()
True
sage: H = HasseDiagram({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: H.is_distributive_lattice()
False
```

is_distributive_lattice_fastest()

This function is deprecated and will be removed in a future version of Sage. Please use `self.is_distributive_lattice()` instead.

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.is_distributive_lattice_fastest()
doctest:1: DeprecationWarning: is_distributive_lattice_fastest is deprecated, use is_distrib
False
```

is_gequal(*x*, *y*)

Returns `True` if `x` is greater than or equal to `y`, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: Q = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = 0,1,4
sage: Q.is_gequal(x,y)
False
sage: Q.is_gequal(y,x)
False
sage: Q.is_gequal(x,z)
False
sage: Q.is_gequal(z,x)
True
sage: Q.is_gequal(z,y)
True
```

```
sage: Q.is_gequal(z,z)
True
```

is_graded()

Returns True if the poset is graded, and False otherwise.

A poset is {graded} if it admits a rank function.

EXAMPLES:

```
sage: P = Poset([[1],[2],[3],[4],[]])
sage: P.is_graded()
True
sage: Q = Poset([[1,5],[2,6],[3],[4],[],[6,3],[4]])
sage: Q.is_graded()
False
```

is_greater_than(*x*, *y*)

Returns `True` if `x` is greater than but not equal to `y`, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: Q = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = 0,1,4
sage: Q.is_greater_than(x,y)
False
sage: Q.is_greater_than(y,x)
False
sage: Q.is_greater_than(x,z)
False
sage: Q.is_greater_than(z,x)
True
sage: Q.is_greater_than(z,y)
True
sage: Q.is_greater_than(z,z)
False
```

is_join_semilattice()

Returns `True` if `self` has a join operation, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.is_join_semilattice()
True
sage: H = HasseDiagram({0:[2,3],1:[2,3]})
sage: H.is_join_semilattice()
False
sage: H = HasseDiagram({0:[2,3],1:[2,3],2:[4],3:[4]})
sage: H.is_join_semilattice()
False
```

is_lequal(*i*, *j*)

Returns True if i is less than or equal to j in the poset, and False otherwise.

Note: If the `leq_matrix` has been computed, then this method is redefined to use the cached matrix (see `_alternate_is_lequal`).

TESTS:: sage: from sage.combinat.posets.hasse_diagram import HasseDiagram sage: H = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]}) sage: x,y,z = 0, 1, 4 sage: H.is_lequal(x,y) False sage: H.is_lequal(y,x) False sage: H.is_lequal(x,z) True sage: H.is_lequal(y,z) True sage: H.is_lequal(z,z) True

is_less_than(*x*, *y*)

Returns True if `x` is less than or equal to `y` in the poset, and False otherwise.

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2], 1:[2], 2:[3], 3:[4], 4:[]})
sage: x,y,z = 0, 1, 4
sage: H.is_less_than(x,y)
False
sage: H.is_less_than(y,x)
False
sage: H.is_less_than(x,z)
True
sage: H.is_less_than(y,z)
True
sage: H.is_less_than(z,z)
False
```

is_linear_extension(*lin_ext=None*)

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: H.is_linear_extension(range(4))
True
sage: H.is_linear_extension([3,2,1,0])
False
```

is_meet_semilattice()

Returns `True` if `self` has a meet operation, and `False` otherwise.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.is_meet_semilattice()
True

sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: H.is_meet_semilattice()
True

sage: H = HasseDiagram({0:[2,3],1:[2,3]})
sage: H.is_meet_semilattice()
False
```

is_ranked()

Returns True if the poset is ranked, and False otherwise.

A poset is *ranked* if it admits a rank function.

EXAMPLES:

```
sage: P = Poset([[1],[2],[3],[4],[]])
sage: P.is_ranked()
True
sage: Q = Poset([[1,5],[2,6],[3],[4],[],[6,3],[4]])
sage: Q.is_ranked()
False
```

join_matrix()

Returns the matrix of joins of `self`. The `(x,y)`-entry of this matrix is the join of `x` and `y` in `self`.

This algorithm is modelled after the algorithm of Freese-Jezek-Nation (p217).

Note: Once the matrix has been computed, it is stored in `self._join_matrix`. Delete this attribute if you want to recompute the matrix.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.join_matrix()
[0 1 2 3 4 5 6 7]
[1 1 4 7 4 7 7 7]
[2 4 2 6 4 5 6 7]
[3 7 6 3 7 7 6 7]
[4 4 4 7 4 7 7 7]
[5 7 5 7 7 5 7 7]
[6 7 6 6 7 7 6 7]
[7 7 7 7 7 7 7 7]
```

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2,3],1:[2,3]})
sage: H.join_matrix()
...
ValueError: Not a join-semilattice: no top element.

sage: H = HasseDiagram({0:[2,3],1:[2,3],2:[4],3:[4]})
sage: H.join_matrix()
...
ValueError: No join for x=...
```

lequal_matrix(*ring=Integer Ring, sparse=True*)

Computes a matrix whose `[i,j]` entry is 1 if `self.linear_extension[i] <= self.linear_extension[j]` and 0 otherwise. The matrix is stored in the attribute `_leq_matrix` and the method `__lt__` is redefined to use this matrix.

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: H = P._hasse_diagram
sage: hasattr(H,'_leq_matrix')
False
sage: H.lequal_matrix()
[1 1 1 1 1 1 1 1]
[0 1 0 1 0 0 0 1]
[0 0 1 1 1 0 1 1]
[0 0 0 1 0 0 0 1]
[0 0 0 0 1 0 0 1]
[0 0 0 0 0 1 1 1]
[0 0 0 0 0 0 1 1]
[0 0 0 0 0 0 0 1]
sage: hasattr(H,'_leq_matrix')
True
```

level_sets()

Returns a list l such that l[i+1] is the set of minimal elements of the poset obtained by removing the elements in l[0], l[1], ..., l[i].

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: [len(x) for x in H.level_sets()]
```

```
[1, 2, 1]

sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2], 1:[3], 2:[4], 3:[4]})
sage: [len(x) for x in H.level_sets()]
[1, 2, 1, 1]
```

linear_extension()

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: H.linear_extension()
[0, 1, 2, 3]
```

linear_extensions()

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,2],1:[3],2:[3],3:[]})
sage: H.linear_extensions()
[[0, 1, 2, 3], [0, 2, 1, 3]]
```

lower_covers_iterator(*element*)

Returns the list of elements that are covered by `element`.

EXAMPLES:: sage: from sage.combinat.posets.hasse_diagram import HasseDiagram sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]}) sage: list(H.lower_covers_iterator(0)) [] sage: list(H.lower_covers_iterator(4)) [1, 2]

maximal_elements()

Returns a list of the maximal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P.maximal_elements()
[4]
```

meet_matrix()

Returns the matrix of meets of self. The `(x,y)`-entry of this matrix is the meet of `x` and `y` in `self`.

This algorithm is modelled after the algorithm of Freese-Jezek-Nation (p217).

Note: Once the matrix has been computed, it is stored in self._meet_matrix. Delete this attribute if you want to recompute the matrix.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.meet_matrix()
[0 0 0 0 0 0 0 0]
[0 1 0 0 1 0 0 1]
[0 0 2 0 2 2 2 2]
[0 0 0 3 0 0 3 3]
[0 1 2 0 4 2 2 4]
[0 0 2 0 2 5 2 5]
[0 0 2 3 2 2 6 6]
[0 1 2 3 4 5 6 7]
```

TESTS:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[2,3],1:[2,3]})
sage: H.meet_matrix()
...
ValueError: Not a meet-semilattice: no bottom element.

sage: H = HasseDiagram({0:[1,2],1:[3,4],2:[3,4]})
sage: H.meet_matrix()
...
ValueError: No meet for x=...
```

minimal_elements()

Returns a list of the minimal elements of the poset.

EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4],4:[]})
sage: P(0) in P.minimal_elements()
True
sage: P(1) in P.minimal_elements()
True
sage: P(2) in P.minimal_elements()
True
```

mobius_function(*i*, *j*)

Returns the value of the M"obius function of the poset on the elements `i` and `j`.

EXAMPLES:

```
sage: P = Poset([[1,2,3],[4],[4],[4],[]])
sage: H = P._hasse_diagram
sage: H.mobius_function(0,4)
2
sage: for u,v in P.cover_relations_iterator():
...     if P.mobius_function(u,v) != -1:
...         print "Bug in mobius_function!"
```

mobius_function_matrix()

Returns a matrix whose `(x, y)` entry is the value of the M"obius function of self evaluated on `x` and `y`.

Note: Once this matrix has been computed, it is stored in `self._mobius_function_matrix`. Delete this attribute if you want to recompute the matrix.

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.mobius_function_matrix()
[ 1 -1 -1 -1  1  0  1  0]
[ 0  1  0  0 -1  0  0  0]
[ 0  0  1  0 -1 -1 -1  2]
[ 0  0  0  1  0  0 -1  0]
[ 0  0  0  0  1  0  0 -1]
[ 0  0  0  0  0  1  0 -1]
[ 0  0  0  0  0  0  1 -1]
[ 0  0  0  0  0  0  0  1]
sage: hasattr(H,'_mobius_function_matrix')
True
```

open_interval(*x*, *y*)

Returns a list of the elements z such that $x < z < y$. The order is that induced by the ordering in `self.linear_extension`.

EXAMPLES:

```
sage: uc = [[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]]
sage: dag = DiGraph(dict(zip(range(len(uc)),uc)))
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram(dag)
sage: set([5,6,4]) == set(H.open_interval(2,7))
True
sage: H.open_interval(7,2)
[]
```

order_filter(*elements*)

Returns the order filter generated by a list of elements.

I is an order filter if, for any x in I and y such that $y \geq x$, then y is in I.

EXAMPLES:

```
sage: H = Posets.BooleanLattice(4)._hasse_diagram
sage: H.order_filter([3,8])
[3, 7, 8, 9, 10, 11, 12, 13, 14, 15]
```

order_ideal(*elements*)

Returns the order ideal generated by a list of elements.

I is an order ideal if, for any x in I and y such that $y \leq x$, then y is in I.

EXAMPLES:

```
sage: H = Posets.BooleanLattice(4)._hasse_diagram
sage: H.order_ideal([7,10])
[0, 1, 2, 3, 4, 5, 6, 7, 8, 10]
```

plot(*label_elements=True, element_labels=None, label_font_size=12, label_font_color='black', **kwds*)

Returns a Graphics object corresponding to the Hasse diagram.

EXAMPLES:

```
sage: uc = [[2,3], [], [1], [1], [1], [3,4]]
sage: elm_lbls = Permutations(3).list()
sage: P = Poset(uc,elm_lbls)
sage: H = P._hasse_diagram
sage: levels = H.level_sets()
sage: heights = dict([[i, levels[i]] for i in range(len(levels))])
sage: type(H.plot(label_elements=True))
<class 'sage.plot.plot.Graphics'>

sage: P = Posets.SymmetricGroupBruhatIntervalPoset([0,1,2,3], [2,3,0,1])
sage: P._hasse_diagram.plot()
```

principal_order_filter(*i*)

Returns the order filter generated by `i`.

EXAMPLES:

```
sage: H = Posets.BooleanLattice(4)._hasse_diagram
sage: H.principal_order_filter(2)
[2, 3, 6, 7, 10, 11, 14, 15]
```

principal_order_ideal(*i*)

Returns the order ideal generated by i.

EXAMPLES:

```
sage: H = Posets.BooleanLattice(4)._hasse_diagram
sage: H.principal_order_ideal(6)
[0, 2, 4, 6]
```

rank(*element=None*)

Returns the rank of `element`, or the rank of the poset if `element` is `None`. (The rank of a poset is the length of the longest chain of elements of the poset.)

EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: H.rank(5)
2
sage: H.rank()
3
sage: Q = HasseDiagram({0:[1,2],1:[3],2:[],3:[]})
sage: Q.rank()
2
sage: Q.rank(1)
1
```

rank_function()

Returns a rank function of the poset, if it exists.

A *rank function* of a poset P is a function r from that maps elements of P to integers and satisfies: $r(x) = r(y) + 1$ if x covers y.

EXAMPLES:

```
sage: P = Poset([[1,3,2],[4],[4,5,6],[6],[7],[7],[7],[]])
sage: P.rank_function() is not None
True
sage: r = P.rank_function()
sage: for u,v in P.cover_relations_iterator():
...     if r(v) != r(u) + 1:
...         print "Bug in rank_function!"

sage: Q = Poset([[1,2],[4],[3],[4],[]])
sage: Q.rank_function() is None
True
```

show(*label_elements=True, element_labels=None, label_font_size=12, label_font_color='black', vertex_size=300, vertex_colors=None, **kwds*)

Shows the Graphics object corresponding to the Hasse diagram. Optionally, it is labelled.

INPUT:

- `label_elements` - whether to display element labels
- `element_labels` - a dictionary of element labels

EXAMPLES:

```
sage: uc = [[2,3], [], [1], [1], [1], [3,4]]
sage: elm_lbls = Permutations(3).list()
sage: P = Poset(uc,elm_lbls)
sage: H = P._hasse_diagram
sage: levels = H.level_sets()
sage: heights = dict([[i, levels[i]] for i in range(len(levels))])
sage: H.show(label_elements=True)
```

size()

Returns the number of elements in the poset.

EXAMPLES:

```
sage: Poset([[1,2,3],[4],[4],[4],[]]).size()
5
```

top()
Returns the top element of the poset, if it exists.
EXAMPLES:

```
sage: P = Poset({0:[3],1:[3],2:[3],3:[4,5],4:[],5:[]})
sage: P.top() is None
True
sage: Q = Poset({0:[1],1:[]})
sage: Q.top()
1
```

upper_covers_iterator(*element*)
Returns the list of elements that cover `element`.
EXAMPLES:

```
sage: from sage.combinat.posets.hasse_diagram import HasseDiagram
sage: H = HasseDiagram({0:[1,3,2],1:[4],2:[4,5,6],3:[6],4:[7],5:[7],6:[7],7:[]})
sage: list(H.upper_covers_iterator(0))
[1, 2, 3]
sage: list(H.upper_covers_iterator(7))
[]
```

17.36.3 Elements of posets, lattices, semilattices, etc.

class JoinSemilatticeElement(*poset, element, vertex*)

class LatticePosetElement(*poset, element, vertex*)

class MeetSemilatticeElement(*poset, element, vertex*)

class PosetElement(*poset, element, vertex*)

17.36.4 SemiLattices and Lattices

class FiniteJoinSemilattice(*digraph, elements=None*)
We assume that the argument passed to FiniteJoinSemilattice is the poset of a join-semilattice (i.e. a poset with least upper bound for each pair of elements).
TESTS:

```
sage: J = JoinSemilattice([[1,2],[3],[3]])
sage: J == loads(dumps(J))
True
```

```
sage: P = Poset([[1,2],[3],[3]])
sage: J = JoinSemilattice(P)
sage: J == loads(dumps(J))
True
```

join(*x, y*)
Return the join of `self` and `other` in the lattice.
EXAMPLES:

```
sage: D = Posets.DiamondPoset(5)
sage: D(1) + D(2)
4
sage: D(1) + D(1)
```

```
1
sage: D(1) + D(4)
4
sage: D(1) + D(0)
1
```

class **FiniteLatticePoset**(*digraph, elements=None*)

We assume that the argument passed to FiniteLatticePoset is the poset of a lattice (i.e. a poset with greatest lower bound and least upper bound for each pair of elements).

TESTS:

```
sage: L = LatticePoset([[1,2],[3],[3]])
sage: L == loads(dumps(L))
True

sage: P = Poset([[1,2],[3],[3]])
sage: L = LatticePoset(P)
sage: L == loads(dumps(L))
True
```

complements()

Returns a list of the elements of the lattice.

A complement of `x` is an element `y` such that the meet of `x` and `y` is the bottom element of `self` and the join of `x` and `y` is the top element of `self`.

EXAMPLES:

```
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: L.complements()
[4, 3, 3, 2, 0]

sage: L = LatticePoset({0:[1,2],1:[3],2:[3],3:[4]})
sage: L.complements()
[4, None, None, None, 0]
```

is_complemented()

Returns `True` if `self` is a complemented lattice, and `False` otherwise.

EXAMPLES:

```
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: L.is_complemented()
True

sage: L = LatticePoset({0:[1,2],1:[3],2:[3],3:[4]})
sage: L.is_complemented()
False
```

is_distributive()

Returns `True` if the lattice is distributive, and `False` otherwise.

EXAMPLES:

```
sage: L = LatticePoset({0:[1,2],1:[3],2:[3]})
sage: L.is_distributive()
True
sage: L = LatticePoset({0:[1,2,3],1:[4],2:[4],3:[4]})
sage: L.is_distributive()
False
```

class **FiniteMeetSemilattice**(*digraph, elements=None*)

..note:: We assume that the argument passed to MeetSemilattice is the poset of a meet-semilattice (i.e. a poset with greatest lower bound for each pair of elements).

TESTS:

```
sage: M = MeetSemilattice([[1,2],[3],[3]])
sage: M == loads(dumps(M))
True
```

```
sage: P = Poset([[1,2],[3],[3]])
sage: M = MeetSemilattice(P)
sage: M == loads(dumps(M))
True
```

meet (*x*, *y*)

Return the meet of `self` and `other` in the lattice.

EXAMPLES:

```
sage: D = Posets.DiamondPoset(5)
sage: D(1) * D(2)
0
sage: D(1) * D(1)
1
sage: D(1) * D(0)
0
sage: D(1) * D(4)
1
```

JoinSemilattice (*data*)

Construct a join semi-lattice from various forms of input data.

INPUT:

- `data` - any data that defines a poset that is also a join semilattice. See the documentation for `Poset`.

EXAMPLES:

Using data that defines a poset:

```
sage: JoinSemilattice([[1,2],[3],[3]])
Finite join-semilattice containing 4 elements
```

Using a previously constructed poset:

```
sage: P = Poset([[1,2],[3],[3]])
sage: J = JoinSemilattice(P); J
Finite join-semilattice containing 4 elements
sage: type(J)
<class 'sage.combinat.posets.lattices.FiniteJoinSemilattice'>
```

If the data is not a lattice, then an error is raised:

```
sage: elms = [1,2,3,4,5,6,7]
sage: rels = [[1,2],[3,4],[4,5],[2,5]]
sage: JoinSemilattice((elms, rels))
...
ValueError: Not a join semilattice.
```

LatticePoset(*data*)

Construct a lattice from various forms of input data.

INPUT:

- `data` - any data that defines a poset. See the documentation for `Poset`.

OUTPUT:

FiniteLatticePoset – an instance of FiniteLatticePoset

EXAMPLES:

Using data that defines a poset:

```
sage: LatticePoset([[1,2],[3],[3]])
Finite lattice containing 4 elements
```

Using a previously constructed poset:

```
sage: P = Poset([[1,2],[3],[3]])
sage: L = LatticePoset(P); L
Finite lattice containing 4 elements
sage: type(L)
<class 'sage.combinat.posets.lattices.FiniteLatticePoset'>
```

If the data is not a lattice, then an error is raised:

```
sage: elms = [1,2,3,4,5,6,7]
sage: rels = [[1,2],[3,4],[4,5],[2,5]]
sage: LatticePoset((elms, rels))
...
ValueError: Not a lattice.
```

MeetSemilattice(*data*)

Construct a meet semi-lattice from various forms of input data.

INPUT:

- `data` - any data that defines a poset that is also a meet semilattice. See the documentation for `Poset`.

EXAMPLES:

Using data that defines a poset:

```
sage: MeetSemilattice([[1,2],[3],[3]])
Finite meet-semilattice containing 4 elements
```

Using a previously constructed poset:

```
sage: P = Poset([[1,2],[3],[3]])
sage: L = MeetSemilattice(P); L
Finite meet-semilattice containing 4 elements
sage: type(L)
<class 'sage.combinat.posets.lattices.FiniteMeetSemilattice'>
```

If the data is not a lattice, then an error is raised:

```
sage: elms = [1,2,3,4,5,6,7]
sage: rels = [[1,2],[3,4],[4,5],[2,5]]
sage: MeetSemilattice((elms, rels))
...
ValueError: Not a meet semilattice.
```

17.36.5 Some examples of posets and lattices.

AntichainPoset(**args, **kwds*)
This function is deprecated and will be removed in a future version of Sage. Please use `Posets.AntichainPoset` instead.

TESTS:: sage: AntichainPoset(3) doctest:1: DeprecationWarning: AntichainPoset is deprecated, use Posets.AntichainPoset instead! Finite poset containing 3 elements

BooleanLattice(**args, **kwds*)
This function is deprecated and will be removed in a future version of Sage. Please use `Posets.BooleanLattice` instead.

TESTS:: sage: BooleanLattice(3) doctest:1: DeprecationWarning: BooleanLattice is deprecated, use Posets.BooleanLattice instead! Finite lattice containing 8 elements

ChainPoset(**args, **kwds*)
This function is deprecated and will be removed in a future version of Sage. Please use `Posets.ChainPoset` instead.

TESTS:: sage: ChainPoset(3) doctest:1: DeprecationWarning: ChainPoset is deprecated, use Posets.ChainPoset instead! Finite lattice containing 3 elements

DiamondPoset(**args, **kwds*)
This function is deprecated and will be removed in a future version of Sage. Please use `Posets.DiamondPoset` instead.

TESTS:: sage: DiamondPoset(3) doctest:1: DeprecationWarning: DiamondPoset is deprecated, use Posets.DiamondPoset instead! Finite lattice containing 3 elements

PentagonPoset(**args, **kwds*)
This function is deprecated and will be removed in a future version of Sage. Please use `Posets.PentagonPoset` instead.

TESTS:: sage: PentagonPoset() doctest:1: DeprecationWarning: PentagonPoset is deprecated, use Posets.PentagonPoset instead! Finite lattice containing 5 elements

PosetOfIntegerCompositions(**args, **kwds*)
This function is deprecated and will be removed in a future version of Sage. Please use `Posets.IntegerCompositions` instead.

TESTS:: sage: PosetOfIntegerCompositions(3) doctest:1: DeprecationWarning: PosetOfIntegerCompositions is deprecated, use Posets.IntegerCompositions instead! Finite poset containing 4 elements

PosetOfIntegerPartitions(**args, **kwds*)
This function is deprecated and will be removed in a future version of Sage. Please use `Posets.IntegerPartitions` instead.

TESTS:: sage: PosetOfIntegerPartitions(3) doctest:1: DeprecationWarning: PosetOfIntegerPartitions is deprecated, use Posets.IntegerPartitions instead! Finite poset containing 3 elements

PosetOfRestrictedIntegerPartitions(**args, **kwds*)
This function is deprecated and will be removed in a future version of Sage. Please use `Posets.RestrictedIntegerPartitions` instead.

TESTS:: sage: PosetOfRestrictedIntegerPartitions(3) doctest:1: DeprecationWarning: PosetOfRestrictedIntegerPartitions is deprecated, use Posets.RestrictedIntegerPartitions instead! Finite poset containing 3 elements

class **PosetsGenerator**()

A collection of examples of posets.

EXAMPLES:: sage: P = Posets() sage: P == loads(dumps(P)) True

AntichainPoset(*n*)

Returns an antichain (a poset with no comparable elements) containing n elements.

EXAMPLES:

```
sage: A = Posets.AntichainPoset(6); A
Finite poset containing 6 elements
sage: for i in range(5):
...       for j in range(5):
...           if A.covers(A(i),A(j)):
...              print "TEST FAILED"
```

BooleanLattice(*n*)

Returns the Boolean lattice containing 2^n elements.

EXAMPLES:

```
sage: Posets.BooleanLattice(5)
Finite lattice containing 32 elements
```

ChainPoset(*n*)

Returns a chain (a totally ordered poset) containing n elements.

EXAMPLES:

```
sage: C = Posets.ChainPoset(6); C
Finite lattice containing 6 elements
sage: C.linear_extension()
[0, 1, 2, 3, 4, 5]
sage: for i in range(5):
...       for j in range(5):
...           if C.covers(C(i),C(j)) and j != i+1:
...              print "TEST FAILED"
```

DiamondPoset(*n*)

Returns the lattice of rank two containing n elements.

EXAMPLES:

```
sage: Posets.DiamondPoset(7)
Finite lattice containing 7 elements
```

IntegerCompositions(*n*)

Returns the poset of integer compositions of the integer n.

A composition of a positive integer n is a list of positive integers that sum to n. The order is reverse refinement: `[p_1,p_2,...,p_l] < [q_1,q_2,...,q_m]` if q consists of an integer composition of `p_1`, followed by an integer composition of `p_2`, and so on.

EXAMPLES:

```
sage: P = Posets.IntegerCompositions(7); P
Finite poset containing 64 elements
sage: len(P.cover_relations())
192
```

IntegerPartitions(*n*)

Returns the poset of integer partitions on the integer n.

A partition of a positive integer n is a non-increasing list of positive integers that sum to n. If p and q are integer partitions of n, then p covers q if and only if q is obtained from p by joining two parts of p (and sorting, if necessary).

EXAMPLES:

```
sage: P = Posets.IntegerPartitions(7); P
Finite poset containing 15 elements
sage: len(P.cover_relations())
28
```

PentagonPoset()

Return the "pentagon".

EXAMPLES:

```
sage: Posets.PentagonPoset()
Finite lattice containing 5 elements
```

RandomPoset(*n, p*)

Generate a random poset on n vertices according to a probability distribution p.

EXAMPLES:

```
sage: Posets.RandomPoset(17,.15)
Finite poset containing 17 elements
```

RestrictedIntegerPartitions(*n*)

Returns the poset of integer partitions on the integer n ordered by restricted refinement. That is, if p and q are integer partitions of n, then p covers q if and only if q is obtained from p by joining two distinct parts of p (and sorting, if necessary).

EXAMPLES:

```
sage: P = Posets.RestrictedIntegerPartitions(7); P
Finite poset containing 15 elements
sage: len(P.cover_relations())
17
```

SymmetricGroupBruhatIntervalPoset(*start, end*)

The poset of permutations with respect to Bruhat order.

INPUT:

- start - list permutation
- end - list permutation (same n, of course)

Note: Must have start <= end.

EXAMPLES:

Any interval is rank symmetric if and only if it avoids these permutations:

```
sage: P1 = Posets.SymmetricGroupBruhatIntervalPoset([0,1,2,3], [2,3,0,1])
sage: P2 = Posets.SymmetricGroupBruhatIntervalPoset([0,1,2,3], [3,1,2,0])
sage: ranks1 = [P1.rank(v) for v in P1]
sage: ranks2 = [P2.rank(v) for v in P2]
sage: [ranks1.count(i) for i in uniq(ranks1)]
[1, 3, 5, 4, 1]
sage: [ranks2.count(i) for i in uniq(ranks2)]
[1, 3, 5, 6, 4, 1]
```

SymmetricGroupBruhatOrderPoset(*n*)

The poset of permutations with respect to Bruhat order.

EXAMPLES:

```
sage: Posets.SymmetricGroupBruhatOrderPoset(4)
Finite poset containing 24 elements
```

SymmetricGroupWeakOrderPoset(*n, labels='permutations'*)

The poset of permutations with respect to weak order.

EXAMPLES:

```
sage: Posets.SymmetricGroupWeakOrderPoset(4)
Finite poset containing 24 elements
```

RandomPoset(**args, **kwds*)

This function is deprecated and will be removed in a future version of Sage. Please use `Posets.RandomPoset` instead.

TESTS:: sage: RandomPoset(17,.15) doctest:1: DeprecationWarning: RandomPoset is deprecated, use Posets.RandomPoset instead! Finite poset containing 17 elements

SymmetricGroupBruhatOrderPoset(**args, **kwds*)

This function is deprecated and will be removed in a future version of Sage. Please use `Posets.SymmetricGroupBruhatOrderPoset` instead.

TESTS:: sage: SymmetricGroupBruhatOrderPoset(3) doctest:1: DeprecationWarning: SymmetricGroupBruhatOrderPoset is deprecated, use Posets.SymmetricGroupBruhatOrderPoset instead! Finite poset containing 6 elements

SymmetricGroupWeakOrderPoset(**args, **kwds*)

This function is deprecated and will be removed in a future version of Sage. Please use `Posets.SymmetricGroupWeakOrderPoset` instead.

TESTS:: sage: SymmetricGroupWeakOrderPoset(3) doctest:1: DeprecationWarning: SymmetricGroupWeakOrderPoset is deprecated, use Posets.SymmetricGroupWeakOrderPoset instead! Finite poset containing 6 elements

17.37 Designs and Incidence Structures

17.37.1 Block designs.

A module to help with constructions and computations of block designs and other incidence structures.

A block design is an incidence structure consisting of a set of points P and a set of blocks B, where each block is considered as a subset of P. More precisely, a *block design* B is a class of k-element subsets of P such that the number r of blocks that contain any point x in P is independent of x, and the number lambda of blocks that contain any given t-element subset T is independent of the choice of T (see [1] for more). Such a block design is also called a t-(v,k,lambda)-design, and v (the number of points), b (the number of blocks), k, r, and lambda are the parameters of the design. (In Python, lambda is reserved, so we sometimes use lmbda or L instead.)

In Sage, sets are replaced by (ordered) lists and the standard representation of a block design uses P = [0,1,..., v-1], so a block design is specified by (v,B).

This software is released under the terms of the GNU General Public License, version 2 or above (your choice). For details on licencing, see the accompanying documentation.

REFERENCES:

- [1] Block design from wikipedia, http://en.wikipedia.org/wiki/Block_design
- [2] What is a block design?, http://designtheory.org/library/extrep/html/node4.html (in 'The External Representation of Block Designs' by Peter J. Cameron, Peter Dobcsanyi, John P. Morgan, Leonard H. Soicher)

This is a significantly modified form of the module block_design.py (version 0.6) written by Peter Dobcsanyi peter@designtheory.org. Thanks go to Robert Miller for lots of good design suggestions.

Copyright 2007-2008 by Peter Dobcsanyi peter@designtheory.org, and David Joyner wdjoyner@gmail.com.

TODO: Implement DerivedDesign, ComplementaryDesign, Hadamard3Design

AffineGeometryDesign(*n, d, F*)

Input: n is the Euclidean dimension, so the number of points is $v = |F^n|$ (F = GF(q), some q) d is the dimension of the (affine) subspaces of $P = GF(q)^n$ which make up the blocks.

$AG_{n,d}(F)$, as it is sometimes denoted, is a 2 - (v, k, λ) design of points and d- flats (cosets of dimension n) in the affine geometry $AG_n(F)$, where

$$v = q^n,\ k = q^d, \lambda = \frac{(q^{n-1}-1)\cdots(q^{n+1-d}-1)}{(q^{n-1}-1)\cdots(q-1)}.$$

Wraps some functions used in GAP Design's PGPointFlatBlockDesign. Does *not* require GAP's Design.

EXAMPLES:

```
sage: BD = AffineGeometryDesign(3, 1, GF(2))
sage: BD.parameters()
(2, 8, 2, 2)
sage: BD.is_block_design()
(True, [2, 8, 2, 2])
sage: BD = AffineGeometryDesign(3, 2, GF(2))
sage: BD.parameters()
(2, 8, 4, 12)
sage: BD.is_block_design()
(True, [3, 8, 4, 4])
```

BlockDesign(*max_pt, blks, name=None, test=True*)

Returns an instance of the IncidenceStructure class. Requires each B in blks to be contained in range(max_pt). Does not test if the result is a block design.

EXAMPLES:

```
sage: BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]], name="Fano plane"
Incidence structure with 7 points and 7 blocks
sage: print BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]], name="Fano
Fano plane<points=[0, 1, 2, 3, 4, 5, 6], blocks=[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1,
```

HadamardDesign(*n*)

As described in Section 1, p. 10, in [CvL]. The input n must have the property that there is a Hadamard matrix of order n+1 (and that a construction of that Hadamard matrix has been implemented...).

EXAMPLES:

```
sage: HadamardDesign(7)
Incidence structure with 7 points and 7 blocks
sage: print HadamardDesign(7)
HadamardDesign<points=[0, 1, 2, 3, 4, 5, 6], blocks=[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5],
```

REFERENCES:

•[CvL] P. Cameron, J. H. van Lint, Designs, graphs, codes and their links, London Math. Soc., 1991.

ProjectiveGeometryDesign(*n, d, F, method=None*)

Input: n is the projective dimension, so the number of points is v = PPn(GF(q)) d is the dimension of the subspaces of P = PPn(GF(q)) which make up the blocks, so b is the number of d-dimensional subspaces of P

Wraps GAP Design's PGPointFlatBlockDesign. Does *not* require GAP's Design.

EXAMPLES:

```
sage: ProjectiveGeometryDesign(2, 1, GF(2))
Incidence structure with 7 points and 7 blocks
sage: BD = ProjectiveGeometryDesign(2, 1, GF(2), method="gap")      # requires optional gap pack
sage: BD.is_block_design()                                          # requires optional gap package
(True, [2, 7, 3, 1])
```

WittDesign(*n*)

Input: n is in 9,10,11,12,21,22,23,24.

Wraps GAP Design's WittDesign. If n=24 then this function returns the large Witt design W24, the unique (up to isomorphism) 5-(24,8,1) design. If n=12 then this function returns the small Witt design W12, the unique (up to isomorphism) 5-(12,6,1) design. The other values of n return a block design derived from these.

REQUIRES: GAP's Design package.

EXAMPLES:

```
sage: BD = WittDesign(9)    # requires optional gap package
sage: BD.parameters()       # requires optional gap package
(2, 9, 3, 1)
sage: BD                    # requires optional gap package
Incidence structure with 9 points and 12 blocks
sage: print BD              # requires optional gap package
WittDesign<points=[0, 1, 2, 3, 4, 5, 6, 7, 8], blocks=[[0, 1, 7], [0, 2, 5], [0, 3, 4], [0, 6, 8
sage: BD = WittDesign(12)   # requires optional gap package
sage: BD.parameters(t=5)    # requires optional gap package
(5, 12, 6, 1)
```

tdesign_params(*t, v, k, L*)

Return the design's parameters: (t, v, b, r , k, L). Note t must be given.

EXAMPLES:

```
sage: BD = BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: from sage.combinat.designs.block_design import tdesign_params
sage: tdesign_params(2,7,3,1)
(2, 7, 7, 3, 3, 1)
```

17.37.2 Incidence structures.

An incidence structure is specified by a list of points, blocks, and an incidence matrix ([1], [2]).

Classes:

IncidenceStructure

This software is released under the terms of the GNU General Public License, version 2 or above (your choice). For details on licencing, see the accompanying documentation.

REFERENCES:

- [1] Block designs and incidence structures from wikipedia, http://en.wikipedia.org/wiki/Block_design http://en.wikipedia.org/wiki/Incidence_structure
- [2] E. Assmus, J. Key, Designs and their codes, CUP, 1992.

This is a significantly modified form of part of the module block_design.py (version 0.6) written by Peter Dobcsanyi peter@designtheory.org.

class IncidenceStructure(*pts, blks, inc_mat=None, name=None, test=True*)

This the base class for block designs.

automorphism_group()

Returns the subgroup of the automorphism group of the incidence graph which respects the P B partition. This is (isomorphic to) the automorphism group of the block design, although the degrees differ.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import BlockDesign
sage: BD = BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: G = BD.automorphism_group(); G
Permutation Group with generators [(4,5)(6,7), (4,6)(5,7), (2,3)(6,7), (2,4)(3,5), (1,2)(5,6
sage: BD = BlockDesign(4,[[0],[0,1],[1,2],[3,3]],test=False)
sage: G = BD.automorphism_group(); G
Permutation Group with generators [()]
sage: BD = BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: G = BD.automorphism_group(); G
Permutation Group with generators [(4,5)(6,7), (4,6)(5,7), (2,3)(6,7), (2,4)(3,5), (1,2)(5,6
```

block_design_checker(*t, v, k, lmbda, type=None*)

This is *not* a wrapper for GAP Design's IsBlockDesign. The GAP Design function IsBlockDesign http://www.gap-system.org/Manuals/pkg/design/htm/CHAP004.htmSSEC001.1 apparently simply checks the record structure and no mathematical properties. Instead, the function below checks some necessary (but not sufficient) "easy" identities arising from the identity.

INPUT:

- `t` - the t as in "t-design"
- `v` - the number of points
- `k` - the number of blocks incident to a point
- `lmbda` - each t-tuple of points should be incident with lmbda blocks
- `type` - can be 'simple' or 'binary' or 'connected' Depending on the option, this wraps IsBinaryBlockDesign, IsSimpleBlockDesign, or IsConnectedBlockDesign.
  - –Binary: no block has a repeated element.
  - –Simple: no block is repeated.
  - –Connected: its incidence graph is a connected graph.

WRNING: This is very fast but can return false positives.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import BlockDesign
sage: BD = BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: BD.parameters()
(2, 7, 3, 1)
sage: BD.block_design_checker(2, 7, 3, 1)
True
sage: BD.block_design_checker(2, 7, 3, 1,"binary")
True
sage: BD.block_design_checker(2, 7, 3, 1,"connected")
True
sage: BD.block_design_checker(2, 7, 3, 1,"simple")
True
```

block_sizes()

Return a list of block's sizes.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import BlockDesign
sage: BD = BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
```

```
sage: BD.block_sizes()
[3, 3, 3, 3, 3, 3, 3]
```

blocks()

Return the list of blocks.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import BlockDesign
sage: BD = BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: BD.blocks()
[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1, 3, 5], [1, 4, 6], [2, 3, 6], [2, 4, 5]]
```

dual_design(*method=None*)

Wraps GAP Design's DualBlockDesign (see [1]). The dual of a block design may not be a block design.

Also can be called with `dual_design`.

REQUIRES: method="gap" option requires GAP's Design package. method=None option does *not* require GAP's Design.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import BlockDesign
sage: D = BlockDesign(4, [[0,2],[1,2,3],[2,3]], test=False)
sage: D
Incidence structure with 4 points and 3 blocks
sage: D.dual_design()
Incidence structure with 3 points and 4 blocks
sage: print D.dual_design(method="gap")       # optional - gap_design
IncidenceStructure<points=[0, 1, 2], blocks=[[0], [0, 1, 2], [1], [1, 2]]>
sage: BD = IncidenceStructure(range(7),[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4
sage: BD
Incidence structure with 7 points and 7 blocks
sage: print BD.dual_design(method="gap")          # optional - gap_design
IncidenceStructure<points=[0, 1, 2, 3, 4, 5, 6], blocks=[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1
sage: BD.dual_incidence_structure()
Incidence structure with 7 points and 7 blocks
```

REFERENCE:

- Soicher, Leonard, Design package manual, available at http://www.gap-system.org/Manuals/pkg/design/htm/CHAP003.htm

dual_incidence_structure(*method=None*)

Wraps GAP Design's DualBlockDesign (see [1]). The dual of a block design may not be a block design.

Also can be called with `dual_design`.

REQUIRES: method="gap" option requires GAP's Design package. method=None option does *not* require GAP's Design.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import BlockDesign
sage: D = BlockDesign(4, [[0,2],[1,2,3],[2,3]], test=False)
sage: D
Incidence structure with 4 points and 3 blocks
sage: D.dual_design()
Incidence structure with 3 points and 4 blocks
sage: print D.dual_design(method="gap")       # optional - gap_design
IncidenceStructure<points=[0, 1, 2], blocks=[[0], [0, 1, 2], [1], [1, 2]]>
sage: BD = IncidenceStructure(range(7),[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4
sage: BD
Incidence structure with 7 points and 7 blocks
```

```
sage: print BD.dual_design(method="gap")          # optional - gap_design
IncidenceStructure<points=[0, 1, 2, 3, 4, 5, 6], blocks=[[0, 1, 2], [0, 3, 4], [0, 5, 6], [1
sage: BD.dual_incidence_structure()
Incidence structure with 7 points and 7 blocks
```

REFERENCE:

- Soicher, Leonard, Design package manual, available at http://www.gap-system.org/Manuals/pkg/design/htm/CHAP003.htm

incidence_graph()

Returns the incidence graph of the design, where the incidence matrix of the design is the adjacency matrix of the graph.

EXAMPLE:

```
sage: BD = IncidenceStructure(range(7),[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4
sage: BD.incidence_graph()
Bipartite graph on 14 vertices
sage: A = BD.incidence_matrix()
sage: Graph(block_matrix([A*0,A,A.transpose(),A*0])) == BD.incidence_graph()
True
```

REFERENCE:

- Sage Reference Manual on Graphs

incidence_matrix()

Return the incidence matrix A of the design. A is a (v x b) matrix defined by: A[i,j] = 1 if i is in block B_j 0 otherwise

EXAMPLES:

```
sage: BD = IncidenceStructure(range(7),[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4
sage: BD.block_sizes()
[3, 3, 3, 3, 3, 3, 3]
sage: BD.incidence_matrix()
[1 1 1 0 0 0 0]
[1 0 0 1 1 0 0]
[1 0 0 0 0 1 1]
[0 1 0 1 0 1 0]
[0 1 0 0 1 0 1]
[0 0 1 1 0 0 1]
[0 0 1 0 1 1 0]
```

is_block_design()

Returns a pair True, pars if the incidence structure is a t-design, for some t, where pars is the list of parameters [t, v, k, lmbda]. The largest possible t is returned, provided t=10.

EXAMPLES:

```
sage: BD = IncidenceStructure(range(7),[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4
sage: BD.is_block_design()
(True, [2, 7, 3, 1])
sage: BD.block_design_checker(2, 7, 3, 1)
True
sage: BD = WittDesign(9)         # requires optional gap package
sage: BD.is_block_design()       # requires optional gap package
(True, [2, 9, 3, 1])
sage: BD = WittDesign(12)        # requires optional gap package
sage: BD.is_block_design()       # requires optional gap package
(True, [5, 12, 6, 1])
sage: BD = AffineGeometryDesign(3, 1, GF(2))
```

```
sage: BD.is_block_design()
(True, [2, 8, 2, 2])
```

parameters(*t=2*)

Returns (t,v,k,lambda). Does not check if the input is a block design. Uses t=2 by default.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import BlockDesign
sage: BD = BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]], name="Fa
sage: BD.parameters()
(2, 7, 3, 1)
sage: BD.parameters(t=3)
(3, 7, 3, 0)
```

points()

Returns the list of points.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import BlockDesign
sage: BD = BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: BD.points()
[0, 1, 2, 3, 4, 5, 6]
```

points_from_gap()

Literally pushes this block design over to GAP and returns the points of that. Other than debugging, usefulness is unclear.

REQUIRES: GAP's Design package.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import BlockDesign
sage: BD = BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: BD.points_from_gap()      # requires optional gap package
[1, 2, 3, 4, 5, 6, 7]
```

IncidenceStructureFromMatrix(*M, name=None*)

M must be a (0,1)-matrix. Creates a set of "points" from the rows and a set of "blocks" from the columns.

EXAMPLES:

```
sage: from sage.combinat.designs.block_design import BlockDesign
sage: BD1 = BlockDesign(7,[[0,1,2],[0,3,4],[0,5,6],[1,3,5],[1,4,6],[2,3,6],[2,4,5]])
sage: M = BD1.incidence_matrix()
sage: BD2 = IncidenceStructureFromMatrix(M)
sage: BD1 == BD2
True
```

coordinatewise_product(*L*)

L is a list of n-vectors or lists all of length n with a common parent. This returns the vector whose i-th coordinate is the product of the i-th coordinates of the vectors.

EXAMPLES:

```
sage: from sage.combinat.designs.incidence_structures import coordinatewise_product
sage: L = [[1,2,3],[-1,-1,-1],[5,7,11]]
sage: coordinatewise_product(L)
[-5, -14, -33]
```

17.38 Combinatorial Species

17.38.1 Power Series

Streams or Infinite Arrays

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatse12.html.

Stream(*x=None, const=None*)

Returns a stream.

EXAMPLES: We can create a constant stream by just passing a

```
sage: from sage.combinat.species.stream import Stream
sage: s = Stream(const=0)
sage: [s[i] for i in range(10)]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

class **Stream_class**(*gen=None, const=None, func=None*)

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: from itertools import izip
sage: s = Stream(const=0)
sage: len(s)
1
sage: [x for (x,i) in izip(s, range(4))]
[0, 0, 0, 0]
sage: len(s)
1
```

```
sage: s = Stream(const=4)
sage: g = iter(s)
sage: l1 = [x for (x,i) in izip(g, range(10))]
sage: l = [4 for k in range(10)]
sage: l == l1
True
```

```
sage: h = lambda l: 1 if len(l) < 2 else l[-1] + l[-2]
sage: fib = Stream(h)
sage: [x for (x,i) in izip(fib, range(11))]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

```
sage: s = Stream()
sage: l = Stream([1, 0])
sage: y = Stream([0,1,0])
sage: s.set_gen(iter(l + y * s * s))
sage: [x for (x,i) in izip(s, range(6))]
[1, 1, 2, 5, 14, 42]
```

```
sage: r = [4, 3, 5, 2, 6, 1, 1, 1, 1, 1]
sage: l = [4, 3, 5, 2, 6, 1]
sage: s = Stream(l)
sage: s[3] = -1
```

```
sage: [x for (x,i) in izip(s, r)]
[4, 3, 5, -1, 6, 1, 1, 1, 1, 1]
sage: s[5] = -2
sage: [x for (x,i) in izip(s, r)]
[4, 3, 5, -1, 6, -2, 1, 1, 1, 1]
sage: s[6] = -3
sage: [x for (x,i) in izip(s, r)]
[4, 3, 5, -1, 6, -2, -3, 1, 1, 1]
sage: s[8] = -4
sage: [x for (x,i) in izip(s, r)]
[4, 3, 5, -1, 6, -2, -3, 1, -4, 1]
sage: a = Stream(const=0)
sage: a[2] = 3
sage: [x for (x,i) in izip(a, range(4))]
[0, 0, 3, 0]
```

data()

Returns a list of all the coefficients computed so far.

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream, _integers_from
sage: s = Stream(_integers_from(3))
sage: s.data()
[]
sage: s[5]
8
sage: s.data()
[3, 4, 5, 6, 7, 8]
```

is_constant()

Returns True if and only if

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: s = Stream([1,2,3])
sage: s.is_constant()
False
sage: s[3]
3
sage: s.data()
[1, 2, 3]
sage: s.is_constant()
True
```

TESTS:

```
sage: l = [2,3,5,7,11,0]
sage: s = Stream(l)
sage: s.is_constant()
False
sage: s[3]
7
sage: s.is_constant()
False
sage: s[5]
0
sage: s.is_constant()
False
sage: s[6]
```

```
0
sage: s.is_constant()
True

sage: s = Stream(const='I am constant.')
sage: s.is_constant()
True
```

map(*f*)

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: s = Stream(ZZ)
sage: square = lambda x: x^2
sage: ss = s.map(square)
sage: [ss[i] for i in range(10)]
[0, 1, 1, 4, 4, 9, 9, 16, 16, 25]
```

TESTS:

```
sage: from itertools import izip
sage: f = lambda l: 0 if len(l) == 0 else l[-1] + 1
sage: o = Stream(f)
sage: [x for (x,i) in izip(o, range(10))]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: double = lambda z: 2*z
sage: t = o.map(double)
sage: [x for (x,i) in izip(t, range(10))]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]

sage: double = lambda z: 2*z
sage: o = Stream([0,1,2,3])
sage: [x for (x,i) in izip(o, range(6))]
[0, 1, 2, 3, 3, 3]
sage: t = o.map(double)
sage: [x for (x,i) in izip(t, range(6))]
[0, 2, 4, 6, 6, 6]
```

number_computed()

Returns the number of coefficients computed so far.

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: l = [4,3,5,7,4,1,9,7]
sage: s = Stream(l)
sage: s[3]
7
sage: len(s)
4
sage: s[3]
7
sage: len(s)
4
sage: s[1]
3
sage: len(s)
4
sage: s[4]
4
```

```
sage: len(s)
5
```

TESTS:

```
sage: l = ['Hello', ' ', 'World', '!']
sage: s = Stream(l)
sage: len(s)
0
sage: s[2]
'World'
sage: len(s)
3
sage: u = ""
sage: for i in range(len(s)): u += s[i]
sage: u
'Hello World'
sage: v = ""
sage: for i in range(10): v += s[i]
sage: v
'Hello World!!!!!!!'
sage: len(s)
4
```

set_gen(*gen*)

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: from itertools import izip
sage: fib = Stream()
sage: def g():
...           yield 1
...           yield 1
...           n = 0
...           while True:
...               yield fib[n] + fib[n+1]
...               n += 1

sage: fib.set_gen(g())
sage: [x for (x,i) in izip(fib, range(11))]
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]

sage: l = [4,3,5,2,6,1]
sage: s = Stream(l)
sage: s[3]
2
sage: len(s)
4
sage: g = iter(l)
sage: s.set_gen(g)
sage: s[5]
3
sage: len(s)
6
```

stretch(*k*)

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: s = Stream(range(1, 10))
```

```
sage: s2 = s.stretch(2)
sage: [s2[i] for i in range(10)]
[1, 0, 2, 0, 3, 0, 4, 0, 5, 0]
```

Series Order

This file provides some utility classes which are useful when working with unknown, known, and infinite series orders for univariate power series.

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatsu30.html.

class **InfiniteSeriesOrder**()

class **SeriesOrderElement**()

class **UnknownSeriesOrder**()

bounded_decrement(*x*)

EXAMPLES:

```
sage: from sage.combinat.species.series_order import *
sage: u = UnknownSeriesOrder()
sage: bounded_decrement(u)
Unknown series order
sage: bounded_decrement(4)
3
sage: bounded_decrement(0)
0
```

increment(*x*)

EXAMPLES:

```
sage: from sage.combinat.species.series_order import *
sage: u = UnknownSeriesOrder()
sage: increment(u)
Unknown series order
sage: increment(2)
3
```

Lazy Power Series

This file provides an implementation of lazy univariate power series, which uses the stream class for its internal data structure. The lazy power series keep track of their approximate order as much as possible without forcing the computation of any additional coefficients. This is required for recursively defined power series.

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatse9.html.

class **LazyPowerSeries**(*A*, *stream=None*, *order=None*, *aorder=None*, *aorder_changed=True*, *is_initialized=False*, *name=None*)

add(*y*)

EXAMPLES: Test Plus 1

```
sage: from sage.combinat.species.series import *
sage: from sage.combinat.species.stream import Stream
sage: L = LazyPowerSeriesRing(QQ)
sage: gs0 = L([0])
sage: gs1 = L([1])
sage: sum1 = gs0 + gs1
sage: sum2 = gs1 + gs1
sage: sum3 = gs1 + gs0
sage: [gs0.coefficient(i) for i in range(11)]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
sage: [gs1.coefficient(i) for i in range(11)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
sage: [sum1.coefficient(i) for i in range(11)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
sage: [sum2.coefficient(i) for i in range(11)]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
sage: [sum3.coefficient(i) for i in range(11)]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

Test Plus 2

```
sage: gs1 = L([1,2,4,8,0])
sage: gs2 = L([-1, 0,-1,-9,22,0])
sage: sum = gs1 + gs2
sage: sum2 = gs2 + gs1
sage: [ sum.coefficient(i) for i in range(5) ]
[0,  2, 3, -1, 22]
sage: [ sum.coefficient(i) for i in range(5, 11) ]
[0, 0, 0, 0, 0, 0]
sage: [ sum2.coefficient(i) for i in range(5) ]
[0,  2, 3, -1, 22]
sage: [ sum2.coefficient(i) for i in range(5, 11) ]
[0, 0, 0, 0, 0, 0]
```

coefficient(*n*)

Returns the coefficient of xn in self.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L(ZZ)
sage: [f.coefficient(i) for i in range(5)]
[0, 1, -1, 2, -2]
```

coefficients(*n*)

Returns the first n coefficients of self.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L([1,2,3,0])
sage: f.coefficients(5)
[1, 2, 3, 0, 0]
```

compose(*y*)

Returns the composition of this power series and the power series y.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: s = L([1])
sage: t = L([0,0,1])
```

```
sage: u = s(t)
sage: u.coefficients(11)
[1, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Test Compose 2

```
sage: s = L([1])
sage: t = L([0,0,1,0])
sage: u = s(t)
sage: u.aorder
0
sage: u.order
Unknown series order
sage: u.coefficients(10)
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
sage: u.aorder
0
sage: u.order
0
```

Test Compose 3 s = 1/(1-x), t = x/(1-x) s(t) = (1-x)/(1-2x)

```
sage: s = L([1])
sage: t = L([0,1])
sage: u = s(t)
sage: u.coefficients(14)
[1, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
```

composition(*y*)

Returns the composition of this power series and the power series y.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: s = L([1])
sage: t = L([0,0,1])
sage: u = s(t)
sage: u.coefficients(11)
[1, 0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

Test Compose 2

```
sage: s = L([1])
sage: t = L([0,0,1,0])
sage: u = s(t)
sage: u.aorder
0
sage: u.order
Unknown series order
sage: u.coefficients(10)
[1, 0, 1, 0, 1, 0, 1, 0, 1, 0]
sage: u.aorder
0
sage: u.order
0
```

Test Compose 3 s = 1/(1-x), t = x/(1-x) s(t) = (1-x)/(1-2x)

```
sage: s = L([1])
sage: t = L([0,1])
sage: u = s(t)
sage: u.coefficients(14)
```

```
[1, 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]
```

compute_aorder(**args, **kwargs*)

The default compute_aorder does nothing.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L(1)
sage: a.compute_aorder() is None
True
```

compute_coefficients(*i*)

Computes all the coefficients of self up to i.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L([1,2,3])
sage: a.compute_coefficients(5)
sage: a
1 + 2*x + 3*x^2 + 3*x^3 + 3*x^4 + 3*x^5 + ...
```

define(*x*)

EXAMPLES: Test Recursive 0

```
sage: L = LazyPowerSeriesRing(QQ)
sage: one = L(1)
sage: monom = L.gen()
sage: s = L()
sage: s._name = 's'
sage: s.define(one+monom*s)
sage: s.aorder
0
sage: s.order
Unknown series order
sage: [s.coefficient(i) for i in range(6)]
[1, 1, 1, 1, 1, 1]
```

Test Recursive 1

```
sage: s = L()
sage: s._name = 's'
sage: s.define(one+monom*s*s)
sage: s.aorder
0
sage: s.order
Unknown series order
sage: [s.coefficient(i) for i in range(6)]
[1, 1, 2, 5, 14, 42]
```

Test Recursive 1b

```
sage: s = L()
sage: s._name = 's'
sage: s.define(monom + s*s)
sage: s.aorder
1
sage: s.order
Unknown series order
sage: [s.coefficient(i) for i in range(7)]
[0, 1, 1, 2, 5, 14, 42]
```

Test Recursive 2

```
sage: s = L()
sage: s._name = 's'
sage: t = L()
sage: t._name = 't'
sage: s.define(one+monom*t*t*t)
sage: t.define(one+monom*s*s)
sage: [s.coefficient(i) for i in range(9)]
[1, 1, 3, 9, 34, 132, 546, 2327, 10191]
sage: [t.coefficient(i) for i in range(9)]
[1, 1, 2, 7, 24, 95, 386, 1641, 7150]
```

Test Recursive 2b

```
sage: s = L()
sage: s._name = 's'
sage: t = L()
sage: t._name = 't'
sage: s.define(monom + t*t*t)
sage: t.define(monom + s*s)
sage: [s.coefficient(i) for i in range(9)]
[0, 1, 0, 1, 3, 3, 7, 30, 63]
sage: [t.coefficient(i) for i in range(9)]
[0, 1, 1, 0, 2, 6, 7, 20, 75]
```

Test Recursive 3

```
sage: s = L()
sage: s._name = 's'
sage: s.define(one+monom*s*s*s)
sage: [s.coefficient(i) for i in range(10)]
[1, 1, 3, 12, 55, 273, 1428, 7752, 43263, 246675]
```

derivative()

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: L = LazyPowerSeriesRing(QQ)
sage: one = L(1)
sage: monom = L.gen()
sage: s = L([1])
sage: u = s.derivative()
sage: u.coefficients(10)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sage: s = L()
sage: s._name = 's'
sage: s.define(one+monom*s*s)
sage: u = s.derivative()
sage: u.coefficients(5) #[1*1, 2*2, 3*5, 4*14, 5*42]
[1, 4, 15, 56, 210]

sage: s = L([1])
sage: t = L([0,1])
sage: u = s(t).derivative()
sage: v = (s.derivative().compose(t))*t.derivative()
sage: u.coefficients(11)
[1, 4, 12, 32, 80, 192, 448, 1024, 2304, 5120, 11264]
sage: v.coefficients(11)
[1, 4, 12, 32, 80, 192, 448, 1024, 2304, 5120, 11264]
```

```
sage: s = L(); s._name='s'
sage: t = L(); t._name='t'
sage: s.define(monom+t*t*t)
sage: t.define(monom+s*s)
sage: u = (s*t).derivative()
sage: v = s.derivative()*t + s*t.derivative()
sage: u.coefficients(10)
[0, 2, 3, 4, 30, 72, 133, 552, 1791, 4260]
sage: v.coefficients(10)
[0, 2, 3, 4, 30, 72, 133, 552, 1791, 4260]
sage: u.coefficients(10) == v.coefficients(10)
True

sage: f = L._new_initial(2, Stream([0,0,4,5,6,0]))
sage: d = f.derivative()
sage: d.get_aorder()
1
sage: d.coefficients(5)
[0, 8, 15, 24, 0]
```

exponential()

TESTS:

```
sage: def inv_factorial():
...       q = 1
...       yield 0
...       yield q
...       n = 2
...       while True:
...           q = q / n
...           yield q
...           n += 1
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L(inv_factorial()) #e^(x)-1
sage: u = f.exponential()
sage: g = inv_factorial()
sage: z1 = [1,1,2,5,15,52,203,877,4140,21147,115975]
sage: l1 = [z*g.next() for z in z1]
sage: l1 = [1] + l1[1:]
sage: u.coefficients(11)
[1, 1, 1, 5/6, 5/8, 13/30, 203/720, 877/5040, 23/224, 1007/17280, 4639/145152]
sage: l1 == u.coefficients(11)
True
```

get_aorder()

Returns the approximate order of self.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L.gen()
sage: a.get_aorder()
1
```

get_order()

Returns the order of self.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L.gen()
```

```
sage: a.get_order()
1
```

get_stream()

Returns self's underlying Stream object.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L.gen()
sage: s = a.get_stream()
sage: [s[i] for i in range(5)]
[0, 1, 0, 0, 0]
```

initialize_coefficient_stream(*compute_coefficients*)

Initializes the coefficient stream.

INPUT: compute_coefficients

TESTS:

```
sage: from sage.combinat.species.series_order import inf, unk
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L()
sage: compute_coefficients = lambda ao: iter(ZZ)
sage: f.order = inf
sage: f.aorder = inf
sage: f.initialize_coefficient_stream(compute_coefficients)
sage: f.coefficients(5)
[0, 0, 0, 0, 0]

sage: f = L()
sage: compute_coefficients = lambda ao: iter(ZZ)
sage: f.order = 1
sage: f.aorder = 1
sage: f.initialize_coefficient_stream(compute_coefficients)
sage: f.coefficients(5)
[0, 1, -1, 2, -2]
```

integral(*integration_constant=0*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: zero = L(0)
sage: s = zero
sage: t = s.integral()
sage: t.is_zero()
True

sage: s = zero
sage: t = s.integral(1)
sage: t.coefficients(6)
[1, 0, 0, 0, 0, 0]
sage: t._stream.is_constant()
True

sage: s = L.term(1, 0)
sage: t = s.integral()
sage: t.coefficients(6)
[0, 1, 0, 0, 0, 0]
sage: t._stream.is_constant()
True
```

```
sage: s = L.term(1,0)
sage: t = s.integral(1)
sage: t.coefficients(6)
[1, 1, 0, 0, 0, 0]
sage: t._stream.is_constant()
True

sage: s = L.term(1, 4)
sage: t = s.integral()
sage: t.coefficients(10)
[0, 0, 0, 0, 0, 1/5, 0, 0, 0, 0]

sage: s = L.term(1,4)
sage: t = s.integral(1)
sage: t.coefficients(10)
[1, 0, 0, 0, 0, 1/5, 0, 0, 0, 0]
```

TESTS:

```
sage: from sage.combinat.species.stream import Stream
sage: f = L._new_initial(2, Stream([0,0,4,5,6,0]))
sage: i = f.derivative().integral()
sage: i.get_aorder()
2
sage: i.coefficients(5)
[0, 0, 4, 5, 6]
sage: i = f.derivative().integral(1)
sage: i.get_aorder()
0
sage: i.coefficients(5)
[1, 0, 4, 5, 6]
```

is_finite(*n=None*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L([0,0,1,0,0]); a
O(1)
sage: a.is_finite()
False
sage: c = a[4]
sage: a.is_finite()
False
sage: a.is_finite(4)
False
sage: c = a[5]
sage: a.is_finite()
True
sage: a.is_finite(4)
True
```

is_zero()

Returns True if and only if self is zero.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: s = L([0,2,3,0])
sage: s.is_zero()
False
```

```
sage: s = L(0)
sage: s.is_zero()
True

sage: s = L([0])
sage: s.is_zero()
False
sage: s.coefficient(0)
0
sage: s.coefficient(1)
0
sage: s.is_zero()
True
```

iterator(*n=0, initial=None*)

Returns an iterator for the coefficients of self starting at n.

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L(range(10))
sage: g = f.iterator(2)
sage: [g.next() for i in range(5)]
[2, 3, 4, 5, 6]
sage: g = f.iterator(2, initial=[0,0])
sage: [g.next() for i in range(5)]
[0, 0, 2, 3, 4]
```

refine_aorder()

Refines the approximate order of self as much as possible without computing any coefficients.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L([0,0,0,0,1])
sage: a.aorder
0
sage: a.coefficient(2)
0
sage: a.aorder
0
sage: a.refine_aorder()
sage: a.aorder
3

sage: a = L([0,0])
sage: a.aorder
0
sage: a.coefficient(5)
0
sage: a.refine_aorder()
sage: a.aorder
Infinite series order

sage: a = L([0,0,1,0,0,0])
sage: a[4]
0
sage: a.refine_aorder()
sage: a.aorder
2
```

restricted(*min=None, max=None*)

Returns the power series restricted to the coefficients starting at min and going up to, but not including max. If min is not specified, then it is assumed to be zero. If max is not specified, then it is assumed to be infinity.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: a = L([1])
sage: a.restricted().coefficients(10)
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
sage: a.restricted(min=2).coefficients(10)
[0, 0, 1, 1, 1, 1, 1, 1, 1, 1]
sage: a.restricted(max=5).coefficients(10)
[1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
sage: a.restricted(min=2, max=6).coefficients(10)
[0, 0, 1, 1, 1, 1, 0, 0, 0, 0]
```

set_approximate_order(*new_order*)

Sets the approximate order of self and returns True if the approximate order has changed otherwise it will return False.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L([0,0,0,3,2,1,0])
sage: f.get_aorder()
0
sage: f.set_approximate_order(3)
True
sage: f.set_approximate_order(3)
False
```

tail()

Returns the power series whose coefficients obtained by subtracting the constant term from this series and then dividing by x.

EXAMPLES:

```
sage: from sage.combinat.species.stream import Stream
sage: L = LazyPowerSeriesRing(QQ)
sage: f = L(range(20))
sage: g = f.tail()
sage: g.coefficients(10)
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

times(*y*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: gs0 = L(0)
sage: gs1 = L([1])

sage: prod0 = gs0 * gs1
sage: [prod0.coefficient(i) for i in range(11)]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

sage: prod1 = gs1 * gs0
sage: [prod1.coefficient(i) for i in range(11)]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

```
sage: prod2 = gs1 * gs1
sage: [prod2.coefficient(i) for i in range(11)]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]

sage: gs1 = L([1,2,4,8,0])
sage: gs2 = L([-1, 0,-1,-9,22,0])

sage: prod1 = gs1 * gs2
sage: [prod1.coefficient(i) for i in range(11)]
[-1, -2, -5, -19, 0, 0, 16, 176, 0, 0, 0]

sage: prod2 = gs2 * gs1
sage: [prod2.coefficient(i) for i in range(11)]
[-1, -2, -5, -19, 0, 0, 16, 176, 0, 0, 0]
```

class **LazyPowerSeriesRing**(*R, element_class=None, names=None*)

gen(*i=0*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: L.gen().coefficients(5)
[0, 1, 0, 0, 0]
```

identity_element()

Returns the one power series.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: L.identity_element()
1
```

ngens()

EXAMPLES:

```
sage: LazyPowerSeriesRing(QQ).ngens()
1
```

product_generator(*g*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: s1 = L([1,1,0])
sage: s2 = L([1,0,1,0])
sage: s3 = L([1,0,0,1,0])
sage: s4 = L([1,0,0,0,1,0])
sage: s5 = L([1,0,0,0,0,1,0])
sage: s6 = L([1,0,0,0,0,0,1,0])
sage: s = [s1, s2, s3, s4, s5, s6]
sage: def g():
...       for a in s:
...           yield a
sage: p = L.product_generator(g())
sage: p.coefficients(26)
[1, 1, 1, 2, 2, 3, 4, 4, 4, 5, 5, 5, 5, 4, 4, 4, 3, 2, 2, 1, 1, 1, 0, 0, 0, 0]

sage: def m(n):
...       yield 1
...       while True:
```

```
...            for i in range(n-1):
...                yield 0
...            yield 1
...
sage: def s(n):
...        q = 1/n
...        yield 0
...        while True:
...            for i in range(n-1):
...                yield 0
...            yield q
...

sage: def lhs_gen():
...        n = 1
...        while True:
...            yield L(m(n))
...            n += 1
...

sage: def rhs_gen():
...        n = 1
...        while True:
...            yield L(s(n))
...            n += 1
...
sage: lhs = L.product_generator(lhs_gen())
sage: rhs = L.sum_generator(rhs_gen()).exponential()
sage: lhs.coefficients(10)
[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]
sage: rhs.coefficients(10)
[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]
```

sum(*a*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: l = [L(ZZ)]*3
sage: L.sum(l).coefficients(10)
[0, 3, -3, 6, -6, 9, -9, 12, -12, 15]
```

sum_generator(*g*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: g = [L([1])]*6 + [L(0)]
sage: t = L.sum_generator(g)
sage: t.coefficients(10)
[1, 2, 3, 4, 5, 6, 6, 6, 6, 6]

sage: s = L([1])
sage: def g():
...        while True:
...            yield s
sage: t = L.sum_generator(g())
sage: t.coefficients(9)
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

term(*r*, *n*)

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: L.term(0,0)
0
sage: L.term(3,2).coefficients(5)
[0, 0, 3, 0, 0]
```

zero_element()

Returns the zero power series.

EXAMPLES:

```
sage: L = LazyPowerSeriesRing(QQ)
sage: L.zero_element()
0
```

uninitialized()

EXAMPLES:

```
sage: from sage.combinat.species.series import uninitialized
sage: uninitialized()
...
RuntimeError: we should never be here
```

Generating Series

This file makes a number of extensions to lazy power series by endowing them with some semantic content for how they're to be interpreted.

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatse10.html. One notable difference is that we use power-sum symmetric functions as the coefficients of our cycle index series.

TESTS:

```
sage: from sage.combinat.species.stream import Stream, _integers_from
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: p = SFAPower(QQ)
sage: CIS = CycleIndexSeriesRing(QQ)


sage: geo1 = CIS((p([1])^i  for i in _integers_from(0)))
sage: geo2 = CIS((p([2])^i  for i in _integers_from(0)))
sage: s = geo1 * geo2
sage: s[0]
p[]
sage: s[1]
p[1] + p[2]
sage: s[2]
p[1, 1] + p[2, 1] + p[2, 2]
sage: s[3]
p[1, 1, 1] + p[2, 1, 1] + p[2, 2, 1] + p[2, 2, 2]
```

Whereas the coefficients of the above test are homogeneous with respect to total degree, the following test groups with respect to weighted degree where each variable x_i has weight i.

```
sage: def g():
...       for i in _integers_from(0):
...           yield p([2])^i
...           yield p(0)
sage: geo1 = CIS((p([1])^i  for i in _integers_from(0)))
sage: geo2 = CIS(g())
sage: s = geo1 * geo2
sage: s[0]
p[]
sage: s[1]
p[1]
sage: s[2]
p[1, 1] + p[2]
sage: s[3]
p[1, 1, 1] + p[2, 1]
sage: s[4]
p[1, 1, 1, 1] + p[2, 1, 1] + p[2, 2]
```

class CycleIndexSeries (*A, stream=None, order=None, aorder=None, aorder_changed=True, is_initialized=False, name=None*)

coefficient_cycle_type (*t*)

Returns the coefficient of a cycle type t.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: p = SFAPower(QQ)
sage: CIS = CycleIndexSeriesRing(p)
sage: f = CIS([0, p([1]), 2*p([1,1]),3*p([2,1])])
sage: f.coefficient_cycle_type([1])
1
sage: f.coefficient_cycle_type([1,1])
2
sage: f.coefficient_cycle_type([2,1])
3
```

count (*t*)

Returns the number of structures corresponding to a certain cycle type.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: p = SFAPower(QQ)
sage: CIS = CycleIndexSeriesRing(p)
sage: f = CIS([0, p([1]), 2*p([1,1]),3*p([2,1])])
sage: f.count([1])
1
sage: f.count([1,1])
4
sage: f.count([2,1])
6
```

functorial_composition (*g*)

Returns the functorial composition of self and g.

If F, G, and H are combinatorial species such that ' H = F Box G ', $then the cycle index series ' Z_H = Z_F \Box Z_G$ is defined as

EXAMPLES:

```
sage: S = species.SimpleGraphSpecies()
sage: S.cycle_index_series().coefficients(5)
[p[],
 p[1],
 p[1, 1] + p[2],
 4/3*p[1, 1, 1] + 2*p[2, 1] + 2/3*p[3],
 8/3*p[1, 1, 1, 1] + 4*p[2, 1, 1] + 2*p[2, 2] + 4/3*p[3, 1] + p[4]]
```

generating_series()

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: cis = P.cycle_index_series()
sage: f = cis.generating_series()
sage: f.coefficients(5)
[1, 1, 1, 5/6, 5/8]
```

isotype_generating_series()

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: cis = P.cycle_index_series()
sage: f = cis.isotype_generating_series()
sage: f.coefficients(10)
[1, 1, 2, 3, 5, 7, 11, 15, 22, 30]
```

stretch(k)

Returns the stretch of a cycle index series by a positive integer k.

If

$$f = \sum_{n=0}^{\infty} f_n(x_1, x_2, \ldots, x_n),$$

then the stretch g of f by k is

$$g = \sum_{n=0}^{\infty} f_n(x_k, x_{2k}, \ldots, x_{nk}).$$

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import CycleIndexSeriesRing
sage: p = SFAPower(QQ)
sage: CIS = CycleIndexSeriesRing(p)
sage: f = CIS([p([1])])
sage: f.stretch(3).coefficients(10)
[p[3], 0, 0, p[3], 0, 0, p[3], 0, 0, p[3]]
```

weighted_composition(*y_species*)

Returns the composition of this cycle index series with the cycle index series of y_species where y_species is a weighted species. Note that this is basically the same algorithm as composition except we can't use the optimization that the powering of cycle index series commutes with 'stretching'.

EXAMPLES:

```
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: E_cis = E.cycle_index_series()
sage: E_cis.weighted_composition(C).coefficients(4)
[p[], p[1], p[1, 1] + p[2], p[1, 1, 1] + p[2, 1] + p[3]]
sage: E(C).cycle_index_series().coefficients(4)
[p[], p[1], p[1, 1] + p[2], p[1, 1, 1] + p[2, 1] + p[3]]
```

class `CycleIndexSeriesRing_class`(*R*)

class `ExponentialGeneratingSeries`(*A, stream=None, order=None, aorder=None, aorder_changed=True, is_initialized=False, name=None*)

`count`(*n*)

Returns the number of structures of size n.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import ExponentialGeneratingSeriesRing
sage: R = ExponentialGeneratingSeriesRing(QQ)
sage: f = R([1])
sage: [f.count(i) for i in range(7)]
[1, 1, 2, 6, 24, 120, 720]
```

`counts`(*n*)

Returns the number of structures on a set for size i for i in range(n).

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import ExponentialGeneratingSeriesRing
sage: R = ExponentialGeneratingSeriesRing(QQ)
sage: f = R(range(20))
sage: f.counts(5)
[0, 1, 4, 18, 96]
```

`functorial_composition`(*y*)

Returns the exponential generating series which is the functorial composition of self with y.

If $f = \sum_{n=0}^{\infty} f_n \frac{x^n}{n!}$ and $g = \sum_{n=0}^{\infty} f_n \frac{x^n}{n!}$, then functorial composition $f \Box g$ is defined as

$$f \Box g = \sum_{n=0}^{\infty} f_{g_n} \frac{x^n}{n!}$$

REFERENCES:

•Section 2.2 of BLL.

EXAMPLES:

```
sage: G = species.SimpleGraphSpecies()
sage: g = G.generating_series()
sage: g.coefficients(10)
[1, 1, 1, 4/3, 8/3, 128/15, 2048/45, 131072/315, 2097152/315, 536870912/2835]
```

class `ExponentialGeneratingSeriesRing_class`(*R*)

class `OrdinaryGeneratingSeries`(*A, stream=None, order=None, aorder=None, aorder_changed=True, is_initialized=False, name=None*)

`count`(*n*)

Returns the number of structures on a set of size n.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import OrdinaryGeneratingSeriesRing
sage: R = OrdinaryGeneratingSeriesRing(QQ)
sage: f = R(range(20))
sage: f.count(10)
10
```

`counts`(*n*)

Returns the number of structures on a set for size i for i in range(n).

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import OrdinaryGeneratingSeriesRing
sage: R = OrdinaryGeneratingSeriesRing(QQ)
sage: f = R(range(20))
sage: f.counts(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

class **OrdinaryGeneratingSeriesRing_class**(*R*)

factorial_gen()

A generator for the factorials starting at 0.

EXAMPLES:

```
sage: from sage.combinat.species.generating_series import factorial_gen
sage: g = factorial_gen()
sage: [g.next() for i in range(5)]
[1, 1, 2, 6, 24]
```

17.38.2 Basic Species

Combinatorial Species

This file defines the main classes for working with combinatorial species, operations on them, as well as some implementations of basic species required for other constructions.

This code is based on the work of Ralf Hemmecke and Martin Rubey's Aldor-Combinat, which can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/index.html. In particular, the relevant section for this file can be found at http://www.risc.uni-linz.ac.at/people/hemmecke/AldorCombinat/combinatse8.html.

Weighted Species:

As a first application of weighted species, we count unlabeled ordered trees by total number of nodes and number of internal nodes. To achieve this, we assign a weight of 1 to the leaves and of q to internal nodes:

```
sage: q = QQ['q'].gen()
sage: leaf = species.SingletonSpecies()
sage: internal_node = species.SingletonSpecies(weight=q)
sage: L = species.LinearOrderSpecies(min=1)
sage: T = species.CombinatorialSpecies()
sage: T.define(leaf + internal_node*L(T))
sage: T.isotype_generating_series().coefficients(6)
[0, 1, q, q^2 + q, q^3 + 3*q^2 + q, q^4 + 6*q^3 + 6*q^2 + q]
```

Consider the following:

```
sage: T.isotype_generating_series().coefficient(4)
q^3 + 3*q^2 + q
```

This means that, among the trees on 4 nodes, one has a single internal node, three have two internal nodes, and one has three internal nodes.

class **GenericCombinatorialSpecies**(*min=None, max=None, weight=None*)

algebraic_equation_system()

Returns a system of algebraic equations satisfied by this species. The nodes are numbered in the order that they appear as vertices of the associated digraph.

EXAMPLES:

```
sage: B = species.BinaryTreeSpecies()
sage: B.algebraic_equation_system()
[-node3^2 + node1, -node1 + node3 - z]

sage: B.digraph().vertices()
[Combinatorial species,
 Product of (Combinatorial species) and (Combinatorial species),
 Singleton species,
 Sum of (Singleton species) and (Product of (Combinatorial species) and (Combinatorial speci

sage: B.algebraic_equation_system()[0].parent()
Multivariate Polynomial Ring in node0, node1, node2, node3 over Fraction Field of Univariate
```

composition(*g*)

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: S(S)
Composition of (Set species) and (Set species)
```

cycle_index_series()

Returns the cycle index series for this species.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: g = P.cycle_index_series()
sage: g.coefficients(4)
[p[], p[1], p[1, 1] + p[2], p[1, 1, 1] + p[2, 1] + p[3]]
```

digraph()

Returns a directed graph where the vertices are the individual species that make up this one.

EXAMPLES:

```
sage: X = species.SingletonSpecies()
sage: B = species.CombinatorialSpecies()
sage: B.define(X+B*B)
sage: g = B.digraph(); g
Multi-digraph on 4 vertices

sage: g, labels = g.canonical_label(certify=True)
sage: list(sorted(labels.items()))
[(Combinatorial species, 0),
 (Product of (Combinatorial species) and (Combinatorial species), 2),
 (Singleton species, 1),
 (Sum of (Singleton species) and (Product of (Combinatorial species) and (Combinatorial spec
sage: g.edges()
[(0, 3, None), (2, 0, None), (2, 0, None), (3, 1, None), (3, 2, None)]
```

functorial_composition(*g*)

Returns the functorial composition of self with g.

EXAMPLES:

```
sage: E = species.SetSpecies()
sage: E2 = E.restricted(min=2, max=3)
sage: WP = species.SubsetSpecies()
sage: P2 = E2*E
sage: G = WP.functorial_composition(P2)
```

```
sage: G.isotype_generating_series().coefficients(5)
[1, 1, 2, 4, 11]
```

generating_series()

Returns the generating series for this species. This is an exponential generating series so the nth coefficient of the series corresponds to the number of labeled structures with n labels divided by n!.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: g = P.generating_series()
sage: g.coefficients(4)
[1, 1, 1, 1]
sage: g.counts(4)
[1, 1, 2, 6]
sage: P.structures([1,2,3]).list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
sage: len(_)
6
```

is_weighted()

Returns True if this species has a nontrivial weighting associated with it.

EXAMPLES:

```
sage: C = species.CycleSpecies()
sage: C.is_weighted()
False
```

isotype_generating_series()

Returns the isotype generating series for this species. The nth coefficient of this series corresponds to the number of isomorphism types for the structures on n labels.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: g = P.isotype_generating_series()
sage: g.coefficients(4)
[1, 1, 2, 3]
sage: g.counts(4)
[1, 1, 2, 3]
sage: P.isotypes([1,2,3]).list()
[[2, 3, 1], [2, 1, 3], [1, 2, 3]]
sage: len(_)
3
```

isotypes(*labels, structure_class=None*)

EXAMPLES:

```
sage: F = CombinatorialSpecies()
sage: F.isotypes([1,2,3]).list()
...
NotImplementedError
```

product(*g*)

Returns the product of self and g.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: F = P * P; F
Product of (Permutation species) and (Permutation species)
```

restricted(*min=None, max=None*)

EXAMPLES:

```
sage: S = species.SetSpecies().restricted(min=3); S
Set species with min=3
sage: S.structures([1,2]).list()
[]
sage: S.generating_series().coefficients(5)
[0, 0, 0, 1/6, 1/24]
```

structures(*labels, structure_class=None*)

EXAMPLES:

```
sage: F = CombinatorialSpecies()
sage: F.structures([1,2,3]).list()
...
NotImplementedError
```

sum(*g*)

Returns the sum of self and g.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: F = P + P; F
Sum of (Permutation species) and (Permutation species)
sage: F.structures([1,2]).list()
[[1, 2], [2, 1], [1, 2], [2, 1]]
```

weight_ring()

Returns the ring in which the weights of this species occur.

By default, this is just the field of rational numbers.

EXAMPLES:

```
sage: species.SetSpecies().weight_ring()
Rational Field
```

weighted(*weight*)

Returns a version of this species with the specified weight.

EXAMPLES:

```
sage: t = ZZ['t'].gen()
sage: C = species.CycleSpecies(); C
Cyclic permutation species
sage: C.weighted(t)
Cyclic permutation species with weight=t
```

Recursive Species

class **CombinatorialSpecies**()

define(*x*)

Defines self to be equal to the combinatorial species x. This is used to define combinatorial species recursively. All of the real work is done by calling the .set() method for each of the series associated to self.

EXAMPLES: The species of linear orders L can be recursively defined by $L = 1 + X * L$ where 1 represents the empty set species and X represents the singleton species.

```
sage: X = species.SingletonSpecies()
sage: E = species.EmptySetSpecies()
sage: L = CombinatorialSpecies()
sage: L.define(E+X*L)
sage: L.generating_series().coefficients(4)
[1, 1, 1, 1]
sage: L.structures([1,2,3]).cardinality()
6
sage: L.structures([1,2,3]).list()
[1*(2*(3*{})),
 1*(3*(2*{})),
 2*(1*(3*{})),
 2*(3*(1*{})),
 3*(1*(2*{})),
 3*(2*(1*{}))]

sage: L = species.LinearOrderSpecies()
sage: L.generating_series().coefficients(4)
[1, 1, 1, 1]
sage: L.structures([1,2,3]).cardinality()
6
sage: L.structures([1,2,3]).list()
[[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
```

TESTS:

```
sage: A = CombinatorialSpecies()
sage: A.define(E+X*A*A)
sage: A.generating_series().coefficients(6)
[1, 1, 2, 5, 14, 42]
sage: A.generating_series().counts(6)
[1, 1, 4, 30, 336, 5040]
sage: len(A.structures([1,2,3,4]).list())
336
sage: A.isotype_generating_series().coefficients(6)
[1, 1, 2, 5, 14, 42]
sage: len(A.isotypes([1,2,3,4]).list())
14

sage: A = CombinatorialSpecies()
sage: A.define(X+A*A)
sage: A.generating_series().coefficients(6)
[0, 1, 1, 2, 5, 14]
sage: A.generating_series().counts(6)
[0, 1, 2, 12, 120, 1680]
sage: len(A.structures([1,2,3]).list())
12
sage: A.isotype_generating_series().coefficients(6)
[0, 1, 1, 2, 5, 14]
sage: len(A.isotypes([1,2,3,4]).list())
5

sage: X2 = X*X
sage: X5 = X2*X2*X
sage: A = CombinatorialSpecies()
sage: B = CombinatorialSpecies()
sage: C = CombinatorialSpecies()
sage: A.define(X5+B*B)
sage: B.define(X5+C*C)
```

```
sage: C.define(X2+C*C+A*A)
sage: A.generating_series().coefficients(Integer(10))
[0, 0, 0, 0, 0, 1, 0, 0, 1, 2]
sage: A.generating_series().coefficients(15)
[0, 0, 0, 0, 0, 1, 0, 0, 1, 2, 5, 4, 14, 10, 48]
sage: B.generating_series().coefficients(15)
[0, 0, 0, 0, 1, 1, 2, 0, 5, 0, 14, 0, 44, 0, 138]
sage: C.generating_series().coefficients(15)
[0, 0, 1, 0, 1, 0, 2, 0, 5, 0, 15, 0, 44, 2, 142]

sage: F = CombinatorialSpecies()
sage: F.define(E+X+(X*F+X*X*F))
sage: F.generating_series().counts(10)
[1, 2, 6, 30, 192, 1560, 15120, 171360, 2217600, 32296320]
sage: F.generating_series().coefficients(10)
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
sage: F.isotype_generating_series().coefficients(10)
[1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

weight_ring()

EXAMPLES:

```
sage: F = species.CombinatorialSpecies()
sage: F.weight_ring()
Rational Field

sage: X = species.SingletonSpecies()
sage: E = species.EmptySetSpecies()
sage: L = CombinatorialSpecies()
sage: L.define(E+X*L)
sage: L.weight_ring()
Rational Field
```

class **CombinatorialSpeciesStructure**(*parent, s, **options*)

Characteristic Species

class **CharacteristicSpeciesStructure**(*parent, labels, list*)

automorphism_group()

Returns the group of permutations whose action on this structure leave it fixed. For the characteristic species, there is only one structure, so every permutation is in its automorphism group.

EXAMPLES:

```
sage: F = species.CharacteristicSpecies(3)
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.automorphism_group()
Symmetric group of order 3! as a permutation group
```

canonical_label()

EXAMPLES:

```
sage: F = species.CharacteristicSpecies(3)
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.canonical_label()
```

```
{'a', 'b', 'c'}
```

transport(*perm*)

Returns the transport of this structure along the permutation perm.

EXAMPLES:

```
sage: F = species.CharacteristicSpecies(3)
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: p = PermutationGroupElement((1,2))
sage: a.transport(p)
{'a', 'b', 'c'}
```

class CharacteristicSpecies_class(*n, min=None, max=None, weight=None*)

class EmptySetSpecies_class(*min=None, max=None, weight=None*)

class SingletonSpecies_class(*min=None, max=None, weight=None*)

Cycle Species

CycleSpecies(**args, **kwds*)

Returns the species of cycles.

EXAMPLES:

```
sage: C = species.CycleSpecies(); C
Cyclic permutation species
sage: C.structures([1,2,3,4]).list()
[(1, 2, 3, 4),
 (1, 2, 4, 3),
 (1, 3, 2, 4),
 (1, 3, 4, 2),
 (1, 4, 2, 3),
 (1, 4, 3, 2)]
```

TESTS: We check to verify that the caching of species is actually working.

```
sage: species.CycleSpecies() is species.CycleSpecies()
True
```

class CycleSpeciesStructure(*parent, labels, list*)

automorphism_group()

Returns the group of permutations whose action on this structure leave it fixed.

EXAMPLES:

```
sage: P = species.CycleSpecies()
sage: a = P.structures([1, 2, 3, 4]).random_element(); a
(1, 2, 3, 4)
sage: a.automorphism_group()
Permutation Group with generators [(1,2,3,4)]

sage: [a.transport(perm) for perm in a.automorphism_group()]
[(1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4), (1, 2, 3, 4)]
```

canonical_label()

EXAMPLES:

```
sage: P = species.CycleSpecies()
sage: P.structures(["a","b","c"]).random_element().canonical_label()
('a', 'b', 'c')
```

permutation_group_element()
Returns this cycle as a permutation group element.
EXAMPLES:

```
sage: F = species.CycleSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
('a', 'b', 'c')
sage: a.permutation_group_element()
(1,2,3)
```

transport(*perm*)
Returns the transport of this structure along the permutation perm.
EXAMPLES:

```
sage: F = species.CycleSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
('a', 'b', 'c')
sage: p = PermutationGroupElement((1,2))
sage: a.transport(p)
('a', 'c', 'b')
```

class CycleSpecies_class(*min=None, max=None, weight=None*)

Partition Species

PartitionSpecies(**args, **kwds*)
Returns the species of partitions.
EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: P.generating_series().coefficients(5)
[1, 1, 1, 5/6, 5/8]
sage: P.isotype_generating_series().coefficients(5)
[1, 1, 2, 3, 5]
```

class PartitionSpeciesStructure(*parent, labels, list*)

automorphism_group()
Returns the group of permutations whose action on this set partition leave it fixed.
EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: from sage.combinat.species.partition_species import PartitionSpeciesStructure
sage: a = PartitionSpeciesStructure(None, [2,3,4], [[1,2],[3]]); a
{{2, 3}, {4}}
sage: a.automorphism_group()
Permutation Group with generators [(1,2)]
```

canonical_label()
EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.canonical_label() for s in S]
[{{'a', 'b', 'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a', 'b'}, {'c'}},
 {{'a'}, {'b'}, {'c'}}]
```

change_labels(*labels*)

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: from sage.combinat.species.partition_species import PartitionSpeciesStructure
sage: a = PartitionSpeciesStructure(None, [2,3,4], [[1,2],[3]]); a
{{2, 3}, {4}}
sage: a.change_labels([1,2,3])
{{1, 2}, {3}}
```

transport(*perm*)

Returns the transport of this set partition along the permutation perm. For set partitions, this is the direct product of the automorphism groups for each of the blocks.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: from sage.combinat.species.partition_species import PartitionSpeciesStructure
sage: a = PartitionSpeciesStructure(None, [2,3,4], [[1,2],[3]]); a
{{2, 3}, {4}}
sage: a.transport(p)
{{2, 4}, {3}}
```

class **PartitionSpecies_class**(*min=None, max=None, weight=None*)

Permutation species

PermutationSpecies(**args, **kwds*)

Returns the species of permutations.

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: P.generating_series().coefficients(5)
[1, 1, 1, 1, 1]
sage: P.isotype_generating_series().coefficients(5)
[1, 1, 2, 3, 5]
```

class **PermutationSpeciesStructure**(*parent, labels, list*)

automorphism_group()

Returns the group of permutations whose action on this structure leave it fixed.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3,4))
sage: P = species.PermutationSpecies()
sage: a = P.structures(["a", "b", "c", "d"]).random_element(); a
['a', 'c', 'b', 'd']
```

```
sage: a.automorphism_group()
Permutation Group with generators [(2,3), (1,4)]

sage: [a.transport(perm) for perm in a.automorphism_group()]
[['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd'],
 ['a', 'c', 'b', 'd']]
```

canonical_label()

EXAMPLES:

```
sage: P = species.PermutationSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.canonical_label() for s in S]
[['a', 'b', 'c'],
 ['b', 'a', 'c'],
 ['b', 'a', 'c'],
 ['b', 'c', 'a'],
 ['b', 'c', 'a'],
 ['b', 'a', 'c']]
```

permutation_group_element()

Returns self as a permutation group element.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3,4))
sage: P = species.PermutationSpecies()
sage: a = P.structures(["a", "b", "c", "d"]).random_element(); a
['a', 'c', 'b', 'd']
sage: a.permutation_group_element()
(2,3)
```

transport(*perm*)

Returns the transport of this structure along the permutation perm.

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3,4))
sage: P = species.PermutationSpecies()
sage: a = P.structures(["a", "b", "c", "d"]).random_element(); a
['a', 'c', 'b', 'd']
sage: a.transport(p)
['a', 'd', 'c', 'b']
```

class **PermutationSpecies_class**(*min=None, max=None, weight=None*)

Linear-order Species

LinearOrderSpecies(**args, **kwds*)

Returns the species of linear orders.

EXAMPLES:

```
sage: L = species.LinearOrderSpecies()
sage: L.generating_series().coefficients(5)
[1, 1, 1, 1, 1]
```

class **LinearOrderSpeciesStructure**(*parent, labels, list*)

automorphism_group()
Returns the group of permutations whose action on this structure leave it fixed. For the species of linear orders, there is no non-trivial automorphism.

EXAMPLES:

```
sage: F = species.LinearOrderSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
['a', 'b', 'c']
sage: a.automorphism_group()
Symmetric group of order 1! as a permutation group
```

canonical_label()
EXAMPLES:

```
sage: P = species.LinearOrderSpecies()
sage: s = P.structures(["a", "b", "c"]).random_element()
sage: s.canonical_label()
['a', 'b', 'c']
```

transport(*perm*)
Returns the transport of this structure along the permutation perm.

EXAMPLES:

```
sage: F = species.LinearOrderSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
['a', 'b', 'c']
sage: p = PermutationGroupElement((1,2))
sage: a.transport(p)
['b', 'a', 'c']
```

class **LinearOrderSpecies_class**(*min=None, max=None, weight=None*)

Set Species

SetSpecies(**args, **kwds*)
Returns the species of sets.

EXAMPLES:

```
sage: E = species.SetSpecies()
sage: E.structures([1,2,3]).list()
[{1, 2, 3}]
sage: E.isotype_generating_series().coefficients(4)
[1, 1, 1, 1]
```

class **SetSpeciesStructure**(*parent, labels, list*)

automorphism_group()
Returns the group of permutations whose action on this set leave it fixed. For the species of sets, there is only one isomorphism class, so every permutation is in its automorphism group.

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.automorphism_group()
Symmetric group of order 3! as a permutation group
```

canonical_label()
EXAMPLES:

```
sage: S = species.SetSpecies()
sage: a = S.structures(["a","b","c"]).random_element(); a
{'a', 'b', 'c'}
sage: a.canonical_label()
{'a', 'b', 'c'}
```

transport(*perm*)
Returns the transport of this set along the permutation perm.
EXAMPLES:

```
sage: F = species.SetSpecies()
sage: a = F.structures(["a", "b", "c"]).random_element(); a
{'a', 'b', 'c'}
sage: p = PermutationGroupElement((1,2))
sage: a.transport(p)
{'a', 'b', 'c'}
```

class **SetSpecies_class**(*min=None, max=None, weight=None*)

Subset Species

SubsetSpecies(**args, **kwds*)
Returns the species of subsets.
EXAMPLES:

```
sage: S = species.SubsetSpecies()
sage: S.generating_series().coefficients(5)
[1, 2, 2, 4/3, 2/3]
sage: S.isotype_generating_series().coefficients(5)
[1, 2, 3, 4, 5]
```

class **SubsetSpeciesStructure**(*parent, labels, list*)

automorphism_group()
Returns the group of permutations whose action on this subset leave it fixed.
EXAMPLES:

```
sage: F = species.SubsetSpecies()
sage: a = F.structures([1,2,3,4])[6]; a
{1, 3}
sage: a.automorphism_group()
Permutation Group with generators [(2,4), (1,3)]

sage: [a.transport(g) for g in a.automorphism_group()]
[{1, 3}, {1, 3}, {1, 3}, {1, 3}]
```

canonical_label()
EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.canonical_label() for s in S]
[{}, {'a'}, {'a'}, {'a'}, {'a', 'b'}, {'a', 'b'}, {'a', 'b'}, {'a', 'b', 'c'}]
```

complement()
EXAMPLES:

```
sage: F = species.SubsetSpecies()
sage: a = F.structures(["a", "b", "c"])[5]; a
{'a', 'c'}
sage: a.complement()
{'b'}
```

labels()
EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.labels() for s in S]
[[], ['a'], ['b'], ['c'], ['a', 'b'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c']]
```

transport(*perm*)
Returns the transport of this subset along the permutation perm.
EXAMPLES:

```
sage: F = species.SubsetSpecies()
sage: a = F.structures(["a", "b", "c"])[5]; a
{'a', 'c'}
sage: p = PermutationGroupElement((1,2))
sage: a.transport(p)
{'b', 'c'}
sage: p = PermutationGroupElement((1,3))
sage: a.transport(p)
{'a', 'c'}
```

class **SubsetSpecies_class**(*min=None, max=None, weight=None*)

Examples of Combinatorial Species

17.38.3 Operations on Species

Sum species

class **SumSpeciesStructure**(*parent, s, **options*)

class **SumSpecies_class**(*F, G, min=None, max=None, weight=None*)

weight_ring()
Returns the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you add elements of the weight rings for each of the operands.
EXAMPLES:

```
sage: S = species.SetSpecies()
sage: C = S+S
sage: C.weight_ring()
Rational Field

sage: S = species.SetSpecies(weight=QQ['t'].gen())
sage: C = S + S
sage: C.weight_ring()
Univariate Polynomial Ring in t over Rational Field
```

Sum species

class ProductSpeciesStructure(*parent, labels, subset, left, right*)

automorphism_group()
EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: S = species.SetSpecies()
sage: F = S * S
sage: a = F.structures([1,2,3,4]).random_element(); a
{1}*{2, 3, 4}
sage: a.automorphism_group()
Permutation Group with generators [(2,3), (2,3,4)]

sage: [a.transport(g) for g in a.automorphism_group()]
[{1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4},
 {1}*{2, 3, 4}]

sage: a = F.structures([1,2,3,4]).random_element(); a
{2, 3}*{1, 4}
sage: [a.transport(g) for g in a.automorphism_group()]
[{2, 3}*{1, 4}, {2, 3}*{1, 4}, {2, 3}*{1, 4}, {2, 3}*{1, 4}]
```

canonical_label()
EXAMPLES:

```
sage: S = species.SetSpecies()
sage: F = S * S
sage: S = F.structures(['a','b','c']).list(); S
[{}*{'a', 'b', 'c'},
 {'a'}*{'b', 'c'},
 {'b'}*{'a', 'c'},
 {'c'}*{'a', 'b'},
 {'a', 'b'}*{'c'},
 {'a', 'c'}*{'b'},
 {'b', 'c'}*{'a'},
 {'a', 'b', 'c'}*{}]

sage: F.isotypes(['a','b','c']).cardinality()
4
sage: [s.canonical_label() for s in S]
[{}*{'a', 'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a'}*{'b', 'c'},
 {'a', 'b'}*{'c'},
 {'a', 'b'}*{'c'},
 {'a', 'b'}*{'c'},
 {'a', 'b', 'c'}*{}]
```

change_labels(*labels*)
EXAMPLES:

```
sage: S = species.SetSpecies()
sage: F = S * S
sage: a = F.structures(['a','b','c']).random_element(); a
{}*{'a', 'b', 'c'}
sage: a.change_labels([1,2,3])
{}*{1, 2, 3}
```

transport(*perm*)

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: S = species.SetSpecies()
sage: F = S * S
sage: a = F.structures(['a','b','c'])[4]; a
{'a', 'b'}*{'c'}
sage: a.transport(p)
{'a', 'c'}*{'b'}
```

class **ProductSpecies_class**(*F, G, min=None, max=None, weight=None*)

weight_ring()

Returns the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you multiply (and add) elements of the weight rings for each of the operands.

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: C = S*S
sage: C.weight_ring()
Rational Field

sage: S = species.SetSpecies(weight=QQ['t'].gen())
sage: C = S*S
sage: C.weight_ring()
Univariate Polynomial Ring in t over Rational Field

sage: S = species.SetSpecies()
sage: C = (S*S).weighted(QQ['t'].gen())
sage: C.weight_ring()
Univariate Polynomial Ring in t over Rational Field
```

Composition species

class **CompositionSpeciesStructure**(*parent, labels, pi, f, gs*)

change_labels(*labels*)

EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: L = E(C)
sage: S = L.structures(['a','b','c']).list()
sage: a = S[2]; a
F-structure: {{'a', 'c'}, {'b'}}; G-structures: [('a', 'c'), ('b')]
sage: a.change_labels([1,2,3])
F-structure: {{1, 3}, {2}}; G-structures: [(1, 3), (2)]
```

transport(*perm*)
EXAMPLES:

```
sage: p = PermutationGroupElement((2,3))
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: L = E(C)
sage: S = L.structures(['a','b','c']).list()
sage: a = S[2]; a
F-structure: {{'a', 'c'}, {'b'}}; G-structures: [('a', 'c'), ('b')]
sage: a.transport(p)
F-structure: {{'a', 'b'}, {'c'}}; G-structures: [('a', 'c'), ('b')]
```

class CompositionSpecies_class(*F, G, min=None, max=None, weight=None*)

weight_ring()
Returns the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you multiply (and add) elements of the weight rings for each of the operands.
EXAMPLES:

```
sage: E = species.SetSpecies(); C = species.CycleSpecies()
sage: L = E(C)
sage: L.weight_ring()
Rational Field
```

Functorial composition species

class FunctorialCompositionSpecies_class(*F, G, min=None, max=None, weight=None*)

weight_ring()
Returns the weight ring for this species. This is determined by asking Sage's coercion model what the result is when you multiply (and add) lements of the weight rings for each of the operands.
EXAMPLES:

```
sage: G = species.SimpleGraphSpecies()
sage: G.weight_ring()
Rational Field
```

class FunctorialCompositionStructure(*parent, labels, list*)

17.38.4 Miscellaneous

Species structures

We will illustrate the use of the structure classes using the "balls and bars" model for integer compositions. An integer composition of 6 such as [2, 1, 3] can be represented in this model as 'oooooo' where the 6 o's correspond to the balls and the 2 's correspond to the bars. If BB is our species for this model, the it satisfies the following recursive definition:

BB = o + o*BB + o*|*BB

Here we define this species using the default structures:

```
sage: ball = species.SingletonSpecies(); o = var('o')
sage: bar = species.EmptySetSpecies()
sage: BB = CombinatorialSpecies()
```

```
sage: BB.define(ball + ball*BB + ball*bar*BB)
sage: BB.isotypes([o]*3).list()
[o*(o*o), o*((o*{})*o), (o*{})*(o*o), (o*{})*((o*{})*o)]
```

If we ignore the parentheses, we can read off that the integer compositions are [3], [2, 1], [1, 2], and [1, 1, 1].

class GenericSpeciesStructure(*parent, labels, list*)

change_labels(*labels*)

EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.change_labels([1,2,3]) for s in S]
[{}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}]
```

is_isomorphic(*x*)

EXAMPLES:

```
sage: S = species.SetSpecies()
sage: a = S.structures([1,2,3]).random_element(); a
{1, 2, 3}
sage: b = S.structures(['a','b','c']).random_element(); b
{'a', 'b', 'c'}
sage: a.is_isomorphic(b)
True
```

labels()

EXAMPLES:

```
sage: P = species.SubsetSpecies()
sage: S = P.structures(["a", "b", "c"])
sage: [s.labels() for s in S]
[[], ['a'], ['b'], ['c'], ['a', 'b'], ['a', 'c'], ['b', 'c'], ['a', 'b', 'c']]
```

class IsotypesWrapper(*species, labels, structure_class*)

class SimpleIsotypesWrapper(*species, labels, structure_class*)

class SimpleStructuresWrapper(*species, labels, structure_class*)

class SpeciesStructure()

parent()

Returns the species that this structure is associated with.

EXAMPLES:

```
sage: L = species.LinearOrderSpecies()
sage: a,b = L.structures([1,2])
sage: a.parent()
Linear order species
```

class SpeciesStructureWrapper(*parent, s, **options*)

canonical_label()

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: s = (P+P).structures([1,2,3]).random_element(); s
```

```
{{1, 3}, {2}}
sage: s.canonical_label()
{{1, 2}, {3}}
```

labels()

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: s = (P+P).structures([1,2,3]).random_element(); s
{{1, 3}, {2}}
sage: s.labels()
[{1, 3}, {2}]
sage: type(_)
<type 'list'>
```

transport(*perm*)

EXAMPLES:

```
sage: P = species.PartitionSpecies()
sage: s = (P+P).structures([1,2,3]).random_element(); s
{{1, 3}, {2}}
sage: s.transport(PermutationGroupElement((2,3)))
{{1, 2}, {3}}
```

class **SpeciesWrapper**(*species, labels, iterator, generating_series, name, structure_class*)

cardinality()

EXAMPLES:

```
sage: F = species.SetSpecies()
sage: F.structures([1,2,3]).cardinality()
1
```

class **StructuresWrapper**(*species, labels, structure_class*)

Miscellaneous Functions

accept_size(*f*)

The purpose of this decorator is to change calls like species.SetSpecies(size=1) to species.SetSpecies(min=1, max=2). This is to make caching species easier and to restrict the number of parameters that the lower level code needs to know about.

EXAMPLES:

```
sage: from sage.combinat.species.misc import accept_size
sage: def f(*args, **kwds):
...       print args, list(sorted(kwds.items()))
sage: f = accept_size(f)
sage: f(min=1)
() [('min', 1)]
sage: f(size=2)
() [('max', 3), ('min', 2)]
```

change_support(*perm, support, change_perm=None*)

Changes the support of a permutation defined on [1, ..., n] to support.

EXAMPLES:

```
sage: from sage.combinat.species.misc import change_support
sage: p = PermutationGroupElement((1,2,3)); p
(1,2,3)
sage: change_support(p, [3,4,5])
(3,4,5)
```

17.39 Developer Tools

17.39.1 Backtracking

This library contains generic tools for constructing large sets whose elements can be enumerated by exploring a search space with a (lazy) tree or graph structure.

- `SearchForest`: Depth first search through a tree described by a `children` function.
- `GenericBacktracker`: Depth first search through a tree described by a `children` function, with branch pruning, etc.
- `TransitiveIdeal`: Depth first search through a graph described by a `neighbours` relation.
- `TransitiveIdealGraded`: Breath first search through a graph described by a `neighbours` relation.

TODO:

1. Find a good and consistent naming scheme! Do we want to emphasize the underlying graph/tree structure? The branch & bound aspect? The transitive closure of a relation point of view?
2. Do we want `TransitiveIdeal(relation, generators)` or `TransitiveIdeal(generators, relation)`? The code needs to be standardized once the choice is made.

class `GenericBacktracker`(*initial_data, initial_state*)

A generic backtrack tool for exploring a search space organized as a tree, with branch pruning, etc.

See also `SearchForest` and `TransitiveIdeal` for handling simple special cases.

class `SearchForest`(*roots, children*)

Returns the set of nodes of the forest having the given roots, and where `children(x)` returns the children of the node `x` of the forest.

See also `GenericBacktracker`, `TransitiveIdeal`, and `TransitiveIdealGraded`.

INPUT:

- `roots`: a list (or iterable)
- `children`: a function returning a list (or iterable)

EXAMPLES:

A generator object for binary sequencences of length 3, listed:

```
sage: list(SearchForest([[]], lambda l: [l+[0], l+[1]] if len(l) < 3 else []))
[[], [0], [0, 0], [0, 0, 0], [0, 0, 1], [0, 1], [0, 1, 0], [0, 1, 1], [1], [1, 0], [1, 0, 0], [1
```

A generator object for ordered sequences of length 2 from a 4-set, sampled:

```
sage: tb = SearchForest([[]], lambda l: [l + [i] for i in range(4) if i not in l] if len(l) < 2
sage: tb[0]
[]
sage: tb[16]
[3, 2]
```

class `TransitiveIdeal`(*succ, generators*)

Generic tool for constructing ideals of a relation.

INPUT:

- `relation`: a function (or callable) returning a list (or iterable)
- `generators`: a list (or iterable)

Returns the set S of elements that can be obtained by repeated application of `relation` on the elements of `generators`.

Consider `relation` as modeling a directed graph (possibly with loops, cycles, or circuits). Then S is the ideal generated by `generators` under this relation.

Enumerating the elements of S is achieved by depth first search through the graph. The time complexity is $O(n + m)$ where n is the size of the ideal, and m the number of edges in the relation. The memory complexity is the depth, that is the maximal distance between a generator and an element of S.

See also `SearchForest` and `TransitiveIdealGraded`.

EXAMPLES:

```
sage: [i for i in TransitiveIdeal(lambda i: [i+1] if i<10 else [], [0])]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

sage: [i for i in TransitiveIdeal(lambda i: [mod(i+1,3)], [0])]
[0, 1, 2]
sage: [i for i in TransitiveIdeal(lambda i: [mod(i+2,3)], [0])]
[0, 2, 1]
sage: [i for i in TransitiveIdeal(lambda i: [mod(i+2,10)], [0])]
[0, 2, 4, 6, 8]
sage: [i for i in TransitiveIdeal(lambda i: [mod(i+3,10),mod(i+5,10)], [0])]
[0, 3, 8, 1, 4, 5, 6, 7, 9, 2]
sage: [i for i in TransitiveIdeal(lambda i: [mod(i+4,10),mod(i+6,10)], [0])]
[0, 4, 8, 2, 6]
sage: [i for i in TransitiveIdeal(lambda i: [mod(i+3,9)], [0,1])]
[0, 1, 3, 4, 6, 7]

sage: [p for p in TransitiveIdeal(lambda x:[x],[Permutation([3,1,2,4]), Permutation([2,1,3,4])])]
[[2, 1, 3, 4], [3, 1, 2, 4]]
```

We now illustrate that the enumeration is done lazily, by depth first search:

```
sage: C = TransitiveIdeal(lambda x: [x-1, x+1], (-10, 0, 10))
sage: f = C.__iter__()
sage: [ f.next() for i in range(6) ]
[0, 1, 2, 3, 4, 5]
```

We compute all the permutations of 3:

```
sage: [p for p in TransitiveIdeal(attrcall("permutohedron_succ"), [Permutation([1,2,3])])]
[[1, 2, 3], [2, 1, 3], [1, 3, 2], [2, 3, 1], [3, 2, 1], [3, 1, 2]]
```

We compute all the permutations which are larger than [3,1,2,4], [2,1,3,4] in the right permutohedron:

```
sage: [p for p in TransitiveIdeal(attrcall("permutohedron_succ"), [Permutation([3,1,2,4]), Permu
[[2, 1, 3, 4], [2, 1, 4, 3], [2, 4, 1, 3], [4, 2, 1, 3], [4, 2, 3, 1], [4, 3, 2, 1], [3, 1, 2, 4
```

class TransitiveIdealGraded(*succ, generators*)

Generic tool for constructing ideals of a relation.

INPUT:

- `relation`: a function (or callable) returning a list (or iterable)
- `generators`: a list (or iterable)

Returns the set S of elements that can be obtained by repeated application of `relation` on the elements of `generators`.

Consider `relation` as modeling a directed graph (possibly with loops, cycles, or circuits). Then S is the ideal generated by `generators` under this relation.

Enumerating the elements of S is achieved by breath first search through the graph; hence elements are enumerated by increasing distance from the generators. The time complexity is $O(n + m)$ where n is the size of the ideal, and m the number of edges in the relation. The memory complexity is the depth, that is the maximal distance between a generator and an element of S.

See also `SearchForest` and `TransitiveIdeal`.

EXAMPLES:

```
sage: [i for i in TransitiveIdealGraded(lambda i: [i+1] if i<10 else [], [0])]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

We now illustrates that the enumeration is done lazily, by breath first search:

```
sage: C = TransitiveIdealGraded(lambda x: [x-1, x+1], (-10, 0, 10))
sage: f = C.__iter__()
```

The elements at distance 0 from the generators:

```
sage: sorted([ f.next() for i in range(3) ])
[-10, 0, 10]
```

The elements at distance 1 from the generators:

```
sage: sorted([ f.next() for i in range(6) ])
[-11, -9, -1, 1, 9, 11]
```

The elements at distance 2 from the generators:

```
sage: sorted([ f.next() for i in range(6) ])
[-12, -8, -2, 2, 8, 12]
```

The enumeration order between elements at the same distance is not specified.

We compute all the permutations which are larger than [3,1,2,4] or [2,1,3,4] in the permutohedron:

```
sage: [p for p in TransitiveIdealGraded(attrcall("permutohedron_succ"), [Permutation([3,1,2,4]),
[[3, 1, 2, 4], [2, 1, 3, 4], [2, 1, 4, 3], [3, 2, 1, 4], [2, 3, 1, 4], [3, 1, 4, 2], [2, 3, 4, 1
```

search_forest_iterator(*roots, children*)

Returns an iterator on the nodes of the forest having the given roots, and where `children(x)` returns the children of the node `x` of the forest. Note that every node of the tree is returned, not simply the leaves.

INPUT:

- `roots`: a list (or iterable)
- `children`: a function returning a list (or iterable)

EXAMPLES:

Search tree where leaves are binary sequences of length 3:

```
sage: from sage.combinat.backtrack import search_forest_iterator
sage: list(search_forest_iterator([[]], lambda l: [l+[0], l+[1]] if len(l) < 3 else []))
[[], [0], [0, 0], [0, 0, 0], [0, 0, 1], [0, 1], [0, 1, 0], [0, 1, 1], [1], [1, 0], [1, 0, 0], [1
```

Search tree where leaves are ordered sequences of length 2 from a 4-set:

```
sage: from sage.combinat.backtrack import search_forest_iterator
sage: list(search_forest_iterator([[]], lambda l: [l + [i] for i in range(4) if i not in l] if l
[[], [0], [0, 1], [0, 2], [0, 3], [1], [1, 0], [1, 2], [1, 3], [2], [2, 0], [2, 1], [2, 3], [3],
```

17.39.2 Output functions

tex_from_array(*a*)

EXAMPLES:

```
sage: from sage.combinat.output import tex_from_array
sage: print tex_from_array([[1,2,3],[4,5]])
{\def\lr#1{\multicolumn{1}{|@{\hspace{.6ex}}c@{\hspace{.6ex}}|}{\raisebox{-.3ex}{$#1$}}}
\raisebox{-.6ex}{$\begin{array}[b]{ccc}
\cline{1-1}\cline{2-2}\cline{3-3}
\lr{1}&\lr{2}&\lr{3}\\
\cline{1-1}\cline{2-2}\cline{3-3}
\lr{4}&\lr{5}\\
\cline{1-1}\cline{2-2}
\end{array}$}
}
```

17.39.3 Low-level permutations

class PermutationsNK(*n, k*)

cardinality()

Returns the number of permutations of k things from a list of n things.

EXAMPLES:

```
sage: from sage.combinat.permutation_nk import PermutationsNK
sage: PermutationsNK(3,2).cardinality()
6
sage: PermutationsNK(5,4).cardinality()
120
```

random_element()

Returns a random permutation of k things from range(n).

EXAMPLES:

```
sage: from sage.combinat.permutation_nk import PermutationsNK
sage: PermutationsNK(3,2).random_element()
[0, 1]
```

17.39.4 Low-level splits

SplitNK(*n*, *k*)

Returns the combinatorial class of splits of a the set range(n) into a set of size k and a set of size n-k.

EXAMPLES:

```
sage: from sage.combinat.split_nk import SplitNK
sage: S = SplitNK(5,2); S
Splits of {0, ..., 4} into a set of size 2 and one of size 3
sage: S.first()
[[0, 1], [2, 3, 4]]
sage: S.last()
[[3, 4], [0, 1, 2]]
sage: S.list()
[[[0, 1], [2, 3, 4]],
 [[0, 2], [1, 3, 4]],
 [[0, 3], [1, 2, 4]],
 [[0, 4], [1, 2, 3]],
 [[1, 2], [0, 3, 4]],
 [[1, 3], [0, 2, 4]],
 [[1, 4], [0, 2, 3]],
 [[2, 3], [0, 1, 4]],
 [[2, 4], [0, 1, 3]],
 [[3, 4], [0, 1, 2]]]
```

class SplitNK_nk(*n*, *k*)

cardinality()

Returns the number of choices of set partitions of range(n) into a set of size k and a set of size n-k.

EXAMPLES:

```
sage: from sage.combinat.split_nk import SplitNK
sage: SplitNK(5,2).cardinality()
10
```

random_element()

Returns a random set partition of range(n) into a set of size k and a set of size n-k.

EXAMPLES:

```
sage: from sage.combinat.split_nk import SplitNK
sage: SplitNK(5,2).random_element()
[[0, 2], [1, 3, 4]]
```

17.39.5 Low-level Combinations

class ChooseNK(*n*, *k*)

Low level combinatorial class of all possible choices of k elements out of range(n) without repetitions.

This is a low-level combinatorial class, with a simplistic interface by design. It aim at speed. It's element are returned as plain list of python int.

EXAMPLES:

```
sage: from sage.combinat.choose_nk import ChooseNK
sage: c = ChooseNK(4,2)
sage: c.first()
[0, 1]
```

```
sage: c.list()
[[0, 1], [0, 2], [0, 3], [1, 2], [1, 3], [2, 3]]
sage: type(c.list()[1])
<type 'list'>
sage: type(c.list()[1][1])
<type 'int'>
```

cardinality()

Returns the number of choices of k things from a list of n things.

EXAMPLES:

```
sage: from sage.combinat.choose_nk import ChooseNK
sage: ChooseNK(3,2).cardinality()
3
sage: ChooseNK(5,2).cardinality()
10
```

random_element()

Returns a random choice of k things from range(n).

EXAMPLES:

```
sage: from sage.combinat.choose_nk import ChooseNK
sage: ChooseNK(5,2).random_element()
[0, 2]
```

rank(*x*)

EXAMPLES:

```
sage: from sage.combinat.choose_nk import ChooseNK
sage: c52 = ChooseNK(5,2)
sage: range(c52.cardinality()) == map(c52.rank, c52)
True
```

unrank(*r*)

EXAMPLES:

```
sage: from sage.combinat.choose_nk import ChooseNK
sage: c52 = ChooseNK(5,2)
sage: c52.list() == map(c52.unrank, range(c52.cardinality()))
True
```

from_rank(*r*, *n*, *k*)

Returns the combination of rank r in the subsets of range(n) of size k when listed in lexicographic order.

The algorithm used is based on combinadics and James McCaffrey's MSDN article. See: http://en.wikipedia.org/wiki/Combinadic

EXAMPLES:

```
sage: import sage.combinat.choose_nk as choose_nk
sage: choose_nk.from_rank(0,3,0)
[]
sage: choose_nk.from_rank(0,3,1)
[0]
sage: choose_nk.from_rank(1,3,1)
[1]
sage: choose_nk.from_rank(2,3,1)
[2]
sage: choose_nk.from_rank(0,3,2)
[0, 1]
```

```
sage: choose_nk.from_rank(1,3,2)
[0, 2]
sage: choose_nk.from_rank(2,3,2)
[1, 2]
sage: choose_nk.from_rank(0,3,3)
[0, 1, 2]
```

rank(*comb*, *n*)

Returns the rank of comb in the subsets of range(n) of size k.

The algorithm used is based on combinadics and James McCaffrey's MSDN article. See: http://en.wikipedia.org/wiki/Combinadic

EXAMPLES:

```
sage: import sage.combinat.choose_nk as choose_nk
sage: choose_nk.rank([], 3)
0
sage: choose_nk.rank([0], 3)
0
sage: choose_nk.rank([1], 3)
1
sage: choose_nk.rank([2], 3)
2
sage: choose_nk.rank([0,1], 3)
0
sage: choose_nk.rank([0,2], 3)
1
sage: choose_nk.rank([1,2], 3)
2
sage: choose_nk.rank([0,1,2], 3)
0
```

17.39.6 Low-level multichoose

class MultichooseNK(*n*, *k*)

cardinality()

Returns the number of multichoices of k things from a list of n things.

EXAMPLES:

```
sage: MultichooseNK(3,2).cardinality()
6
```

random_element()

Returns a random multichoice of k things from range(n).

EXAMPLES:

```
sage: MultichooseNK(5,2).random_element()
[0, 2]
sage: MultichooseNK(5,2).random_element()
[0, 1]
```

17.39.7 Tools

transitive_ideal (*f, x*)

Given an initial value x and a successor function f, return a list containing x and all of its successors. The successor function should return a list of all the successors of f.

Note that if x has an infinite number of successors, transitive_ideal won't return.

EXAMPLES:

```
sage: f = lambda x: [x-1] if x > 0 else []
sage: sage.combinat.tools.transitive_ideal(f, 10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

17.39.8 Generators

concat (*gens*)

Returns a generator that is the concatenation of the generators in the list.

EXAMPLES:

```
sage: list(sage.combinat.generator.concat([[1,2,3],[4,5,6]]))
[1, 2, 3, 4, 5, 6]
```

element (*element, n=1*)

Returns a generator that yield a single element n times.

EXAMPLES:

```
sage: list(sage.combinat.generator.element(1))
[1]
sage: list(sage.combinat.generator.element(1, n=3))
[1, 1, 1]
```

map (*f, gen*)

Returns a generator that returns f(g) for g in gen.

EXAMPLES:

```
sage: f = lambda x: x*2
sage: list(sage.combinat.generator.map(f,[4,5,6]))
[8, 10, 12]
```

select (*f, gen*)

Returns a generator for all the elements g of gen such that f(g) is True.

EXAMPLES:

```
sage: f = lambda x: x % 2 == 0
sage: list(sage.combinat.generator.select(f,range(7)))
[0, 2, 4, 6]
```

successor (*initial, succ*)

Given an initial value and a successor function, yield the initial value and each following successor. The generator will continue to generate values until the successor function yields None.

EXAMPLES:

```
sage: f = lambda x: x+1 if x < 10 else None
sage: list(sage.combinat.generator.successor(0,f))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

17.39.9 Rankers

from_list(*l*)

Returns a ranker from the list l.

INPUT:

- `l` - a list

OUTPUT:

- `[rank, unrank]` - functions

EXAMPLES:

```
sage: import sage.combinat.ranker as ranker
sage: l = [1,2,3]
sage: r,u = ranker.from_list(l)
sage: r(1)
0
sage: r(3)
2
sage: u(2)
3
sage: u(0)
1
```

on_fly()

Returns a pair of enumeration functions rank / unrank.

rank assigns on the fly an integer, starting from 0, to any object passed as argument. The object should be hashable. unrank is the inverse function; it returns None for indices that have not yet been assigned.

EXAMPLES:

```
sage: [rank, unrank] = sage.combinat.ranker.on_fly()
sage: rank('a')
0
sage: rank('b')
1
sage: rank('c')
2
sage: rank('a')
0
sage: unrank(2)
'c'
sage: unrank(3)
sage: rank('d')
3
sage: unrank(3)
'd'
```

TODO: add tests as in combinat::rankers

rank_from_list(*l*)
Returns a rank function given a list l.

EXAMPLES:

```
sage: import sage.combinat.ranker as ranker
sage: l = [1,2,3]
sage: r = ranker.rank_from_list(l)
sage: r(1)
0
sage: r(3)
2
```

unrank_from_list(*l*)
Returns an unrank function from a list.

EXAMPLES:

```
sage: import sage.combinat.ranker as ranker
sage: l = [1,2,3]
sage: u = ranker.unrank_from_list(l)
sage: u(2)
3
sage: u(0)
1
```

17.40 Words

17.40.1 Alphabets

Alphabet(*data=None, name=None*)
Returns an object representing an ordered alphabet.

EXAMPLES:

```
sage: Alphabet("ab")
Ordered Alphabet ['a', 'b']
sage: Alphabet([0,1,2])
Ordered Alphabet [0, 1, 2]
sage: Alphabet(name="positive integers")
Ordered Alphabet of Positive Integers
sage: Alphabet(name="PP")
Ordered Alphabet of Positive Integers
sage: Alphabet(name="natural numbers")
Ordered Alphabet of Natural Numbers
sage: Alphabet(name="NN")
Ordered Alphabet of Natural Numbers
```

OrderedAlphabet(*data=None, name=None*)
Returns an object representing an ordered alphabet.

EXAMPLES:

```
sage: Alphabet("ab")
Ordered Alphabet ['a', 'b']
sage: Alphabet([0,1,2])
Ordered Alphabet [0, 1, 2]
```

```
sage: Alphabet(name="positive integers")
Ordered Alphabet of Positive Integers
sage: Alphabet(name="PP")
Ordered Alphabet of Positive Integers
sage: Alphabet(name="natural numbers")
Ordered Alphabet of Natural Numbers
sage: Alphabet(name="NN")
Ordered Alphabet of Natural Numbers
```

class **OrderedAlphabet_Finite**(*alphabet*)

rank(*letter*)

Returns the index of letter in self.

INPUT:

- `letter` - a letter contained in this alphabet

OUTPUT:

- `integer` - the integer mapping for the letter

EXAMPLES:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_Finite
sage: OrderedAlphabet_Finite('abcd').rank('a')
0
sage: OrderedAlphabet_Finite('abcd').rank('d')
3
sage: OrderedAlphabet_Finite('abcd').rank('e')
...
IndexError: letter not in alphabet: 'e'
sage: OrderedAlphabet_Finite('abcd').rank('')
...
IndexError: letter not in alphabet: ''
```

string_rep()

Returns the string representation of the alphabet.

TESTS:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_Finite
sage: OrderedAlphabet_Finite('cba').string_rep()
"['c', 'b', 'a']"
sage: OrderedAlphabet_Finite([1, 3, 2]).string_rep()
'[1, 3, 2]'
```

unrank(*n*)

Returns the letter in position n of the alphabet self.

INPUT:

- `n` - a nonnegative integer

OUTPUT: the (n+1)-th object output by iter(self)

EXAMPLES:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_Finite
sage: OrderedAlphabet_Finite('abcd').unrank(0)
'a'
sage: OrderedAlphabet_Finite('abcd').unrank(3)
'd'
sage: OrderedAlphabet_Finite('abcd').unrank(5)
```

```
...
IndexError: list index out of range
```

class OrderedAlphabet_Infinite()

cardinality()

Return the number of elements in self.

OUTPUT: +Infinity

EXAMPLES:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_Infinite
sage: OrderedAlphabet_Infinite().cardinality()
+Infinity
```

list()

Returns NotImplementedError since we cannot list all the nonnegative integers.

TESTS:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_Infinite
sage: OrderedAlphabet_Infinite().list()
...
NotImplementedError
```

class OrderedAlphabet_NaturalNumbers()

The alphabet of nonnegative integers, ordered in the usual way.

TESTS:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_NaturalNumbers
sage: NN = OrderedAlphabet_NaturalNumbers()
sage: NN == loads(dumps(NN))
True
```

next(*n*)

Returns the letter following n in the alphabet self.

INPUT:

- n - nonnegative integer

OUTPUT: n+1

EXAMPLES:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_NaturalNumbers
sage: NN = OrderedAlphabet_NaturalNumbers()
sage: NN.next(0)
1
sage: NN.next(117)
118
sage: NN.next(-1)
...
ValueError: letter(=-1) not in the alphabet
```

rank(*letter*)

Returns the index of letter in self.

INPUT:

- `letter` - a letter contained in this alphabet

OUTPUT:

- `integer` - the integer mapping for the letter

EXAMPLES:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_NaturalNumbers
sage: NN = OrderedAlphabet_NaturalNumbers()
sage: NN.rank(0)
0
sage: NN.rank(17)
17
```

TESTS:

```
sage: NN.rank(-1)
...
ValueError: letter(=-1) not in the alphabet
sage: NN.rank("a")
...
ValueError: letter(=a) not in the alphabet
```

string_rep()

Returns the string representation of the alphabet.

TESTS:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_NaturalNumbers
sage: OrderedAlphabet_NaturalNumbers().string_rep()
'Natural Numbers'
```

unrank(*n*)

Returns the letter in position n in self, which in this case is n.

INPUT:

- n - nonnegative integer

OUTPUT:

- n - nonnegative integer

EXAMPLES:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_NaturalNumbers
sage: NN = OrderedAlphabet_NaturalNumbers()
sage: NN.unrank(0)
0
sage: NN.unrank(17)
17
```

TESTS:

```
sage: NN.unrank(-1)
...
ValueError: -1 is not a nonnegative integer
sage: NN.unrank("a")
...
ValueError: a is not a nonnegative integer
```

class **OrderedAlphabet_PositiveIntegers**()

The alphabet of nonnegative integers, ordered in the usual way.

TESTS:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_PositiveIntegers
sage: PP = OrderedAlphabet_PositiveIntegers()
sage: PP == loads(dumps(PP))
True
```

next(*n*)

Returns the letter following n in the alphabet self.

INPUT:

- n - positive integer

OUTPUT: n+1

EXAMPLES:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_PositiveIntegers
sage: PP = OrderedAlphabet_PositiveIntegers()
sage: PP.next(1)
2
sage: PP.next(117)
118
sage: PP.next(0)
...
ValueError: letter(=0) not in the alphabet
```

rank(*letter*)

Returns the index of letter in self.

INPUT:

- letter - a positive integer

OUTPUT: letter-1

EXAMPLES:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_PositiveIntegers
sage: OrderedAlphabet_PositiveIntegers().rank(1)
0
sage: OrderedAlphabet_PositiveIntegers().rank(8)
7
```

TESTS:

```
sage: OrderedAlphabet_PositiveIntegers().rank(-1)
...
TypeError: -1 not in alphabet
```

string_rep()

Returns the string representation of the alphabet.

TESTS:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_PositiveIntegers
sage: OrderedAlphabet_PositiveIntegers().string_rep()
'Positive Integers'
```

unrank(*i*)

Returns the i-th letter in self, where the first letter is the 0-th letter.

INPUT:

- i - an integer

OUTPUT: i+1

EXAMPLES:

```
sage: from sage.combinat.words.alphabet import OrderedAlphabet_PositiveIntegers
sage: OrderedAlphabet_PositiveIntegers().unrank(0)
1
sage: OrderedAlphabet_PositiveIntegers().unrank(7)
8
```

class OrderedAlphabet_class()
Generic class for ordered alphabets.

17.40.2 Shuffle products

class ShuffleProduct_overlapping(*w1, w2*)

class ShuffleProduct_overlapping_r(*w1, w2, r*)

class ShuffleProduct_shifted(*w1, w2*)

class ShuffleProduct_w1w2(*w1, w2*)

cardinality()
Returns the number of words in the shuffle product of w1 and w2.
It is given by binomial(len(w1)+len(w2), len(w1)).
EXAMPLES:

```
sage: from sage.combinat.words.shuffle_product import ShuffleProduct_w1w2
sage: w, u = map(Words("abcd"), ["ab", "cd"])
sage: S = ShuffleProduct_w1w2(w,u)
sage: S.cardinality()
6
```

17.40.3 Suffix Tries and Suffix Trees

class ImplicitSuffixTree(*word*)

active_state()
Returns the active state of the suffix tree.
EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.active_state()
(0, (6, 6))
```

edge_iterator()
Returns an iterator over the edges of the suffix tree. The edge from u to v labelled by (i,j) is returned as the tuple (u,v,(i,j)).
EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: sorted( ImplicitSuffixTree(Word("aaaaa")).edge_iterator() )
[(0, 1, (0, None))]
sage: sorted( ImplicitSuffixTree(Word([0,1,0,1])).edge_iterator() )
[(0, 1, (0, None)), (0, 2, (1, None))]
sage: sorted( ImplicitSuffixTree(Word()).edge_iterator() )
[]
```

factor_iterator(*n=None*)
Generate distinct factors of self.
INPUT:

- n - an integer, or None.

OUTPUT: If n is an integer, returns an iterator over all distinct factors of length n. If n is None, returns an iterator generating all distinct factors.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator() )
[word: , word: a, word: ac, word: aca, word: acao, word: ao, word: c, word: ca, word: cac, w
sage: sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator(1) )
[word: a, word: c, word: o]
sage: sorted( ImplicitSuffixTree(Word("cacao")).factor_iterator(2) )
[word: ac, word: ao, word: ca]
sage: sorted( ImplicitSuffixTree(Word([0,0,0])).factor_iterator() )
[word: , word: 0, word: 00, word: 000]
sage: sorted( ImplicitSuffixTree(Word([0,0,0])).factor_iterator(2) )
[word: 00]
sage: sorted( ImplicitSuffixTree(Word([0,0,0])).factor_iterator(0) )
[word: ]
sage: sorted( ImplicitSuffixTree(Word()).factor_iterator() )
[word: ]
sage: sorted( ImplicitSuffixTree(Word()).factor_iterator(2) )
[]
```

number_of_factors(*n=None*)

Count the number of distinct factors of self.word().

INPUT:

- n - an integer, or None.

OUTPUT: If n is an integer, returns the number of distinct factors of length n. If n is None, returns the total number of distinct factors.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: t = ImplicitSuffixTree(Word([1,2,1,3,1,2,1]))
sage: t.number_of_factors()
22
sage: t.number_of_factors(1)
3
sage: t.number_of_factors(9)
0
sage: t.number_of_factors(0)
1

sage: t = ImplicitSuffixTree(Word("cacao"))
sage: t.number_of_factors()
13
sage: map(t.number_of_factors, range(10))
[1, 3, 3, 3, 2, 1, 0, 0, 0, 0]

sage: t = ImplicitSuffixTree(Word("c"*1000))
sage: t.number_of_factors()
1001
sage: t.number_of_factors(17)
1
sage: t.number_of_factors(0)
1

sage: ImplicitSuffixTree(Word()).number_of_factors()
1
```

```
sage: blueberry = ImplicitSuffixTree(Word("blueberry"))
sage: blueberry.number_of_factors()
43
sage: map(blueberry.number_of_factors, range(10))
[1, 6, 8, 7, 6, 5, 4, 3, 2, 1]
```

plot(*word_labels=False, layout='tree', tree_root=0, tree_orientation='up', vertex_colors=None, edge_labels=True, *args, **kwds*)

Returns a Graphics object corresponding to the transition graph of the suffix tree.

INPUT:

- `word_labels` - if False, labels the edges by pairs (i, j); if True, labels the edges by word[i:j].

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: ImplicitSuffixTree(Word('cacao')).plot(word_labels=True)
sage: ImplicitSuffixTree(Word('cacao')).plot(word_labels=False)
```

TESTS:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: type(ImplicitSuffixTree(Word('cacao')).plot(word_labels=True))
<class 'sage.plot.plot.Graphics'>
sage: type(ImplicitSuffixTree(Word('cacao')).plot(word_labels=False))
<class 'sage.plot.plot.Graphics'>
```

process_letter(*letter*)

Modifies the current implicit suffix tree producing the implicit suffix tree for self.word() + letter.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("aco")("cacao")
sage: t = ImplicitSuffixTree(w[:-1]); t
Implicit Suffix Tree of the word: caca
sage: t.process_letter(w[-1]); t
Implicit Suffix Tree of the word: cacao

sage: W = Words([0,1])
sage: s = ImplicitSuffixTree(W([0,1,0,1])); s
Implicit Suffix Tree of the word: 0101
sage: s.process_letter(W([1])[0]); s
Implicit Suffix Tree of the word: 01011
```

show(*word_labels=None, *args, **kwds*)

Displays the output of self.plot().

INPUT:

- `word_labels` - if False, labels the edges by pairs (i, j); if True, labels the edges by word[i:j].

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("cao")("cacao")
sage: t = ImplicitSuffixTree(w)
sage: t.show(word_labels=True)
sage: t.show(word_labels=False)
```

states()

Returns the states (explicit nodes) of the suffix tree.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.states()
[0, 1, 2, 3, 4, 5, 6, 7]
```

suffix_link(*state*)

Evaluates the suffix link map of the implicit suffix tree on state. Note that the suffix link is not defined for all states.

The suffix link of a state x' that corresponds to the suffix x is defined to be -1 is x' is the root (0) and y' otherwise, where y' is the state corresponding to the suffix x[1:].

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.suffix_link(3)
5
sage: t.suffix_link(5)
0
sage: t.suffix_link(0)
-1
sage: t.suffix_link(-1)
Traceback (most recent call last):
    ...
TypeError: there is no suffix link from -1
```

to_digraph(*word_labels=False*)

Returns a DiGraph object of the transition graph of the suffix tree.

word_labels - if False, labels the edges by pairs (i, j); if True, labels the edges by word[i:j].

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.to_digraph()
Digraph on 8 vertices
```

to_explicit_suffix_tree()

Converts self to an explicit suffix tree. It is obtained by processing an end of string letter as if it were a regular letter, except that no new leaf nodes are created (thus, the only thing that happens is that some implicit nodes become explicit).

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: w = Words("aco")("cacao")
sage: t = ImplicitSuffixTree(w[:-1])
sage: t.to_explicit_suffix_tree()

sage: W = Words([0,1])
sage: s = ImplicitSuffixTree(W([0,1,0,1, 1]))
sage: s.to_explicit_suffix_tree()
```

transition_function(*word, node=0*)

Returns the node obtained by starting from node and following the edges labelled by the letters of word. Returns ("explicit", end_node) if we end at end_node, or ("implicit", (edge, d)) if we end d spots along an edge.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words([0,1,2])
sage: t = ImplicitSuffixTree(W([0,1,0,1,2]))
sage: t.transition_function(W([0,1,0]))
('implicit', (3, 1), 1)
sage: t.transition_function(W([0,1,2]))
('explicit', 4)
sage: t.transition_function(W([0,1,2]), 5)
('explicit', 2)
sage: t.transition_function(W([0,1]), 5)
('implicit', (5, 2), 2)
```

transition_function_dictionary()

Returns the transition function as a dictionary of dictionaries. The format is consistent with the input format for DiGraph.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: W = Words("aco")
sage: t = ImplicitSuffixTree(W("cac"))
sage: t.transition_function_dictionary()
{0: {1: (0, None), 2: (1, None)}}

sage: W = Words([0,1])
sage: t = ImplicitSuffixTree(W([0,1,0]))
sage: t.transition_function_dictionary()
{0: {1: (0, None), 2: (1, None)}}
```

trie_type_dict()

Returns a dictionary in a format compatible with that of the suffix trie transition function.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree, SuffixTrie
sage: W = Words("ab")
sage: t = ImplicitSuffixTree(W("aba"))
sage: t.trie_type_dict() == dict([[(0, W("a")), 4], [(0, W("b")), 3], [(3, W("a")), 2], [(4,
True
```

uncompactify()

Returns the tree obtained from self by splitting edges so that they are labelled by exactly one letter. The resulting tree is isomorphic to the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree, SuffixTrie
sage: abbab = Words("ab")("abbab")
sage: s = SuffixTrie(abbab)
sage: t = ImplicitSuffixTree(abbab)
sage: t.uncompactify().is_isomorphic(s.to_digraph())
True
```

word()

Returns the word whose implicit suffix tree this is.

TEST:

```
sage: from sage.combinat.words.suffix_trees import ImplicitSuffixTree
sage: ImplicitSuffixTree(Word([0,1,0,1,0])).word() == Word([0,1,0,1,0])
True
```

NaiveSuffixTree(*word*)

Given a word, constructs the suffix tree of the word using a naive algorithm. Useful for testing.

INPUT:

- `word` - any word

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import NaiveSuffixTree
sage: W = Words('abco')
sage: s = NaiveSuffixTree(W("abacabaabaca")); s
Suffix Tree of the word: abacabaabaca

sage: s = NaiveSuffixTree(W("cacao")); s
Suffix Tree of the word: cacao
sage: s.edges()
[(0, 3, word: ca), (0, 5, word: a), (0, 7, word: o), (3, 1, word: cao), (3, 4, word: o), (5, 2,

sage: W = Words([0,1])
sage: s = NaiveSuffixTree(W([0,1,0,1,1])); s
Suffix Tree of the word: 01011
sage: s.edges()
[(0, 3, word: 01), (0, 5, word: 1), (3, 1, word: 011), (3, 4, word: 1), (5, 2, word: 011), (5, 6
```

class **NaiveSuffixTreeClass**(*data=None, pos=None, loops=None, format=None, boundary=, [], weighted=None, implementation='networkx', sparse=True, vertex_labels=True, **kwds*)

naive_process_suffix(*suffix*)

Helper function for constructing the suffix tree.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import NaiveSuffixTreeClass
sage: W = Words([0,1,2])
sage: NST = NaiveSuffixTreeClass({0:[]})
sage: NST.set_vertex(0,{'position':0,'suffix':True})
sage: NST.naive_process_suffix(W([0,1,1,2,0])); NST
Suffix Tree of the word: 01120
```

node_to_word(*node*)

Returns the word obtained by reading the edge labels from the root to node.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import NaiveSuffixTree
sage: NaiveSuffixTree(Word("abacabaabaca")).node_to_word(17)
word: ca
sage: NaiveSuffixTree(Word("cacao")).node_to_word(0)
word:
sage: NaiveSuffixTree(Word([0,1,1,0,1,0,0,1])).node_to_word(10)
word: 001
```

plot(*layout='tree', tree_root=0, tree_orientation='up', vertex_colors=None, edge_labels=True, *args, **kwds*)

Returns a Graphics object associated to self.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import NaiveSuffixTree
sage: NaiveSuffixTree(Word("abacabaabaca")).plot()
```

TESTS:

```
sage: from sage.combinat.words.suffix_trees import NaiveSuffixTree
sage: type(NaiveSuffixTree(Word("abacabaabaca")).plot())
<class 'sage.plot.plot.Graphics'>
```

word()

Returns the word whose suffix tree this is.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import NaiveSuffixTree
sage: NaiveSuffixTree(Word("abcba")).word()
word: abcba
```

word_to_node(*node, word*)

Returns the node obtained by starting from node and following the edges labelled by the letters of word. Returns ("node", node2, word2) if we end at node2, or ("implicit", edge, word2) if we end in the middle of an edge (called an implicit node), where word2 is the suffix of word that does not match any more edge labels.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import NaiveSuffixTree
sage: W = Words("abc")
sage: NST = NaiveSuffixTree(W("abacabaabaca"))
sage: NST.word_to_node(0, W("ca"))
('node', 17, word: )
sage: NST.word_to_node(0, W("cab"))
('implicit', ((17, 5, word: baabaca), 1), word: )
sage: NST.word_to_node(17, W("b"))
('implicit', ((17, 5, word: baabaca), 1), word: )
sage: NST.word_to_node(15, W("baabacabca"))
('node', 4, word: bca)
```

class **SuffixTrie**(*word*)

active_state()

Returns the active state of the suffix trie. This is the state corresponding to the word as a suffix of itself.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: t.active_state()
8

sage: u = Words([0,1])([0,1,1,0,1,0,0,1])
sage: s = SuffixTrie(u)
sage: s.active_state()
22
```

final_states()

Returns the set of final states of the suffix trie. These are the states corresponding to the suffixes of self.word(). They are obtained be repeatedly following the suffix link from the active state until we reach 0.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
```

```
sage: t.final_states() == Set([8, 9, 10, 11, 12, 0])
True
```

has_suffix(*word*)

Return True if and only if word is a suffix of self.word().

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: [t.has_suffix(w[i:]) for i in range(len(w)+1)]
[True, True, True, True, True, True]
sage: [t.has_suffix(w[:i]) for i in range(len(w)+1)]
[True, False, False, False, False, True]
```

node_to_word(*state=0*)

Returns the word obtained by reading the edge labels from 0 to state.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("abc")("abcba")
sage: t = SuffixTrie(w)
sage: t.node_to_word(10)
word: abcba
sage: t.node_to_word(7)
word: abcb
```

plot(*layout='tree', tree_root=0, tree_orientation='up', vertex_colors=None, edge_labels=True, *args, **kwds*)

Returns a Graphics object corresponding to the transition graph of the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: SuffixTrie(Word("cacao")).plot()
```

TESTS:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: type(SuffixTrie(Word("cacao")).plot())
<class 'sage.plot.plot.Graphics'>
```

process_letter(*letter*)

Modify self to produce the suffix trie for self.word() + letter.

NOTE: letter must occur within the alphabet of the word.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("ab")("ababba")
sage: t = SuffixTrie(w); t
Suffix Trie of the word: ababba
sage: t.process_letter("a"); t
Suffix Trie of the word: ababbaa
```

TESTS:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w); t
Suffix Trie of the word: cacao
sage: t.process_letter("d")
Traceback (most recent call last):
```

```
    ...
IndexError: letter not in alphabet: 'd'
```

show(**args, **kwds*)

Displays the output of self.plot().

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cac")
sage: t = SuffixTrie(w)
sage: t.show()
```

states()

Returns the states of the automaton defined by the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words([0,1])([0,1,1])
sage: t = SuffixTrie(w)
sage: t.states()
[0, 1, 2, 3, 4]

sage: u = Words("aco")("cacao")
sage: s = SuffixTrie(u)
sage: s.states()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

suffix_link(*state*)

Evaluates the suffix link map of the suffix trie on state. Note that the suffix link map is not defined on -1.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: map(t.suffix_link, range(13))
[-1, 0, 3, 0, 5, 1, 7, 2, 9, 10, 11, 12, 0]
sage: t.suffix_link(0)
-1
```

TESTS:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cacao")
sage: t = SuffixTrie(w)
sage: t.suffix_link([1])
Traceback (most recent call last):
    ...
TypeError: [1] is not an integer
sage: t.suffix_link(-1)
Traceback (most recent call last):
    ...
TypeError: suffix link is not defined for -1
sage: t.suffix_link(17)
Traceback (most recent call last):
    ...
TypeError: 17 is not a state
```

to_digraph()

Returns a DiGraph object of the transition graph of the suffix trie.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("cao")("cac")
sage: t = SuffixTrie(w)
sage: d = t.to_digraph(); d
Digraph on 6 vertices
sage: d.adjacency_matrix()
[0 1 0 1 0 0]
[0 0 1 0 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1]
[0 0 0 0 0 0]
[0 0 0 0 0 0]
```

transition_function(*node, word*)

Returns the state reached by beginning at node and following the arrows in the transition graph labelled by the letters of word.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words([0,1])([0,1,0,1,1])
sage: t = SuffixTrie(w)
sage: [t.transition_function(u,letter) == v \
        for ((u,letter),v) in t._transition_function.iteritems()] \
        == [True] * len(t._transition_function)
True
```

word()

Returns the word whose suffix tree this is.

EXAMPLES:

```
sage: from sage.combinat.words.suffix_trees import SuffixTrie
sage: w = Words("abc")("abcba")
sage: t = SuffixTrie(w)
sage: t.word()
word: abcba
sage: t.word() == w
True
```

17.40.4 Word morphisms

class WordMorphism(*data, codomain=None*)

TESTS:

```
sage: wm = WordMorphism('a->ab,b->ba')
sage: wm == loads(dumps(wm))
True
```

codomain()

Returns the domain of the images.

EXAMPLES

```
sage: WordMorphism('a->ab,b->a').codomain()
Words over Ordered Alphabet ['a', 'b']
sage: WordMorphism('6->ab,y->5,0->asd').codomain()
Words over Ordered Alphabet ['5', 'a', 'b', 'd', 's']
```

conjugate(*pos*)

Returns the morphism where each image of self is conjugate of parameter pos.

Retourne le morphisme dont toutes les images ont etes conjuguees de pos.

EXAMPLES

```
sage: m = WordMorphism('a->abcde')
sage: m.conjugate(0) == m
True
sage: print m.conjugate(1)
WordMorphism: a->bcdea
sage: print m.conjugate(3)
WordMorphism: a->deabc
sage: print WordMorphism('').conjugate(4)
WordMorphism:
sage: m = WordMorphism('a->abcde,b->xyz')
sage: print m.conjugate(2)
WordMorphism: a->cdeab, b->zxy
```

domain()

Returns domain of self.

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').domain()
Words over Ordered Alphabet ['a', 'b']
sage: WordMorphism('b->ba,a->ab').domain()
Words over Ordered Alphabet ['a', 'b']
sage: WordMorphism('6->ab,y->5,0->asd').domain()
Words over Ordered Alphabet ['0', '6', 'y']
```

fixed_point(*letter*)

Returns the fixed point of self beginning by the given letter.

A fixed point of morphism φ is a word w such that $\varphi(w) = w$.

INPUT:

- `self` - an endomorphism, must be prolongable on letter
- `letter` - in the domain of self, the first letter of the fixed point.

OUTPUT:

- `word` - the fixed point of self beginning with letter.

EXAMPLES:

1. Infinite fixed point:

```
sage: WordMorphism('a->ab,b->ba').fixed_point(letter='a')
Fixed point beginning with 'a' of the morphism WordMorphism: a->ab, b->ba
sage: WordMorphism('a->ab,b->a').fixed_point(letter='a')
Fixed point beginning with 'a' of the morphism WordMorphism: a->ab, b->a
sage: WordMorphism('a->ab,b->b,c->ba').fixed_point(letter='a')
Fixed point beginning with 'a' of the morphism WordMorphism: a->ab, b->b, c->ba
```

2. Infinite fixed point of an erasing morphism:

```
sage: WordMorphism('a->ab,b->,c->ba').fixed_point(letter='a')
Fixed point beginning with 'a' of the morphism WordMorphism: a->ab, b->, c->ba
```

3. Finite fixed point:

```
sage: WordMorphism('a->ab,b->b,c->ba').fixed_point(letter='b')
word: b
```

4. Finite fixed point of an erasing morphism:

```
sage: m = WordMorphism('a->abc,b->,c->')
sage: fp = m.fixed_point('a'); fp
Fixed point beginning with 'a' of the morphism WordMorphism: a->abc, b->, c->
sage: fp[:10]
word: abc
sage: m = WordMorphism('a->ba,b->')
sage: m('ba')
word: ba
sage: m.fixed_point('a') #todo: not implemented
word: ba
```

5. Fixed point of a power of a morphism:

```
sage: m = WordMorphism('a->ba,b->ab')
sage: (m^2).fixed_point(letter='a')
Fixed point beginning with 'a' of the morphism WordMorphism: a->abba, b->baab
```

TESTS:

```
sage: WordMorphism('a->ab,b->,c->ba').fixed_point(letter='b')
...
TypeError: self must be prolongable on b
sage: WordMorphism('a->ab,b->,c->ba').fixed_point(letter='c')
...
TypeError: self must be prolongable on c
sage: WordMorphism('a->ab,b->,c->ba').fixed_point(letter='d')
...
TypeError: letter (=d) is not in the domain alphabet (=Ordered Alphabet ['a', 'b', 'c'])
sage: WordMorphism('a->aa,b->aac').fixed_point(letter='a')
...
TypeError: self (=WordMorphism: a->aa, b->aac) is not a endomorphism
```

has_conjugate_in_classP (*f=None*)

Returns True if self has a conjugate in class f-P.

DEFINITION: Let Σ be an alphabet and f be an involution on Σ. "We say that a morphism φ is in class f-P if there exists an f-palindrome $p \in$ f - Pal(Σ^*) and for each $\alpha \in \Sigma$ there exists an f-palindrome $q_\alpha \in$ f - Pal(Σ^*) such that $\varphi(\alpha) = pq_\alpha$." [2]

We say that a morphism φ' est de classe $f - P'$ if there exists a morphism φ conjugate of φ' such that φ is in class f-P. [2]

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's contructor understand (dict, str, ...).

OUTPUT:

- `boolean` - True if f is None and if self has a conjugate in class P; True otherwise and if self has a conjugate in class f-P; False otherwise.

REFERENCES:

- [1] Hof, A., O. Knill et B. Simon, Singular continuous spectrum for palindromic Schrödinger operators, Commun. Math. Phys. 174 (1995) 149-159.
- [2] Labbé, Sébastien. Propriétés combinatoires des f-palindromes, Mémoire de maîtrise en Mathématiques, Montréal, UQAM, 2008, 109 pages.

EXAMPLES:

```
sage: fibo = WordMorphism('a->ab,b->a')
sage: fibo.has_conjugate_in_classP()
True
```

```
sage: (fibo^2).is_in_classP()
False
sage: (fibo^2).has_conjugate_in_classP()
True
```

has_left_conjugate()

Returns true if all the non empty images of self begins with the same letter.

Retourne vrai si toutes les images non vides commencent par la meme lettre.

EXAMPLES:

```
sage: m = WordMorphism('a->abcde,b->xyz')
sage: m.has_left_conjugate()
False
sage: WordMorphism('b->xyz').has_left_conjugate()
True
sage: WordMorphism('').has_left_conjugate()
True
sage: WordMorphism('a->,b->xyz').has_left_conjugate()
True
sage: WordMorphism('a->abbab,b->abb').has_left_conjugate()
True
sage: WordMorphism('a->abbab,b->abb,c->').has_left_conjugate()
True
```

has_right_conjugate()

Returns true if all the non empty images of self ends with the same letter.

Retourne vrai si toutes les images non vide terminent par la meme lettre.

EXAMPLES:

```
sage: m = WordMorphism('a->abcde,b->xyz')
sage: m.has_right_conjugate()
False
sage: WordMorphism('b->xyz').has_right_conjugate()
True
sage: WordMorphism('').has_right_conjugate()
True
sage: WordMorphism('a->,b->xyz').has_right_conjugate()
True
sage: WordMorphism('a->abbab,b->abb').has_right_conjugate()
True
sage: WordMorphism('a->abbab,b->abb,c->').has_right_conjugate()
True
```

images()

Returns the list of all the images of the letters of the alphabet under self.

EXAMPLES

```
sage: WordMorphism('a->ab,b->a').images()
[word: ab, word: a]
sage: WordMorphism('6->ab,y->5,0->asd').images()
[word: 5, word: asd, word: ab]
```

incidence_matrix()

Returns the incidence matrix of the morphism. The order of the rows and column are given by the order defined on the alphabet of the domain and the codomain.

The matrix returned is over the integers. If a different ring is desired, use either the change_ring function or the matrix function.

EXAMPLES:

```
sage: m = WordMorphism('a->abc,b->a,c->c')
sage: m.incidence_matrix()
[1 1 0]
[1 0 0]
[1 0 1]
sage: m = WordMorphism('a->abc,b->a,c->c,d->abbccccabca,e->abc')
sage: m.incidence_matrix()
[1 1 0 3 1]
[1 0 0 3 1]
[1 0 1 5 1]
```

is_empty()

Returns True if the cardinality of the domain is zero and False otherwise.

EXAMPLES:

```
sage: WordMorphism('').is_empty()
True
sage: WordMorphism('a->a').is_empty()
False
```

is_endomorphism()

Returns True if the codomain is a subset of the domain.

EXAMPLES

```
sage: WordMorphism('a->ab,b->a').is_endomorphism()
True
sage: WordMorphism('6->ab,y->5,0->asd').is_endomorphism()
False
sage: WordMorphism('a->a,b->aa,c->aaa').is_endomorphism()
True
sage: Wabc = Words('abc')
sage: m = WordMorphism('a->a,b->aa,c->aaa',codomain = Wabc)
sage: m.is_endomorphism()
True
```

is_erasing()

Returns True if self is an erasing morphism, i.e. the image of a letter is the empty word.

EXAMPLES

```
sage: WordMorphism('a->ab,b->a').is_erasing()
False
sage: WordMorphism('6->ab,y->5,0->asd').is_erasing()
False
sage: WordMorphism('6->ab,y->5,0->asd,7->').is_erasing()
True
sage: WordMorphism('').is_erasing()
False
```

is_identity()

Returns True if self is the identity morphism.

EXAMPLES:

```
sage: m = WordMorphism('a->a,b->b,c->c,d->e')
sage: m.is_identity()
False
sage: WordMorphism('a->a,b->b,c->c').is_identity()
True
sage: WordMorphism('a->a,b->b,c->cb').is_identity()
False
```

```
sage: m = WordMorphism('a->b,b->c,c->a')
sage: (m^2).is_identity()
False
sage: (m^3).is_identity()
True
sage: (m^4).is_identity()
False
sage: WordMorphism('').is_identity()
True
sage: WordMorphism({0:[0],1:[1]}).is_identity()
True
```

is_in_classP (*f=None*)

Returns True if self is in class P (or f-P).

DEFINITION: "[Let A be an alphabet] We say that a primitive substitution S is in the *class P* if there exists a palindrome p and for each $b \in A$ a palindrome q_b such that $S(b) = pq_b$ for all $b \in A$." [1]

Let Σ be an alphabet and f be an involution on Σ. "We say that a morphism φ is in class f-P if there exists an f-palindrome $p \in$ f - Pal(Σ^*) and for each $\alpha \in \Sigma$ there exists an f-palindrome $q_\alpha \in$ f - Pal(Σ^*) such that $\varphi(\alpha) = pq_\alpha$." [2]

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's contructor understand (dict, str, ...).

OUTPUT:

- `boolean` - True if f is None and if self is in class P; True otherwise and if self is in class f-P; False otherwise.

REFERENCES:

- [1] Hof, A., O. Knill et B. Simon, Singular continuous spectrum for palindromic Schrödinger operators, Commun. Math. Phys. 174 (1995) 149-159.
- [2] Labbé, Sébastien. Propriétés combinatoires des f-palindromes, Mémoire de maîtrise en Mathématiques, Montréal, UQAM, 2008, 109 pages.

EXAMPLES:

```
sage: WordMorphism('a->bbaba,b->bba').is_in_classP()
True
sage: tm = WordMorphism('a->ab,b->ba')
sage: tm.is_in_classP()
False
sage: f='a->b,b->a'
sage: tm.is_in_classP(f=f)
True
sage: (tm^2).is_in_classP()
True
sage: (tm^2).is_in_classP(f=f)
False
sage: fibo = WordMorphism('a->ab,b->a')
sage: fibo.is_in_classP()
True
sage: fibo.is_in_classP(f=f)
False
sage: (fibo^2).is_in_classP()
False
sage: f='a->b,b->a,c->c'
sage: WordMorphism('a->acbcc,b->acbab,c->acbba').is_in_classP (f)
True
```

is_involution()

Returns True if self is an involution, i.e. its square is the identity.

EXAMPLES:

```
sage: WordMorphism('a->b,b->a').is_involution()
True
sage: WordMorphism('').is_involution()
True
sage: WordMorphism({0:[1],1:[0]}).is_involution()
True
```

is_primitive()

Returns True if self is primitive.

INPUT:

- `self` - an endomorphism

-In English: A morphism φ is *primitive* if there exists an positive integer k such that for all $\alpha \in \Sigma$, $\varphi^k(\alpha)$ contains all the letters of Σ.

-En français: Un morphisme φ est *primitif* s'il existe un nombre naturel k tel que pour tout $\alpha \in \Sigma$, $\varphi^k(\alpha)$ contient toutes les lettres de Σ.

EXAMPLES:

```
sage: tm = WordMorphism('a->ab,b->ba')
sage: tm.is_primitive()
True
sage: fibo = WordMorphism('a->ab,b->a');
sage: fibo.is_primitive()
True
sage: m = WordMorphism('a->bb,b->aa')
sage: m.is_primitive()
False
sage: f = WordMorphism({0:[1],1:[0]})
sage: f.is_primitive()
False
sage: m = WordMorphism('a->bb,b->aac')
sage: m.is_primitive()
...
TypeError: self (=WordMorphism: a->bb, b->aac) is not a endomorphism
sage: m = WordMorphism('a->,b->',codomain=Words('ab'))
sage: m.is_primitive()
False
sage: m = WordMorphism('a->,b->')
sage: m.is_primitive()
False
```

is_prolongable(*letter*)

Returns True if self is prolongable on letter.

A morphism φ is prolongable on a letter a if a is a prefix of $\varphi(a)$.

INPUT:

- `self` - the codomain must be an instance of Words
- `letter` - in the domain alphabet

OUTPUT:

- `boolean` - if self is prolongable on letter.

EXAMPLES:

```
sage: WordMorphism('a->ab,b->a').is_prolongable(letter='a')
True
sage: WordMorphism('a->ab,b->a').is_prolongable(letter='b')
False
sage: WordMorphism('a->ba,b->ab').is_prolongable(letter='b')
False
sage: (WordMorphism('a->ba,b->ab')^2).is_prolongable(letter='b')
True
sage: WordMorphism('a->ba,b->').is_prolongable(letter='b')
False
sage: WordMorphism('a->bb,b->aac').is_prolongable(letter='a')
False
```

TESTS:

```
sage: WordMorphism('a->ab,b->b,c->ba').is_prolongable(letter='d')
...
TypeError: letter (=d) is not in the domain alphabet (=Ordered Alphabet ['a', 'b', 'c'])
sage: n0, n1 = matrix(2,[1,1,1,0]), matrix(2,[2,1,1,0])
sage: n = {'a':n0, 'b':n1}
sage: WordMorphism(n).is_prolongable(letter='a') #todo: not implemented
...
TypeError: codomain of self must be an instance of Words
```

letter_iterator(*letter*)

Returns an iterator of the letters of the fixed point of self starting with letter.

If w is the word, then this iterator: outputs the elements of morphism[w[i]], appends morphism[w[i+1]] to w, increments i.

INPUT:

- `self` - an endomorphism, must be prolongable on letter
- `letter` - in the domain of self

OUTPUT:

- `iterator` - iterator of the fixed point

EXAMPLES:

```
sage: m = WordMorphism('a->abc,b->,c->')
sage: list(m.letter_iterator('b'))
...
TypeError: self must be prolongable on b
sage: list(m.letter_iterator('a'))
['a', 'b', 'c']
sage: m = WordMorphism('a->aa,b->aac')
sage: list(m.letter_iterator('a'))
...
TypeError: self (=WordMorphism: a->aa, b->aac) is not a endomorphism
```

list_fixed_points()

Returns the list of all fixed points of self.

EXAMPLES

```
sage: WordMorphism('a->ab,b->ba').list_fixed_points() #not implemented
[Fixed point beginning with 'a' of the morphism WordMorphism: a->ab, b->ba,
Fixed point beginning with 'b' of the morphism WordMorphism: a->ab, b->ba]
```

list_of_conjugates()

Retourne une liste des morphismes conjugues du receveur obtenus en conjuguant les prefixes (et suffixes) communs de chacune des images.

Returns the list of all the conjugate morphisms of self obtain by taking the conjugate of the common prefixes and suffixes of all the images.

DEFINITION (from [1]): Recall from Lothaire [2] (Section 2.3.4) that φ is *right conjugate* of φ', noted $\varphi \triangleleft \varphi'$, if there exists $u \in \Sigma^*$ such that

$$\varphi(\alpha)u = u\varphi'(\alpha), \quad \text{for all } \alpha \in \Sigma$$

, or equivalently that $\varphi(x)u = u\varphi'(x)$, for all words $x \in \Sigma^*$. Clearly, this relation is not symmetric so that we say that two morphisms φ and φ' are *conjugate*, noted $\varphi \bowtie \varphi'$, if $\varphi \triangleleft \varphi'$ or $\varphi' \triangleleft \varphi$. It is easy to see that conjugacy of morphisms is an equivalence relation.

REFERENCES:

- [1] A. Blondin-Massé, S. Brlek, S. Labbé, Palindromic lacunas of the Thue-Morse word, 2008, to appear.
- [2] M. Lothaire, Algebraic Combinatorics on words, Cambridge University Press, 2002.

EXAMPLES:

```
sage: m = WordMorphism('a->abbab,b->abb')
sage: map(str, m.list_of_conjugates())
['WordMorphism: a->babba, b->bab',
'WordMorphism: a->abbab, b->abb',
'WordMorphism: a->bbaba, b->bba',
'WordMorphism: a->babab, b->bab',
'WordMorphism: a->ababb, b->abb',
'WordMorphism: a->babba, b->bba',
'WordMorphism: a->abbab, b->bab']
sage: m = WordMorphism('a->aaa,b->aa')
sage: map(str, m.list_of_conjugates())
['WordMorphism: a->aaa, b->aa']
sage: WordMorphism('').list_of_conjugates()
[Morphism from Words over Ordered Alphabet [] to Words over Ordered Alphabet []]
sage: m = WordMorphism('a->aba,b->aba')
sage: map(str, m.list_of_conjugates())
['WordMorphism: a->baa, b->baa',
'WordMorphism: a->aab, b->aab',
'WordMorphism: a->aba, b->aba']
sage: m = WordMorphism('a->abb,b->abbab,c->')
sage: map(str, m.list_of_conjugates())
['WordMorphism: a->bab, b->babba, c->',
'WordMorphism: a->abb, b->abbab, c->',
'WordMorphism: a->bba, b->bbaba, c->',
'WordMorphism: a->bab, b->babab, c->',
'WordMorphism: a->abb, b->ababb, c->',
'WordMorphism: a->bba, b->babba, c->',
'WordMorphism: a->bab, b->abbab, c->']
```

reversal()

Returns the reversal of self.

EXAMPLES

```
sage: print WordMorphism('6->ab,y->5,0->asd').reversal()
WordMorphism: 0->dsa, 6->ba, y->5
sage: print WordMorphism('a->ab,b->a').reversal()
WordMorphism: a->ba, b->a
```

17.40.5 Words of all kinds

AUTHORS:

- Arnaud Bergeron
- Amy Glen
- Sébastien Labbé
- Franco Saliola

EXAMPLES:

We define the set of words over 'a' and 'b'.

```
sage: W = Words('ab'); W
Words over Ordered Alphabet ['a', 'b']
```

Then we can build some words in the set:

```
sage: W('abba')
word: abba
```

If you try to use letters that are not the alphabet of the set you get an error:

```
sage: W([1, 2])
...
IndexError: letter not in alphabet: 1
```

You can also build infinite words backed by a function or an iterator!

```
sage: def f(n):
...       if n % 2 == 1:
...           return 'a'
...       else:
...           return 'b'
...
sage: W(f)
Infinite word over ['a', 'b']
```

class AbstractFiniteWord(*parent, word, mapping=None, format=None, part=slice(None, None, None)*)

class AbstractInfiniteWord(*parent, word, mapping=None, format=None, part=slice(None, None, None)*)

class AbstractWord(*parent, word, mapping=None, format=None, part=slice(None, None, None)*)

coerce(*other*)

Returns a pair of words with a common parent or raises an exception.

This function begins by checking if both words have the same parent. If this is the case, then no work is done and both words are returned as-is.

Otherwise it will attempt to convert other to the domain of self. If that fails, it will attempt to convert self to the domain of other. If both attempts fail, it raises a TypeError to signal failure.

EXAMPLES:

```
sage: W1 = Words('abc'); W2 = Words('ab')
sage: w1 = W1('abc'); w2 = W2('abba'); w3 = W1('baab')
sage: w1.parent() is w2.parent()
False
sage: a, b = w1.coerce(w2)
sage: a.parent() is b.parent()
True
```

parent()

Returns the parent object from which the word was created.

EXAMPLES:

```
sage: from sage.combinat.words.word import AbstractWord
sage: W = Words('ab')
sage: w = AbstractWord(W, '')
sage: w.parent()
Words over Ordered Alphabet ['a', 'b']
sage: w.parent() is W
True
```

size_of_alphabet()

Returns the size of the alphabet of the word.

TESTS:

```
sage: from sage.combinat.words.word import AbstractWord
sage: AbstractWord(Words('ab'), '').size_of_alphabet()
2
sage: AbstractWord(Words('abc'), '').size_of_alphabet()
3
```

class **FiniteWord_over_Alphabet**(*parent, *args, **kwds*)

class **FiniteWord_over_OrderedAlphabet**(*parent, *args, **kwds*)

BWT()

Returns the Burrows-Wheeler Transform (BWT) of self.

The *Burrows-Wheeler transform* of a finite word w is obtained from w by first listing the conjugates of w in lexicographic order and then concatenating the final letters of the conjugates in this order. See [1].

EXAMPLES:

```
sage: W = Words('abc')
sage: W('abaccaaba').BWT()
word: cbaabaaca
sage: W('abaab').BWT()
word: bbaaa
sage: W('bbabbaca').BWT()
word: cbbbbaaa
sage: W('aabaab').BWT()
word: bbaaaa
sage: Word().BWT()
word:
sage: W('a').BWT()
word: a
```

REFERENCES:

- [1] M. Burrows, D.J. Wheeler, "A block-sorting lossless data compression algorithm", HP Lab Technical Report, 1994, available at http://www.hpl.hp.com/techreports/Compaq-DEC/SRC-RR-124.html

apply_morphism(*morphism*)

Returns the word obtained by applying the morphism to self.

INPUT:

- `morphism` - Can be an instance of WordMorphism, or anything that can be used to construct one.

EXAMPLES:

```
sage: w = Word("ab")
sage: d = {'a':'ab', 'b':'ba'}
```

```
sage: w.apply_morphism(d)
word: abba
sage: w.apply_morphism(WordMorphism(d))
word: abba
```

apply_permutation_to_letters(*permutation*)

Return the word obtained by applying permutation to the letters of the alphabet of self.

EXAMPLES:

```
sage: w = Words('abcd')('abcd')
sage: p = [2,1,4,3]
sage: w.apply_permutation_to_letters(p)
word: badc
sage: u = Words('dabc')('abcd')
sage: u.apply_permutation_to_letters(p)
word: dcba
sage: w.apply_permutation_to_letters(Permutation(p))
word: badc
sage: w.apply_permutation_to_letters(PermutationGroupElement(p))
word: badc
```

apply_permutation_to_positions(*permutation*)

Return the word obtained by permuting the positions of the letters in self.

EXAMPLES:

```
sage: w = Words('abcd')('abcd')
sage: w.apply_permutation_to_positions([2,1,4,3])
word: badc
sage: u = Words('dabc')('abcd')
sage: u.apply_permutation_to_positions([2,1,4,3])
word: badc
sage: w.apply_permutation_to_positions(Permutation([2,1,4,3]))
word: badc
sage: w.apply_permutation_to_positions(PermutationGroupElement([2,1,4,3]))
word: badc
```

border()

Returns the longest word that is both a proper prefix and a proper suffix of self.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('121212').border()
word: 1212
sage: W('12321').border()
word: 1
sage: W().border() is None
True
```

charge(*check=True*)

EXAMPLES:

```
sage: Word([1,1,2,2,3]).charge()
0
sage: Word([3,1,1,2,2]).charge()
1
sage: Word([2,1,1,2,3]).charge()
1
sage: Word([2,1,1,3,2]).charge()
2
```

```
sage: Word([3,2,1,1,2]).charge()
2
sage: Word([2,2,1,1,3]).charge()
3
sage: Word([3,2,2,1,1]).charge()
4
```

TESTS:

```
sage: Word([3,3,2,1,1]).charge()
...
ValueError: the evaluation of the word must be a partition
```

colored_vector(*x=0, y=0, width='default', height=1, cmap='hsv', thickness=1*)

Returns a vector (Graphics object) illustrating self. Each letter is represented by a colored rectangle. There is a unique color for each letter of the alphabet.

INPUT:

- `x` - (default: 0) bottom left x-coordinate of the vector
- `y` - (default: 0) bottom left y-coordinate of the vector
- `width` - (default: 'default') width of the vector. By default, the width is the length of self.
- `height` - (default: 1) height of the vector
- `thickness` - (default: 1) thickness of the contour
- `cmap` - (default: 'hsv') color map

OUTPUT: Graphics

EXAMPLES:

```
sage: Word(range(20)).colored_vector()

sage: Word(range(100)).colored_vector(0,0,10,1)

sage: Words(range(100))(range(10)).colored_vector()

sage: w = Word('abbabaab')
sage: w.colored_vector()

sage: w.colored_vector(cmap='autumn')
```

TESTS:

```
sage: Word(range(100)).colored_vector(cmap='jolies')
...
RuntimeError: Color map jolies not known
sage: Word(range(100)).colored_vector(cmap='__doc__')
...
RuntimeError: Color map __doc__ not known
```

commutes_with(*other*)

Returns True if self commutes with other, and False otherwise.

EXAMPLES:

```
sage: W = Words('12')
sage: W('12').commutes_with(W('12'))
True
sage: W('12').commutes_with(W('11'))
False
sage: W().commutes_with(W('21'))
True
```

complete_return_words(*fact*)

Returns the set of complete return words of fact in self.

This is the set of all factors starting by the given factor and ending just after the next occurrence of this factor. See for instance [1].

EXAMPLES:

```
sage: W = Words('123')
sage: W('21331233213231').complete_return_words(W('2')) == set([W('213312'), W('2332'), W('2
True
sage: W('').complete_return_words(W('213')) == set()
True
sage: W('121212').complete_return_words(W('1212')) == set([W('121212')])
True
```

REFERENCES:

- [1] J. Justin, L. Vuillon, Return words in Sturmian and episturmian words, Theor. Inform. Appl. 34 (2000) 343-356.

conjugate(*pos*)

Returns the conjugate at pos of self.

pos can be any integer, the distance used is the modulo by the length of self.

EXAMPLES:

```
sage: W = Words('12')
sage: W('12112').conjugate(1)
word: 21121
sage: W().conjugate(2)
word:
sage: W('12112').conjugate(8)
word: 12121
sage: W('12112').conjugate(-1)
word: 21211
```

conjugate_position(*other*)

Returns the position where self is conjugate with other. Returns None if there is no such position.

EXAMPLES:

```
sage: W = Words('123')
sage: W('12113').conjugate_position(W('31211'))
1
sage: W('12131').conjugate_position(W('12113')) is None
True
sage: W().conjugate_position(W('123')) is None
True
```

conjugates()

Returns the set of conjugates of self.

TESTS:

```
sage: W = Words('abc')
sage: W('cbbca').conjugates() == set([W('cacbb'),W('bbcac'),W('acbbc'),W('cbbca'),W('bcacb')
True
sage: W('abcabc').conjugates() == set([W('abcabc'),W('bcabca'),W('cabcab')])
True
sage: W().conjugates() == set([W()])
True
sage: W('a').conjugates() == set([W('a')])
True
```

count (*letter*)

Counts the number of occurrences of letter in self.

EXAMPLES:

```
sage: Words('ab')('abbabaab').count('a')
4
```

critical_exponent()

Returns the critical exponent of self.

The *critical exponent* of a word is the supremum of the order of all its (finite) factors. See [1].

EXAMPLES:

```
sage: Word('aaba').critical_exponent()
2
sage: Word('aabaa').critical_exponent()
2
sage: Word('aabaaba').critical_exponent()
7/3
sage: Word('ab').critical_exponent()
1
sage: Word('aba').critical_exponent()
3/2
sage: words.ThueMorseWord()[:20].critical_exponent()
2
```

REFERENCES:

- [1] F. Dejean. Sur un théorème de Thue. J. Combinatorial Theory Ser. A 13:90–99, 1972.

crochemore_factorization()

Returns the Crochemore factorization of self as an ordered list of factors.

The *Crochemore factorization* of a finite word w is the unique factorization: $(x_1, x_2, \ldots, x_n)$ of w with each x_i satisfying either: C1. x_i is a letter that does not appear in $u = x_1 \ldots x_{i-1}$; C2. x_i is the longest prefix of $v = x_i \ldots x_n$ that also has an occurrence beginning within $u = x_1 \ldots x_{i-1}$. See [1].

This is not a very good implementation, and should be improved.

EXAMPLES:

```
sage: x = Words('ab')('abababb')
sage: x.crochemore_factorization()
(a.b.abab.b)
sage: mul(x.crochemore_factorization()) == x
True
sage: y = Words('abc')('abaababacabba')
sage: y.crochemore_factorization()
(a.b.a.aba.ba.c.ab.ba)
sage: mul(y.crochemore_factorization()) == y
True
sage: x = Words([0, 1])([0,1,0,1,0,1,1])
sage: x.crochemore_factorization()
(0.1.0101.1)
sage: mul(x.crochemore_factorization()) == x
True
```

REFERENCES:

- [1] M. Crochemore, Recherche linéaire d'un carré dans un mot, C. R. Acad. Sci. Paris Sér. I Math. 296 (1983) 14 781-784.

defect (*f=None*)

Returns the defect of self.

The *defect* of a finite word w is given by $D(w) = |w| + 1 - |PAL(w)|$, where $PAL(w)$ denotes the set of palindromic factors of w (including the empty word). See [1].

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understands (dict, str, ...).

OUTPUT:

- `integer` - If f is None, the palindromic defect of self; otherwise, the f-palindromic defect of self.

EXAMPLES:

```
sage: words.ThueMorseWord()[:100].defect()
16
sage: words.FibonacciWord()[:100].defect()
0
sage: W = Words('01')
sage: W('000000000000').defect()
0
sage: W('011010011001').defect()
2
sage: W('0101001010001').defect()
0
sage: W().defect()
0
sage: Word('abbabaabbaababba').defect()
2
sage: Word('abbabaabbaababba').defect('a->b,b->a')
4
```

REFERENCES:

- [1] S. Brlek, S. Hamel, M. Nivat, C. Reutenauer, On the Palindromic Complexity of Infinite Words, in J. Berstel, J. Karhumaki, D. Perrin, Eds, Combinatorics on Words with Applications, International Journal of Foundation of Computer Science, Vol. 15, No. 2 (2004) 293-306.

deg_inv_lex_less (*other, weights=None*)

Returns True if the word self is degree inverse lexicographically less than other.

EXAMPLES:

```
sage: W = Words([1,2,3,4])
sage: W([1,2,4]).deg_inv_lex_less(W([1,3,2]))
False
sage: W([3,2,1]).deg_inv_lex_less(W([1,2,3]))
True
```

deg_lex_less (*other, weights=None*)

Returns True if self is degree lexicographically less than other, and False otherwise. The weight of each letter in the ordered alphabet is given by weights, which defaults to [1, 2, 3, ...].

EXAMPLES:

```
sage: Word([1,2,3]).deg_lex_less(Word([1,3,2]))
True
sage: Word([3,2,1]).deg_lex_less(Word([1,2,3]))
False
sage: W = Words(range(5))
sage: W([1,2,4]).deg_lex_less(W([1,3,2]))
False
sage: Word("abba").deg_lex_less(Word("abbb"))
True
sage: Word("abba").deg_lex_less(Word("baba"))
True
sage: Word("abba").deg_lex_less(Word("aaba"))
False
```

```
sage: Word("abba").deg_lex_less(Word("aaba"),[1,0])
True
```

deg_rev_lex_less(*other, weights=None*)

Returns True if self is degree reverse lexicographically less than other.

EXAMPLES:

```
sage: Word([3,2,1]).deg_rev_lex_less(Word([1,2,3]))
False
sage: W = Words([1,2,3,4])
sage: W([1,2,4]).deg_rev_lex_less(W([1,3,2]))
False
sage: W([1,2,3]).deg_rev_lex_less(W([1,2,4]))
True
```

degree(*weights=None*)

Returns the weighted degree of self, where the weighted degree of each letter in the ordered alphabet is given by weights, which defaults to [1, 2, 3, ...].

INPUTS: weights - a list or tuple, or a dictionary keyed by the letters occurring in self.

EXAMPLES:

```
sage: Word([1,2,3]).degree()
6
sage: Word([3,2,1]).degree()
6
sage: Word("abba").degree()
6
sage: Word("abba").degree([0,2])
4
sage: Word("abba").degree([-1,-1])
-4
sage: Word("aabba").degree([1,1])
5
sage: Words([1,2,4])([1,2,4]).degree()
6
sage: Words([1,2,3,4])([1,2,4]).degree()
7
sage: Word("aabba").degree({'a':1,'b':2})
7
sage: Word([0,1,0]).degree({0:17,1:0})
34
```

delta()

Returns the delta equivalent of self.

This is the word composed of the length of consecutive runs of the same letter in a given word.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('22112122').delta()
word: 22112
sage: W('555008').delta()
word: 321
sage: W().delta()
word:
```

delta_derivate(*W=None*)

Returns the derivative under delta for self.

EXAMPLES:

```
sage: W = Words('12')
sage: W('12211').delta_derivate()
word: 22
sage: W('1').delta_derivate(Words([1]))
word: 1
sage: W('2112').delta_derivate()
word: 2
sage: W('2211').delta_derivate()
word: 22
sage: W('112').delta_derivate()
word: 2
sage: W('11222').delta_derivate(Words([1, 2, 3]))
word: 3
```

delta_derivate_left(*W=None*)

Returns the derivative under delta for self.

EXAMPLES:

```
sage: W = Words('12')
sage: W('12211').delta_derivate_left()
word: 22
sage: W('1').delta_derivate_left(Words([1]))
word: 1
sage: W('2112').delta_derivate_left()
word: 21
sage: W('2211').delta_derivate_left()
word: 22
sage: W('112').delta_derivate_left()
word: 21
sage: W('11222').delta_derivate_left(Words([1, 2, 3]))
word: 3
```

delta_derivate_right(*W=None*)

Returns the right derivative under delta for self.

EXAMPLES:

```
sage: W = Words('12')
sage: W('12211').delta_derivate_right()
word: 122
sage: W('1').delta_derivate_right(Words([1]))
word: 1
sage: W('2112').delta_derivate_right()
word: 12
sage: W('2211').delta_derivate_right()
word: 22
sage: W('112').delta_derivate_right()
word: 2
sage: W('11222').delta_derivate_right(Words([1, 2, 3]))
word: 23
```

delta_inv(*W=None, s=None*)

Returns the inverse of the delta operator applied to self.

The letters in the returned word will start at the specified letter or the first one if None is specified (the default). The default alphabet is [1, 2].

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([2, 2, 1, 1]).delta_inv()
```

```
word: 112212
sage: W([1, 1, 1, 1]).delta_inv(Words('123'))
word: 1231
sage: W([2, 2, 1, 1, 2]).delta_inv(s=2)
word: 22112122
```

evaluation()

Returns a list a where a[i] is the number of occurrences in self of the i-th letter in self.alphabet().

NOTE: This is slightly different from self.parikh_vector() in that it truncates the list after the last nonzero output.

EXAMPLES:

```
sage: Words('ab')('aabaa').evaluation()
[4, 1]
sage: Words('bac')('aabaa').evaluation()
[1, 4]
sage: Words('bca')('aabaa').evaluation()
[1, 0, 4]
sage: Words('abc')('aabaa').evaluation()
[4, 1]
sage: Word('aabaacaccab').evaluation()
[6, 2, 3]
sage: Word().evaluation()
[]
sage: Words('abcde')('badbcdb').evaluation()
[1, 3, 1, 2]
sage: Word([1,2,2,1,3]).evaluation()
[2, 2, 1]
sage: P = Alphabet(name="positive integers")
sage: Words(P)([]).evaluation()
[]
```

evaluation_dict()

Returns a dictionary keyed by the letters occurring in self with values the number of occurrences of the letter.

EXAMPLES:

```
sage: Word([2,1,4,2,3,4,2]).evaluation_dict()
{1: 1, 2: 3, 3: 1, 4: 2}
sage: Word('badbcdb').evaluation_dict()
{'a': 1, 'c': 1, 'b': 3, 'd': 2}
sage: Word().evaluation_dict()
{}
```

evaluation_partition()

Returns the evaluation of the word w as a partition.

EXAMPLES:

```
sage: Word("acdabda").evaluation_partition()
[3, 2, 1, 1]
sage: Word([2,1,4,2,3,4,2]).evaluation_partition()
[3, 2, 1, 1]
```

evaluation_sparse()

Returns a list representing the evaluation of self. The entries of the list are two-element lists [a, n], where a is a letter occurring in self and n is the number of occurrences of a in self.

EXAMPLES:

```
sage: Word([4,4,2,5,2,1,4,1]).evaluation_sparse()
[(1, 2), (2, 2), (4, 3), (5, 1)]
sage: Words("acdb")("abcaccab").evaluation_sparse()
[('a', 3), ('c', 3), ('b', 2)]
```

exponent()

Returns the exponent of self.

OUTPUT:

- integer - the exponent

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('1231').exponent()
1
sage: W('121212').exponent()
3
sage: W().exponent()
0
```

factor_iterator(*n=None*)

Generates distinct factors of self.

INPUT:

- n - an integer, or None.

OUTPUT: If n is an integer, returns an iterator over all distinct factors of length n. If n is None, returns an iterator generating all distinct factors.

EXAMPLES:

```
sage: w = Words('123')('1213121')
sage: sorted( w.factor_iterator(0) )
[word: ]
sage: sorted( w.factor_iterator(10) )
[]
sage: sorted( w.factor_iterator(1) )
[word: 1, word: 2, word: 3]
sage: sorted( w.factor_iterator(4) )
[word: 1213, word: 1312, word: 2131, word: 3121]
sage: sorted( w.factor_iterator() )
[word: , word: 1, word: 12, word: 121, word: 1213, word: 12131, word: 121312, word: 1213121,

sage: u = Words([1,2,3])([1,2,1,2,3])
sage: sorted( u.factor_iterator(0) )
[word: ]
sage: sorted( u.factor_iterator(10) )
[]
sage: sorted( u.factor_iterator(1) )
[word: 1, word: 2, word: 3]
sage: sorted( u.factor_iterator(5) )
[word: 12123]
sage: sorted( u.factor_iterator() )
[word: , word: 1, word: 12, word: 121, word: 1212, word: 12123, word: 123, word: 2, word: 21

sage: xxx = Word("xxx")
sage: sorted( xxx.factor_iterator(0) )
[word: ]
sage: sorted( xxx.factor_iterator(4) )
[]
```

```
sage: sorted( xxx.factor_iterator(2) )
[word: xx]
sage: sorted( xxx.factor_iterator() )
[word: , word: x, word: xx, word: xxx]

sage: e = Word()
sage: sorted( e.factor_iterator(0) )
[word: ]
sage: sorted( e.factor_iterator(17) )
[]
sage: sorted( e.factor_iterator() )
[word: ]
```

TESTS:

```
sage: type( Words('cao')('cacao').factor_iterator() )
<type 'generator'>
```

factor_occurrences_in(*other*)

Returns an iterator over all occurrences (including overlapping ones) of self in other in their order of appearance.

EXAMPLES:

```
sage: W = Words('123')
sage: list(W('121').factor_occurrences_in(W('121213211213')))
[0, 2, 8]
```

factor_set()

Returns the set of factors of self.

EXAMPLES:

```
sage: Words('123')('1213121').factor_set()    # random
Set of elements of <generator object at 0xa8fde6c>
sage: sorted(  Words([1,2,3])([1,2,1,2,3]).factor_set()  )
[word: , word: 1, word: 12, word: 121, word: 1212, word: 12123, word: 123, word: 2, word: 21
sage: sorted(  Words("x")("xx").factor_set()  )
[word: , word: x, word: xx]
sage: set( Words([])().factor_set() )
set([word: ])
```

first_pos_in(*other*)

Returns the position of the first occurrence of self in other, or None if self is not a factor of other.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('12').first_pos_in(W('131231'))
2
sage: W('32').first_pos_in(W('131231')) is None
True
```

freq()

Returns a table of the frequencies of the letters in self.

OUTPUT:

- `dict` - letters associated to their frequency

EXAMPLES:

```
sage: Words('123')('1213121').freq()     # keys appear in random order
{'1': 4, '2': 2, '3': 1}
```

TESTS:

```
sage: f = Words('123')('1213121').freq()
sage: f['1'] == 4
True
sage: f['2'] == 2
True
sage: f['3'] == 1
True
```

good_suffix_table()

Returns a table of the maximum skip you can do in order not to miss a possible occurrence of self in a word.

This is a part of the Boyer-Moore algorithm to find factors. See [1].

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('121321').good_suffix_table()
[5, 5, 5, 5, 3, 3, 1]
sage: W('12412').good_suffix_table()
[3, 3, 3, 3, 3, 1]
```

REFERENCES:

- [1] R.S. Boyer, J.S. Moore, A fast string searching algorithm, Communications of the ACM 20 (1977) 762-772.

implicit_suffix_tree()

Returns the implicit suffix tree of self.

The *suffix tree* of a word w is a compactification of the suffix trie for w. The compactification removes all nodes that have exactly one incoming edge and exactly one outgoing edge. It consists of two components: a tree and a word. Thus, instead of labelling the edges by factors of w, we can labelled them by indices of the occurrence of the factors in w.

See sage.combinat.words.suffix_trees.ImplicitSuffixTree? for more information.

EXAMPLES:

```
sage: w = Words("cao")("cacao")
sage: w.implicit_suffix_tree()
Implicit Suffix Tree of the word: cacao

sage: w = Words([0,1])([0,1,0,1,1])
sage: w.implicit_suffix_tree()
Implicit Suffix Tree of the word: 01011
```

inv_lex_less(*other*)

Returns True if self is inverse lexicographically less than other.

EXAMPLES:

```
sage: W = Words([1,2,3,4])
sage: W([1,2,4]).inv_lex_less(W([1,3,2]))
False
sage: W([3,2,1]).inv_lex_less(W([1,2,3]))
True
```

inversions()

Returns a list of the inversions of self. An inversion is a pair (i,j) of non-negative integers i j such that self[i] self[j].

EXAMPLES:

```
sage: Words([1,2,3])([1,2,3,2,2,1]).inversions()
[[1, 5], [2, 3], [2, 4], [2, 5], [3, 5], [4, 5]]
sage: Words([3,2,1])([1,2,3,2,2,1]).inversions()
[[0, 1], [0, 2], [0, 3], [0, 4], [1, 2]]
sage: Words('ab')('abbaba').inversions()
[[1, 3], [1, 5], [2, 3], [2, 5], [4, 5]]
sage: Words('ba')('abbaba').inversions()
[[0, 1], [0, 2], [0, 4], [3, 4]]
```

is_balanced(*q=1*)

Returns True if self is q-balanced, and False otherwise.

A finite or infinite word w is said to be *'q'-balanced* if for any two factors u, v of w of the same length, the difference between the number of x's in each of u and v is at most q for all letters x in the alphabet of w. A 1-balanced word is simply said to be balanced. See for instance [1] and Chapter 2 of [2].

INPUT:

- q - integer (default 1), the balance level

OUTPUT:

- boolean - the result

EXAMPLES:

```
sage: Words('123')('1213121').is_balanced()
True
sage: Words('12')('1122').is_balanced()
False
sage: Words('123')('121333121').is_balanced()
False
sage: Words('123')('121333121').is_balanced(2)
False
sage: Words('123')('121333121').is_balanced(3)
True
sage: Words('12')('121122121').is_balanced()
False
sage: Words('12')('121122121').is_balanced(2)
True
sage: Words('12')('121122121').is_balanced(-1)
...
TypeError: the balance level must be a positive integer
sage: Words('12')('121122121').is_balanced(0)
...
TypeError: the balance level must be a positive integer
sage: Words('12')('121122121').is_balanced('a')
...
TypeError: the balance level must be a positive integer
```

REFERENCES:

- [1] J. Cassaigne, S. Ferenczi, L.Q. Zamboni, Imbalances in Arnoux-Rauzy sequences, Ann. Inst. Fourier (Grenoble) 50 (2000) 1265-1276.
- [2] M. Lothaire, Algebraic Combinatorics On Words, vol. 90 of Encyclopedia of Mathematics and its Applications, Cambridge University Press, U.K., 2002.

is_cadence(*seq*)

Returns True if seq is a cadence of self, and False otherwise.

A *cadence* is an increasing sequence of indexes that all map to the same letter.

EXAMPLES:

```
sage: W = Words('123')
sage: W('121132123').is_cadence([0, 2, 6])
True
sage: W('121132123').is_cadence([0, 1, 2])
False
sage: W('121132123').is_cadence([])
True
```

is_conjugate_with(*other*)

Returns True if self is a conjugate of other, and False otherwise.

EXAMPLES:

```
sage: W = Words('123')
sage: W('11213').is_conjugate_with(W('31121'))
True
sage: W().is_conjugate_with(W('123'))
False
sage: W('112131').is_conjugate_with(W('11213'))
False
sage: W('12131').is_conjugate_with(W('11213'))
True
```

is_cube()

Returns True if self is a cube, and False otherwise.

EXAMPLES:

```
sage: W = Words('012')
sage: W('012012012').is_cube()
True
sage: W('01010101').is_cube()
False
sage: W().is_cube()
True
sage: W('012012').is_cube()
False
```

is_cube_free()

Returns True if self does not contain cubes, and False otherwise.

EXAMPLES:

```
sage: W = Words('123')
sage: W('12312').is_cube_free()
True
sage: W('32221').is_cube_free()
False
sage: W().is_cube_free()
True
```

is_empty()

Returns True if the length of self is zero, and False otherwise.

EXAMPLES:

```
sage: W=Words('ab')
sage: W().is_empty()
True
sage: W('a').is_empty()
False
```

is_factor_of(*other*)

Returns True if self is a factor of other, and False otherwise.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('2113').is_factor_of(W('12312133213123321132123'))
True
sage: W('321').is_factor_of(W('12312412313123123312'))
False
```

is_full (*f=None*)

Returns True if self has defect 0, and False otherwise.

A word is *full* if its defect is zero (see [1]).

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understands (dict, str, ...).

OUTPUT:

- `boolean` - If f is None, whether self is full; otherwise, whether self is full of f-palindromes.

EXAMPLES:

```
sage: words.ThueMorseWord()[:100].is_full()
False
sage: words.FibonacciWord()[:100].is_full()
True
sage: Words('0')('000000000000000').is_full()
True
sage: Words('01')('011010011001').is_full()
False
sage: Words('123456789')('2194').is_full()
True
sage: Word().is_full()
True
sage: Word().is_full('a->b,b->a')
True
sage: w = Word('ab')
sage: w.is_full()
True
sage: w.is_full('a->b,b->a')
False
```

REFERENCES:

- [1] S. Brlek, S. Hamel, M. Nivat, C. Reutenauer, On the Palindromic Complexity of Infinite Words, in J. Berstel, J. Karhumaki, D. Perrin, Eds, Combinatorics on Words with Applications, International Journal of Foundation of Computer Science, Vol. 15, No. 2 (2004) 293-306.

is_lyndon()

Returns True if self is a Lyndon word, and False otherwise.

A *Lyndon word* is a non-empty word that is lexicographically smaller than all of its proper suffixes for the given order on its alphabet. That is, w is a Lyndon word if w is non-empty and for each factorization $w = uv$ (with u, v both non-empty), we have $w < v$.

Equivalently, w is a Lyndon word iff w is a non-empty word that is lexicographically smaller than all of its proper conjugates for the given order on its alphabet.

See for instance [1].

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('123132133').is_lyndon()
True
```

```
sage: W().is_lyndon()
True
sage: W('122112').is_lyndon()
False
```

REFERENCES:

- [1] M. Lothaire, Combinatorics On Words, vol. 17 of Encyclopedia of Mathematics and its Applications, Addison-Wesley, Reading, Massachusetts, 1983.

is_overlap()

Returns True if self is an overlap, and False otherwise.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('12121').is_overlap()
True
sage: W('123').is_overlap()
False
sage: W('1231').is_overlap()
False
sage: W('123123').is_overlap()
False
sage: W('1231231').is_overlap()
True
sage: W().is_overlap()
False
```

is_palindrome(*f=None*)

Returns True if self is a palindrome (or a f -palindrome), and False otherwise.

- In French: Soit $f : \Sigma \to \Sigma$ une involution qui s'étend évidemment à un morphisme sur Σ^*. On dit que $w \in \Sigma^*$ est un *'f'-pseudo-palindrome* [1], ou plus simplement un *'f'-palindrome*, si $w = f(\tilde{w})$ (extrait de [2]).
- In English Let $f : \Sigma \to \Sigma$ be an involution that extends to a morphism on Σ^*. We say that $w \in \Sigma^*$ is a *'f'-palindrome* if $w = f(\tilde{w})$ [2].

Also called *'f' -pseudo-palindrome* [1].

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understands (dict, str, ...).

OUTPUT:

- `boolean` - if f is None, whether self is a palindrome; otherwise, whether self is a f-palindrome.

EXAMPLES: Some palindromes...

```
sage: W=Words('abcdefghijklmnopqrstuvwxyz I')
sage: W('esope reste ici et se repose').is_palindrome()
False
sage: W('esoperesteicietserepose').is_palindrome()
True
sage: W('I saw I was I').is_palindrome()
True
sage: Word('abbcbba').is_palindrome()
True
sage: Word('abcbdba').is_palindrome()
False
```

Some f-palindromes... You can use a str:

```
sage: Word('aababb').is_palindrome('a->b,b->a')
True
sage: Word('abacbacbab').is_palindrome('a->b,b->a,c->c')
True
```

You can use WordMorphism:

```
sage: f = WordMorphism('a->b,b->a')
sage: Word('aababb').is_palindrome(f)
True
```

You can use a dictionary:

```
sage: f = {'a':'b','b':'a'}
sage: Word('aababb').is_palindrome(f)
True
sage: w = words.ThueMorseWord()[:8]; w
word: 01101001
sage: w.is_palindrome(f={0:[1],1:[0]})
True
```

self must be in the domain of the involution:

```
sage: f = WordMorphism('a->a')
sage: Word('aababb').is_palindrome(f)
...
ValueError: self must be in the domain of the given involution
```

The given involution must be an involution:

```
sage: f = WordMorphism('a->b,b->b')
sage: Word('abab').is_palindrome(f)
...
ValueError: f must be an involution
```

TESTS:

```
sage: Y = Words('ab')
sage: Y().is_palindrome()
True
sage: Y('a').is_palindrome()
True
sage: Y('ab').is_palindrome()
False
sage: Y('aba').is_palindrome()
True
sage: Y('aa').is_palindrome()
True
sage: E = 'a->b,b->a'
sage: Y().is_palindrome(E)
True
sage: Y('a').is_palindrome(E)
False
sage: Y('ab').is_palindrome(E)
True
sage: Y('aa').is_palindrome(E)
False
sage: Y('aba').is_palindrome(E)
False
sage: Y('abab').is_palindrome(E)
True
```

REFERENCES:

- [1] V. Anne, L.Q. Zamboni, I. Zorca, Palindromes and Pseudo- Palindromes in Episturmian and Pseudo-Palindromic Infinite Words, in : S. Brlek, C. Reutenauer (Eds.), Words 2005, Publications du LaCIM, Vol. 36 (2005) 91-100.
- [2] S. Labbé, Propriétés combinatoires des f-palindromes, Mémoire de maîtrise en Mathématiques, Montréal, UQAM, 2008, 109 pages.

is_prefix_of(*other*)

Returns True if self is a prefix of other, and False otherwise.

EXAMPLES:

```
sage: V = Words('0123456789')
sage: w = V('0123456789')
sage: y = V('012345')
sage: y.is_prefix_of(w)
True
sage: w.is_prefix_of(y)
False
sage: W = Words('ab')
sage: w.is_prefix_of(W())
False
sage: W().is_prefix_of(w)
True
sage: W().is_prefix_of(W())
True
```

is_primitive()

Returns True if self is primitive, and False otherwise.

A finite word w is *primitive* if it is not a positive integer power of a shorter word.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('1231').is_primitive()
True
sage: W('111').is_primitive()
False
```

is_proper_prefix_of(*other*)

Returns True if self is a proper prefix of other, and False otherwise.

EXAMPLES:

```
sage: W = Words('123')
sage: W('12').is_proper_prefix_of(W('123'))
True
sage: W('12').is_proper_prefix_of(W('12'))
False
sage: W().is_proper_prefix_of(W('123'))
True
sage: W('123').is_proper_prefix_of(W('12'))
False
sage: W().is_proper_prefix_of(W())
False
```

is_proper_suffix_of(*other*)

Returns True if self is a proper suffix of other, and False otherwise.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('23').is_proper_suffix_of(W('123'))
True
```

```
sage: W('12').is_proper_suffix_of(W('12'))
False
sage: W().is_proper_suffix_of(W('123'))
True
sage: W('123').is_proper_suffix_of(W('12'))
False
```

is_quasiperiodic()

Returns True if self is quasiperiodic, and False otherwise.

A finite or infinite word w is *quasiperiodic* if it can be constructed by concatenations and superpositions of one of its proper factors u, which is called a *quasiperiod* of w. See for instance [1], [2], and [3].

EXAMPLES:

```
sage: W = Words('abc')
sage: W('abaababaabaababaaba').is_quasiperiodic()
True
sage: W('abacaba').is_quasiperiodic()
False
sage: W('a').is_quasiperiodic()
False
sage: W().is_quasiperiodic()
False
sage: W('abaaba').is_quasiperiodic()
True
```

REFERENCES:

- [1] A. Apostolico, A. Ehrenfeucht, Efficient detection of quasiperiodicities in strings, Theoret. Comput. Sci. 119 (1993) 247-265.
- [2] S. Marcus, Quasiperiodic infinite words, Bull. Eur. Assoc. Theor. Comput. Sci. 82 (2004) 170-174. [3] A. Glen, F. Levé, G. Richomme, Quasiperiodic and Lyndon episturmian words, Preprint, 2008, arXiv:0805.0730.

is_smooth_prefix()

Returns True if self is the prefix of a smooth word, and False otherwise.

Let $A_k = \{1, \ldots, k\}$, $k \geq 2$. An infinite word w in A_k^ω is said to be *smooth* if and only if for all positive integers m, $\Delta^m(w)$ is in A_k^ω, where $\Delta(w)$ is the word obtained from w by composing the length of consecutive runs of the same letter in w. See for instance [1] and [2].

INPUT:

- `self` - must be a word over the integers to get something other than False

OUTPUT:

- `boolean` - whether self is a smooth prefix or not

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([1, 1, 2, 2, 1, 2, 1, 1]).is_smooth_prefix()
True
sage: W([1, 2, 1, 2, 1, 2]).is_smooth_prefix()
False
```

REFERENCES:

- [1] S. Brlek, A. Ladouceur, A note on differentiable palindromes, Theoret. Comput. Sci. 302 (2003) 167-178.
- [2] S. Brlek, S. Dulucq, A. Ladouceur, L. Vuillon, Combinatorial properties of smooth infinite words, Theoret. Comput. Sci. 352 (2006) 306-317.

is_square()
Returns True if self is a square, and False otherwise.
EXAMPLES:

```
sage: W = Words('123')
sage: W('1212').is_square()
True
sage: W('1213').is_square()
False
sage: W('12123').is_square()
False
sage: W().is_square()
True
```

is_square_free()
Returns True if self does not contain squares, and False otherwise.
EXAMPLES:

```
sage: W = Words('123')
sage: W('12312').is_square_free()
True
sage: W('31212').is_square_free()
False
sage: W().is_square_free()
True
```

is_subword_of(*other*)
Returns True is self is a subword of other, and False otherwise.
EXAMPLES:

```
sage: W = Words('123')
sage: W().is_subword_of(W('123'))
True
sage: W('123').is_subword_of(W('3211333213233321'))
True
sage: W('321').is_subword_of(W('11122212112122133111222332'))
False
```

is_suffix_of(*other*)
Returns True if w is a suffix of other, and False otherwise.
EXAMPLES:

```
sage: W = Words('0123456789')
sage: w = W('0123456789')
sage: y = W('56789')
sage: y.is_suffix_of(w)
True
sage: w.is_suffix_of(y)
False
sage: W('579').is_suffix_of(w)
False
sage: W().is_suffix_of(y)
True
sage: w.is_suffix_of(W())
False
sage: W().is_suffix_of(W())
True
```

is_symmetric(*f=None*)
Returns True if self is symmetric (or f-symmetric), and False otherwise.

A word is *symmetric* (resp. *'f'-symmetric*) if it is the product of two palindromes (resp. f-palindromes). See [1] and [2].

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understands (dict, str, ...).

OUTPUT:

- `boolean` - If f is None, whether self is symmetric; otherwise, whether self is f-symmetric.

EXAMPLES:

```
sage: W = Words('ab')
sage: W('abbabab').is_symmetric()
True
sage: W('ababa').is_symmetric()
True
sage: W('aababaabba').is_symmetric()
False
sage: W('aabbbaababba').is_symmetric()
False
sage: W('aabbbaababba').is_symmetric('a->b,b->a')
True
```

REFERENCES:

- [1] S. Brlek, S. Hamel, M. Nivat, C. Reutenauer, On the Palindromic Complexity of Infinite Words, in J. Berstel, J. Karhumaki, D. Perrin, Eds, Combinatorics on Words with Applications, International Journal of Foundation of Computer Science, Vol. 15, No. 2 (2004) 293-306.
- [2] A. de Luca, A. De Luca, Pseudopalindrome closure operators in free monoids, Theoret. Comput. Sci. 362 (2006) 282-300.

iterated_palindromic_closure(*side='right', f=None*)

Returns the iterated (f-)palindromic closure of self.

INPUT:

- `side` - 'right' or 'left' (default: 'right') the direction of the closure
- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understands (dict, str, ...).

OUTPUT:

- `word` - If f is None, the right iterated palindromic closure of self; otherwise, the right iterated f-palindromic closure of self. If side is 'left', the left palindromic closure.

EXAMPLES:

```
sage: W = Words('123')
sage: W('123').iterated_palindromic_closure()
word: 1213121
sage: W('123').iterated_palindromic_closure(side='left')
word: 3231323
sage: W('1').iterated_palindromic_closure()
word: 1
sage: W().iterated_palindromic_closure()
word:
sage: W=Words('ab')
sage: W('ab').iterated_palindromic_closure(f='a->b,b->a')
word: abbaab
sage: W('ab').iterated_palindromic_closure(f='a->b,b->a',side='left')
word: abbaab
sage: W('aab').iterated_palindromic_closure(f='a->b,b->a')
word: ababbaabab
```

```
sage: W('aab').iterated_palindromic_closure(f='a->b,b->a',side='left')
word: abbaabbaab
```

TESTS:

```
sage: W('aab').iterated_palindromic_closure(f='a->b,b->a',side='leftt')
...
ValueError: side must be either 'left' or 'right' (not leftt)
sage: W('aab').iterated_palindromic_closure(f='a->b,b->b',side='left')
...
ValueError: f must be an involution
```

REFERENCES:

- [1] A. de Luca, A. De Luca, Pseudopalindrome closure operators in free monoids, Theoret. Comput. Sci. 362 (2006) 282-300.

lacunas (*f=None*)

Returns the list of all the lacunas of self.

A *lacuna* is a position in a word where the longest palindromic suffix is not unioccurrent (see [1]).

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understands (dict, str, ...).

OUTPUT:

- `list` - list of all the lacunas of self.

EXAMPLES:

```
sage: words.ThueMorseWord()[:100].lacunas()
[8, 9, 24, 25, 32, 33, 34, 35, 36, 37, 38, 39, 96, 97, 98, 99]
sage: words.ThueMorseWord()[:50].lacunas(f={0:[1],1:[0]})
[0, 2, 4, 12, 16, 17, 18, 19, 48, 49]
```

REFERENCES:

- [1] A. Blondin-Massé, S. Brlek, S. Labbé, Palindromic lacunas of the Thue-Morse word, Proc. GASCOM 2008 (June 16-20 2008, Bibbiena, Arezzo-Italia), 53-67.

last_position_table()

Returns a table (of size 256) that contains the last position of each letter in self. The letters not present in the word will have a position of None.

EXAMPLES:

```
sage: Words('01234')('1231232').last_position_table()
[-1, 3, 6, 5, -1]
```

length_border()

Returns the length of the border of self.

The *border* of a word is the longest word that is both a proper prefix and a proper suffix of self.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('121').length_border()
1
sage: W('1').length_border()
0
sage: W('1212').length_border()
2
sage: W('111').length_border()
2
sage: W().length_border() is None
True
```

lengths_lps (*f=None*)

Returns the list of the length of the longest palindromic suffix (lps) for each non-empty prefix of self.

It corresponds to the function G_w defined in [2].

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understand (dict, str, ...).

OUTPUT:

- `list` - list of the length of the longest palindromic suffix (lps) for each non-empty prefix of self.

EXAMPLES:

```
sage: Word().lengths_lps()
[]
sage: Word('a').lengths_lps()
[1]
sage: Word('aaa').lengths_lps()
[1, 2, 3]
sage: Word('abbabaabbaab').lengths_lps()
[1, 1, 2, 4, 3, 3, 2, 4, 2, 4, 6, 8]
sage: Word('abbabaabbaab').lengths_lps(f='a->b,b->a')
[0, 2, 0, 2, 2, 4, 6, 8, 4, 6, 4, 6]
sage: Word([5,8,5,5,8,8,5,5,8,8,5,8,5]).lengths_lps(f={5:[8],8:[5]})
[0, 2, 2, 0, 2, 4, 6, 4, 6, 8, 10, 12, 4]
```

REFERENCES:

- [1] A. Blondin-Massé, S. Brlek, S. Labbé, Palindromic lacunas of the Thue-Morse word, Proc. GASCOM 2008 (June 16-20 2008, Bibbiena, Arezzo-Italia), 53-67.
- [2] A. Blondin-Massé, S. Brlek, A. Frosini, S. Labbé, S. Rinaldi, Reconstructing words from a fixed palindromic length sequence, Proc. TCS 2008, 5th IFIP International Conference on Theoretical Computer Science (September 8-10 2008, Milano, Italia), accepted.

lengths_unioccurrent_lps (*f=None*)

Returns the list of the lengths of the unioccurrent longest palindromic suffixes (lps) for each non-empty prefix of self. No unioccurrent lps are indicated by None.

It corresponds to the function H_w defined in [1] and [2].

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understand (dict, str, ...).

OUTPUT:

- `list` - list of the length of the unioccurrent longest palindromic suffix (lps) for each non-empty prefix of self. No unioccurrent lps are indicated by None.

EXAMPLES:

```
sage: f = words.FibonacciWord()[:20]
sage: f.lengths_unioccurrent_lps() == f.lengths_lps()
True
sage: words.ThueMorseWord()[:20].lengths_unioccurrent_lps()
[1, 1, 2, 4, 3, 3, 2, 4, None, None, 6, 8, 10, 12, 14, 16, 6, 8, 10, 12]
sage: words.ThueMorseWord()[:15].lengths_unioccurrent_lps(f={1:[0],0:[1]})
[None, 2, None, 2, None, 4, 6, 8, 4, 6, 4, 6, None, 4, 6]
```

REFERENCES:

- [1] A. Blondin-Massé, S. Brlek, S. Labbé, Palindromic lacunas of the Thue-Morse word, Proc. GASCOM 2008 (June 16-20 2008, Bibbiena, Arezzo-Italia), 53-67.

- [2] A. Blondin-Massé, S. Brlek, A. Frosini, S. Labbé, S. Rinaldi, Reconstructing words from a fixed palindromic length sequence, Proc. TCS 2008, 5th IFIP International Conference on Theoretical Computer Science (September 8-10 2008, Milano, Italia), accepted.

lex_greater(*other*)

Returns True if self is lexicographically greater than other.

EXAMPLES:

```
sage: w = Word([1,2,3])
sage: u = Word([1,3,2])
sage: v = Word([3,2,1])
sage: w.lex_greater(u)
False
sage: v.lex_greater(w)
True
sage: a = Word("abba")
sage: b = Word("abbb")
sage: a.lex_greater(b)
False
sage: b.lex_greater(a)
True
```

lex_less(*other*)

Returns True if self is lexicographically less than other.

EXAMPLES:

```
sage: w = Word([1,2,3])
sage: u = Word([1,3,2])
sage: v = Word([3,2,1])
sage: w.lex_less(u)
True
sage: v.lex_less(w)
False
sage: a = Word("abba")
sage: b = Word("abbb")
sage: a.lex_less(b)
True
sage: b.lex_less(a)
False
```

longest_common_prefix(*other*)

Returns the longest common prefix of self and other.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: w=W('12345')
sage: y=W('1236777')
sage: w.longest_common_prefix(y)
word: 123
sage: w.longest_common_prefix(w)
word: 12345
sage: y.longest_common_prefix(w)
word: 123
sage: y.longest_common_prefix(w)==w.longest_common_prefix(y)
True
sage: y.longest_common_prefix(y)==y
True
sage: W().longest_common_prefix(w)
word:
sage: w.longest_common_prefix(W()) == w
```

```
False
sage: w.longest_common_prefix(W()) == W()
True
sage: w.longest_common_prefix(w[:3]) == w[:3]
True
sage: w.longest_common_prefix(w[:3]) == w
False
sage: Words('12')("11").longest_common_prefix(Words('1')("1"))
word: 1
```

longest_common_suffix(*other*)

Returns the longest common suffix of self and other.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: y = W('549332345')
sage: w = W('203945')
sage: w.longest_common_suffix(y)
word: 45
sage: w.longest_common_suffix(y.reversal())
word: 3945
sage: w.longest_common_suffix(W())
word:
sage: W().longest_common_suffix(w)
word:
sage: W().longest_common_suffix(W())
word:
sage: w.longest_common_suffix(w[3:]) == w[3:]
True
sage: w.longest_common_suffix(w[3:]) == w
False
```

lps(*f=None*)

Returns the longest palindromic (or f-palindromic) suffix of self.

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understands (dict, str, ...).

OUTPUT:

- `word` - If f is None, the longest palindromic suffix of self; otherwise, the longest f-palindromic suffix of self.

EXAMPLES:

```
sage: Word('0111').lps()
word: 111
sage: Word('011101').lps()
word: 101
sage: Word('6667').lps()
word: 7
sage: Word('abbabaab').lps()
word: baab
sage: Word().lps()
word:
sage: Word('abbabaab').lps('a->b,b->a')
word: abbabaab
```

lyndon_factorization()

Returns the Lyndon factorization of self.

The *Lyndon factorization* of a finite word w is the unique factorization of w as a non-increasing product of Lyndon words, i.e., $w = l_1 \cdots l_n$ where each l_i is a Lyndon word and $l_1 \geq \cdots \geq l_n$. See for instance [1].

OUTPUT:

- `list` - the list of factors obtained

EXAMPLES:

```
sage: Words('01')('010010010001000').lyndon_factorization()
(01.001.001.0001.0.0.0)
sage: Words('10')('010010010001000').lyndon_factorization()
(0.10010010001000)
sage: Words('ab')('abbababbaababba').lyndon_factorization()
(abb.ababb.aababb.a)
sage: Words('ba')('abbababbaababba').lyndon_factorization()
(a.bbababbaaba.bba)
```

TESTS:

```
sage: Words('01')('01').lyndon_factorization()
(01)
sage: Words('10')('01').lyndon_factorization()
(0.1)
sage: lynfac = Words('ab')('abbababbaababba').lyndon_factorization()
sage: map(lambda x:x.is_lyndon(), lynfac)
[True, True, True, True]
sage: lynfac = Words('ba')('abbababbaababba').lyndon_factorization()
sage: map(lambda x:x.is_lyndon(), lynfac)
[True, True, True]
```

REFERENCES:

- [1] J.-P. Duval, Factorizing words over an ordered alphabet, J. Algorithms 4 (1983) 363-381.

minimal_period()

Returns the period of self.

Let A be an alphabet. An integer $p \geq 1$ is a *period* of a word $w = a_1 a_2 \cdots a_n$ where $a_i \in A$ if $a_i = a_{i+p}$ for $i = 1, \ldots, n-p$. The smallest period of w is called *the* period of w. See Chapter 1 of [1].

EXAMPLES:

```
sage: Word('aba').minimal_period()
2
sage: Word('abab').minimal_period()
2
sage: Word('ababa').minimal_period()
2
sage: Word('ababaa').minimal_period()
5
sage: Word('ababac').minimal_period()
6
sage: Word('aaaaaa').minimal_period()
1
sage: Word('a').minimal_period()
1
sage: Word().minimal_period()
1
```

REFERENCES:

- [1] M. Lothaire, Algebraic Combinatorics On Words, vol. 90 of Encyclopedia of Mathematics and its Applications, Cambridge University Press, U.K., 2002.

nb_factor_occurrences_in(*other*)

Returns the number of times self appears as a factor in other.

EXAMPLES:

```
sage: W = Words('123')
sage: W().nb_factor_occurrences_in(W('123'))
...
NotImplementedError: undefined value
sage: W('123').nb_factor_occurrences_in(W('112332312313112332121123'))
4
sage: W('321').nb_factor_occurrences_in(W('112332312313112332211123'))
0
```

nb_subword_occurrences_in(*other*)

Returns the number of times self appears in other as a subword.

EXAMPLES:

```
sage: W = Words('1234')
sage: W().nb_subword_occurrences_in(W('123'))
Traceback (most recent call last):
  ...
NotImplementedError: undefined value
sage: W('123').nb_subword_occurrences_in(W('1133432311132311112'))
11
sage: W('4321').nb_subword_occurrences_in(W('113223111223321234223111 2'))
0
sage: W('3').nb_subword_occurrences_in(W('122332112321213'))
4
```

number_of_factors(*n=None*)

Counts the number of distinct factors of self.

INPUT:

- n - an integer, or None.

OUTPUT: If n is an integer, returns the number of distinct factors of length n. If n is None, returns the total number of distinct factors.

EXAMPLES:

```
sage: w = Words([1,2,3])([1,2,1,2,3])
sage: w.number_of_factors()
13
sage: map(w.number_of_factors, range(6))
[1, 3, 3, 3, 2, 1]

sage: Words('123')('1213121').number_of_factors()
22
sage: Words('123')('1213121').number_of_factors(1)
3

sage: Words('a')('a'*100).number_of_factors()
101
sage: Words('a')('a'*100).number_of_factors(77)
1

sage: Words([])().number_of_factors()
1
sage: Words([])().number_of_factors(17)
0

sage: blueberry = Word("blueberry")
sage: blueberry.number_of_factors()
```

```
43
sage: map(blueberry.number_of_factors, range(10))
[1, 6, 8, 7, 6, 5, 4, 3, 2, 1]
```

order()

Returns the order of self.

Let $p(w)$ be the period of a word w. The positive rational number $|w|/p(w)$ is the *order* of w. See Chapter 8 of [1].

OUTPUT:

- `rational` - the order

EXAMPLES:

```
sage: Word('abaaba').order()
2
sage: Word('ababaaba').order()
8/5
sage: Word('a').order()
1
sage: Word('aa').order()
2
sage: Word().order()
0
```

REFERENCES:

- [1] M. Lothaire, Algebraic Combinatorics On Words, vol. 90 of Encyclopedia of Mathematics and its Applications, Cambridge University Press, U.K., 2002.

overlap_partition(*other, delay=0, p=None*)

Returns the partition of the alphabet induced by the equivalence relation defined below.

Let $u = u_0u_1\cdots u_{n-1}$, $v = v_0v_1\cdots v_{m-1}$ be two words on the alphabet A where u_i, v_j are letters and let d be an integer. We define a relation $R_{u,v,d} \subset AimesA$ by $R_{u,v,d} = \{(u_k, v_{k-d}) : 0 \leq k < n, 0 \leq k-d < m\}$. The equivalence relation returned is the symmetric, reflexive and transitive closure of $R_{self,other,delay} \cup p$ (inspired from [1]).

EXAMPLE ILLUSTRATING THE PRECEDENT DEFINITION: Let $A = \{a, b, c, d, e, f, h, l, v\}$, $u = cheval$, $v = abcdef$ and $d = 3$. Then $0 \leq k < 6$ and $0 \leq k - 3 < 6$ implies that $3 \leq k \leq 5$. Then,

$$R_{u,v,d} = \{(u_3, v_0), (u_4, v_1), (u_5, v_2)\} = \{(v, a), (a, b), (l, c)\}$$

These three couples correspond to the pairs of letters one above the other in the following overlap:

```
cheval
   abcdef
```

The symmetric, reflexive and transitive closure of $R_{u,v,d}$ defines the following partition of the alphabet A:

$$\{\{a, b, v\}, \{c, l\}, \{d\}, \{e\}, \{f\}, \{h\}\}.$$

INPUT:

- `other` - word on the same alphabet as self
- `delay` - integer
- `p` - Set (default: None), a partition of the alphabet

OUTPUT:

- `p` - Set, a set partition of the alphabet of self and other.

EXAMPLES:

```
sage: W = Words('abcdefhlv')
sage: cheval = W('cheval')
sage: abcdef = W('abcdef')
sage: p = cheval.overlap_partition(abcdef,3); p
{{'f'}, {'e'}, {'d'}, {'a', 'b', 'v'}, {'c', 'l'}, {'h'}}
sage: cheval.overlap_partition(abcdef,2,p)
{{'f'}, {'a', 'c', 'b', 'e', 'd', 'v', 'l'}, {'h'}}
sage: W = Words('abcdef')
sage: w = W('abc')
sage: y = W('def')
sage: w.overlap_partition(y, -3)
{{'f'}, {'e'}, {'d'}, {'b'}, {'a'}, {'c'}}
sage: w.overlap_partition(y, -2)
{{'a', 'f'}, {'e'}, {'d'}, {'c'}, {'b'}}
sage: w.overlap_partition(y, -1)
{{'a', 'e'}, {'d'}, {'c'}, {'b', 'f'}}
sage: w.overlap_partition(y, 0)
{{'b', 'e'}, {'a', 'd'}, {'c', 'f'}}
sage: w.overlap_partition(y, 1)
{{'c', 'e'}, {'f'}, {'b', 'd'}, {'a'}}
sage: w.overlap_partition(y, 2)
{{'f'}, {'e'}, {'b'}, {'a'}, {'c', 'd'}}
sage: w.overlap_partition(y, 3)
{{'f'}, {'e'}, {'d'}, {'b'}, {'a'}, {'c'}}
sage: w.overlap_partition(y, 4)
{{'f'}, {'e'}, {'d'}, {'b'}, {'a'}, {'c'}}
sage: W = Words(range(2))
sage: w = W([0,1,0,1,0,1]); w
word: 010101
sage: w.overlap_partition(w, 0)
{{0}, {1}}
sage: w.overlap_partition(w, 1)
{{0, 1}}
```

TESTS:

```
sage: Word().overlap_partition(Word(),'yo')
...
TypeError: delay (type given: <type 'str'>) must be an integer
sage: Word().overlap_partition(Word(),2,'yo')
...
TypeError: p(=yo) is not a Set
sage: Word('a').overlap_partition(Word('b'),0)
...
TypeError: no coercion rule between Ordered Alphabet ['a'] and Ordered Alphabet ['b']
```

REFERENCES:

•[1] S. Labbé, Propriétés combinatoires des f-palindromes, Mémoire de maîtrise en Mathématiques, Montréal, UQAM, 2008, 109 pages.

palindromes(*f=None*)

Returns the set of all palindromic (or f-palindromic) factors of self.

INPUT:

•`f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understands (dict, str, ...).

OUTPUT:

•`set` - If f is None, the set of all palindromic factors of self; otherwise, the set of all f-palindromic factors of self.

EXAMPLES:

```
sage: W = Words('01')
sage: sorted(W('01101001').palindromes())
[word: , word: 0, word: 00, word: 010, word: 0110, word: 1, word: 1001, word: 101, word: 11]
sage: sorted(W('00000').palindromes())
[word: , word: 0, word: 00, word: 000, word: 0000, word: 00000]
sage: sorted(W('0').palindromes())
[word: , word: 0]
sage: sorted(W('').palindromes())
[word: ]
sage: sorted(W().palindromes())
[word: ]
sage: sorted(Word('abbabaab').palindromes('a->b,b->a'))
[word: , word: ab, word: abbabaab, word: ba, word: baba, word: bbabaa]
```

palindromic_closure(*side='right', f=None*)

Returns the shortest palindrome having self as a prefix (or as a suffix if side=='left').

Retourne le plus petit palindrome ayant self comme prefixe (ou comme suffixe si side=='left').

INPUT:

- `side` - 'right' or 'left' (default: 'right') the direction of the closure
- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understands (dict, str, ...).

OUTPUT:

- `word` - If f is None, the right palindromic closure of self; otherwise, the right f-palindromic closure of self. If side is 'left', the left palindromic closure.

EXAMPLES:

```
sage: W = Words('1234567890')
sage: W('1233').palindromic_closure()
word: 123321
sage: W('12332').palindromic_closure()
word: 123321
sage: W('0110343').palindromic_closure()
word: 01103430110
sage: W('0110343').palindromic_closure(side='left')
word: 3430110343
sage: W('01105678').palindromic_closure(side='left')
word: 876501105678
sage: w = Word('abbaba')
sage: w.palindromic_closure()
word: abbababba
sage: w.palindromic_closure(f='a->b,b->a')
word: abbabaab
sage: w.palindromic_closure(f='a->b,b->a',side='left')
word: babaabbaba
sage: w.palindromic_closure(f='a->b,b->b',side='left')
...
ValueError: f must be an involution
sage: w.palindromic_closure(f='a->c,c->a',side='left')
...
ValueError: self must be in the domain of the given involution
```

REFERENCES:

- [1] A. de Luca, A. De Luca, Pseudopalindrome closure operators in free monoids, Theoret. Comput. Sci. 362 (2006) 282-300.

palindromic_lacunas_study(*f=None*)

Returns interesting statistics about longest (f-)palindromic suffixes and lacunas of self (see [1] and [2]).

Note that a word w has at most $|w| + 1$ different palindromic factors (see [4]).

INPUT:

- `f` - involution (default: None) on the alphabet of self. It must be something that WordMorphism's constructor understand (dict, str, ...).

OUTPUT:

- `list` - list of the length of the longest palindromic suffix (lps) for each non-empty prefix of self;
- `list` - list of all the lacunas, i.e. positions where there is no unioccurrent lps;
- `set` - set of palindromic factors of self.

EXAMPLES:

```
sage: W=Words('ab')
sage: a,b,c = W('abbabaabbaab').palindromic_lacunas_study()
sage: a
[1, 1, 2, 4, 3, 3, 2, 4, 2, 4, 6, 8]
sage: b
[8, 9]
sage: c           # random order
set([word: , word: b, word: bab, word: abba, word: bb, word: aa, word: baabbaab, word: baab,
sage: a,b,c = W('abbabaab').palindromic_lacunas_study(f='a->b,b->a')
sage: a
[0, 2, 0, 2, 2, 4, 6, 8]
sage: b
[0, 2, 4]
sage: c            # random order
set([word: , word: ba, word: baba, word: ab, word: bbabaa, word: abbabaab])
sage: c == set([W(), W('ba'), W('baba'), W('ab'), W('bbabaa'), W('abbabaab')])
True
```

REFERENCES:

- [1] A. Blondin-Massé, S. Brlek, S. Labbé, Palindromic lacunas of the Thue-Morse word, Proc. GASCOM 2008 (June 16-20 2008, Bibbiena, Arezzo-Italia), 53-67.
- [2] A. Blondin-Massé, S. Brlek, A. Frosini, S. Labbé, S. Rinaldi, Reconstructing words from a fixed palindromic length sequence, Proc. TCS 2008, 5th IFIP International Conference on Theoretical Computer Science (September 8-10 2008, Milano, Italia), accepted.
- [3] S. Labbé, Propriétés combinatoires des f-palindromes, Mémoire de maitrise en Mathématiques, Montréal, UQAM, 2008, 109 pages.
- [4] X. Droubay, J. Justin, G. Pirillo, Episturmian words and some constructions of de Luca and Rauzy, Theoret. Comput. Sci. 255 (2001) 539-553.

parikh_vector()

Returns the Parikh vector of self, i.e., the vector containing the number of occurrences of each letter, given in the order of the alphabet.

See also evaluation, which returns a list truncated after the last nonzero entry.

EXAMPLES:

```
sage: Word('aabaa').parikh_vector()
[4, 1]
sage: Word('aabaacaccab').parikh_vector()
[6, 2, 3]
sage: Words('abc')('aabaa').parikh_vector()
[4, 1, 0]
sage: Word().parikh_vector()
[]
```

```
sage: Word('a').parikh_vector()
[1]
sage: Words('abc')('a').parikh_vector()
[1, 0, 0]
sage: Words('ab')().parikh_vector()
[0, 0]
sage: Words('abc')().parikh_vector()
[0, 0, 0]
sage: Words('abcd')().parikh_vector()
[0, 0, 0, 0]
```

TESTS:

```
sage: P = Alphabet(name="positive integers")
sage: w = Words(P)(range(1,10))
sage: w.parikh_vector()
...
TypeError: the alphabet is infinite; use evaluation() instead
```

phi()

Applies the phi function to self and returns the result.

See for instance [1] and [2].

INPUT:

- `self` - must be a word over integers

OUTPUT:

- `word` - the result of the phi function

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([2, 2, 1, 1, 2, 1, 2, 2, 1, 2, 2, 1, 1, 2]).phi()
word: 222222
sage: W().phi()
word:
sage: W([2, 1, 2, 2, 1, 2, 2, 1, 2, 1]).phi()
word: 212113
```

REFERENCES:

- [1] S. Brlek, A. Ladouceur, A note on differentiable palindromes, Theoret. Comput. Sci. 302 (2003) 167-178.
- [2] S. Brlek, S. Dulucq, A. Ladouceur, L. Vuillon, Combinatorial properties of smooth infinite words, Theoret. Comput. Sci. 352 (2006) 306-317.

phi_inv(*W=None*)

Applied the inverse of the phi function and returns the result.

INPUT:

- `self` - must be a word over the integers
- `W` - the set of words of the result (must also be over the integers)

OUTPUT:

- `word` - the inverse of the phi function

EXAMPLES:

```
sage: W = Words([1, 2])
sage: W([2, 2, 2, 2, 1, 2]).phi_inv()
word: 22112122
sage: W([2, 2, 2]).phi_inv(Words([2, 3]))
word: 2233
```

prefix_function_table()

Returns a vector containing the length of the proper prefix-suffixes for all the non-empty prefixes of self.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('121321').prefix_function_table()
[0, 0, 1, 0, 0, 1]
sage: W('1241245').prefix_function_table()
[0, 0, 0, 1, 2, 3, 0]
sage: W().prefix_function_table()
[]
```

primitive()

Returns the primitive of self.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('12312').primitive()
word: 12312
sage: W('121212').primitive()
word: 12
```

primitive_length()

Returns the length of the primitive of self.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('1231').primitive_length()
4
sage: W('121212').primitive_length()
2
```

quasiperiods()

Returns the set of quasiperiods of self.

Let w be a finite or infinite word. A *quasiperiod* of w is a proper factor u of w such that the occurrences of u in w entirely cover w, i.e., every position of w falls within some occurrence of u in w. See for instance [1], [2], and [3].

TESTS:

```
sage: W = Words('abc')
sage: W('abaababaabaababaaba').quasiperiods() == set([W('aba'), W('abaaba'), W('abaababaaba'
True
sage: W('abaaba').quasiperiods() == set([W('aba')])
True
sage: W('abacaba').quasiperiods() == set([])
True
```

REFERENCES:

- [1] A. Apostolico, A. Ehrenfeucht, Efficient detection of quasiperiodicities in strings, Theoret. Comput. Sci. 119 (1993) 247-265.
- [2] S. Marcus, Quasiperiodic infinite words, Bull. Eur. Assoc. Theor. Comput. Sci. 82 (2004) 170-174. [3] A. Glen, F. Levé, G. Richomme, Quasiperiodic and Lyndon episturmian words, Preprint, 2008, arXiv:0805.0730.

return_words(*fact*)

Returns the set of return words of fact in self.

This is the set of all factors starting by the given factor and ending just before the next occurrence of this factor. See [1] and [2].

EXAMPLES:

```
sage: W = Words('123')
sage: W('21331233213231').return_words(W('2')) == set([W('21331'), W('233'), W('213')])
True
sage: W().return_words(W('213')) == set()
True
sage: W('121212').return_words(W('1212')) == set([W('12')])
True
```

REFERENCES:

- [1] F. Durand, A characterization of substitutive sequences using return words, Discrete Math. 179 (1998) 89-101.
- [2] C. Holton, L.Q. Zamboni, Descendants of primitive substitutions, Theory Comput. Syst. 32 (1999) 133-157.

return_words_derivate (*fact, W=None*)

Returns the word generated by mapping a letter to each occurrence of the return words for the given factor dropping any dangling prefix and suffix.

The optional set of words parameter must be over an alphabet that contains as much letters as there are different return words in self, otherwise there will be some breakage in the function. The default value for this parameter always respects this property. The letters are attributed to the words in the order they are discovered.

EXAMPLES:

```
sage: Words('123')('121312213123131122').return_words_derivate(Words('1')('1'))
word: 123242
```

REFERENCES:

- [1] F. Durand, A characterization of substitutive sequences using return words, Discrete Math. 179 (1998) 89-101.

rev_lex_less (*other*)

Returns True if the word self is reverse lexicographically less than other.

EXAMPLES:

```
sage: W = Words([1,2,3,4])
sage: W([1,2,4]).rev_lex_less(W([1,3,2]))
True
sage: W([3,2,1]).rev_lex_less(W([1,2,3]))
False
```

reversal ()

Returns the reversal of self.

EXAMPLES:

```
sage: W = Words('0123456789')
sage: W('124563').reversal()
word: 365421
```

shifted_shuffle (*other*)

Returns the combinatorial class representing the shifted shuffle product between words self and other. This is the same as the shuffle product of self with the word obtained from other by incrementing the values by len(self).

EXAMPLES:

```
sage: W = Words("abcd")
sage: ab = W("ab")
sage: cd = W("cd")
sage: sp = ab.shifted_shuffle(ab); sp
```

```
Shuffle product of word: ab and word: cd
sage: sp.cardinality()
6
sage: sp.list()
[word: abcd, word: acbd, word: acdb, word: cabd, word: cadb, word: cdab]
```

shuffle(*other, overlap=0*)

Returns the combinatorial class representing the shuffle product between words self and other. This consists of all words of length len(self)+len(other) that have both self and other as subwords.

If overlap is non-zero, then the combinatorial class representing the shuffle product with overlaps is returned. The calculation of the shift in each overlap is done relative to the order of the alphabet. For example, "a" shifted by "a" is "b" in the alphabet [a, b, c] and 0 shifted by 1 in [0, 1, 2, 3] is 2.

EXAMPLES:

```
sage: W = Words("abcd")
sage: ab = W("ab")
sage: cd = W("cd")
sage: sp = ab.shuffle(cd); sp
Shuffle product of word: ab and word: cd
sage: sp.cardinality()
6
sage: sp.list()
[word: abcd, word: acbd, word: acdb, word: cabd, word: cadb, word: cdab]
sage: W = Words(range(10))
sage: w = W([0,1])
sage: u = W([2,3])
sage: w.shuffle(w)
Shuffle product of word: 01 and word: 01
sage: u.shuffle(u)
Shuffle product of word: 23 and word: 23
sage: w.shuffle(u)
Shuffle product of word: 01 and word: 23
sage: w.shuffle(u,2)
Overlapping shuffle product of word: 01 and word: 23 with 2 overlaps
```

standard_factorization()

Returns the standard factorization of self.

The *standard factorization* of a word w is the unique factorization: $w = uv$ where v is the longest proper suffix of w that qualifies as a Lyndon word.

Note that if w is a Lyndon word with standard factorization $w = uv$, then u and v are also Lyndon words and $u < v$.

See for instance [1] and [2].

OUTPUT:

- `list` - the list of factors

EXAMPLES:

```
sage: Words('01')('0010110011').standard_factorization()
(001011.0011)
sage: Words('123')('1223312').standard_factorization()
(12233.12)
```

REFERENCES:

- [1] K.-T. Chen, R.H. Fox, R.C. Lyndon, Free differential calculus, IV. The quotient groups of the lower central series, Ann. of Math. 68 (1958) 81-95.
- [2] J.-P. Duval, Factorizing words over an ordered alphabet, J. Algorithms 4 (1983) 363-381.

standard_factorization_of_lyndon_factorization()

Returns the standard factorization of the Lyndon factorization of self.

OUTPUT:

- `list of lists` - the factorization

EXAMPLES:

```
sage: Words('123')('1221131122').standard_factorization_of_lyndon_factorization()
[(12.2), (1.13), (1.122)]
```

standard_permutation()

Returns the standard permutation of the word self on the ordered alphabet. It is defined as the permutation with exactly the same number of inversions as w. Equivalently, it is the permutation of minimal length whose inverse sorts self.

EXAMPLES:

```
sage: w = Word([1,2,3,2,2,1]); w
word: 123221
sage: p = w.standard_permutation(); p
[1, 3, 6, 4, 5, 2]
sage: v = Word(p.inverse().action(w)); v
word: 112223
sage: Permutations(len(w)).filter( \
...     lambda q: q.length() <= p.length() and \
...                 q.inverse().action(w) == list(v) ).list()
[[1, 3, 6, 4, 5, 2]]

sage: w = Words([1,2,3])([1,2,3,2,2,1,2,1]); w
word: 12322121
sage: p = w.standard_permutation(); p
[1, 4, 8, 5, 6, 2, 7, 3]
sage: Word(p.inverse().action(w))
word: 11122223

sage: w = Words([3,2,1])([1,2,3,2,2,1,2,1]); w
word: 12322121
sage: p = w.standard_permutation(); p
[6, 2, 1, 3, 4, 7, 5, 8]
sage: Word(p.inverse().action(w))
word: 32222111

sage: w = Words('ab')('abbaba'); w
word: abbaba
sage: p = w.standard_permutation(); p
[1, 4, 5, 2, 6, 3]
sage: Word(p.inverse().action(w))
word: aaabbb

sage: w = Words('ba')('abbaba'); w
word: abbaba
sage: p = w.standard_permutation(); p
[4, 1, 2, 5, 3, 6]
sage: Word(p.inverse().action(w))
word: bbbaaa
```

string_rep()

Returns the raw sequence of letters as a string.

EXAMPLES:

```
sage: Words('ab')('abbabaab').string_rep()
'abbabaab'
sage: Words(range(2))([0, 1, 0, 0, 1]).string_rep()
'01001'
sage: Words(range(1000))([0,1,10,101]).string_rep()
'0,1,10,101'
sage: WordOptions(letter_separator='-')
sage: Words(range(1000))([0,1,10,101]).string_rep()
'0-1-10-101'
```

suffix_tree()

Alias for implicit_suffix_tree().

EXAMPLES:

```
sage: Words('ab')('abbabaab').suffix_tree()
Implicit Suffix Tree of the word: abbabaab
```

suffix_trie()

Returns the suffix trie of self.

The *suffix trie* of a finite word w is a data structure representing the factors of w. It is a tree whose edges are labelled with letters of w, and whose leafs correspond to suffixes of w.

See sage.combinat.words.suffix_trees.SuffixTrie? for more information.

EXAMPLES:

```
sage: w = Words("cao")("cacao")
sage: w.suffix_trie()
Suffix Trie of the word: cacao

sage: w = Words([0,1])([0,1,0,1,1])
sage: w.suffix_trie()
Suffix Trie of the word: 01011
```

swap(*i*, *j=None*)

Returns the word w with entries at positions i and j swapped. By default, j = i+1.

EXAMPLES:

```
sage: Word([1,2,3]).swap(0,2)
word: 321
sage: Word([1,2,3]).swap(1)
word: 132
sage: Words("ba")("abba").swap(1,-1)
word: aabb
```

swap_decrease(*i*)

Returns the word with positions i and i+1 exchanged if self[i] < self[i+1]. Otherwise, it returns self.

EXAMPLES:

```
sage: w = Word([1,3,2])
sage: w.swap_decrease(0)
word: 312
sage: w.swap_decrease(1)
word: 132
sage: w.swap_decrease(1) is w
True
sage: Words("ab")("abba").swap_decrease(0)
word: baba
sage: Words("ba")("abba").swap_decrease(0)
word: abba
```

swap_increase(*i*)
Returns the word with positions i and i+1 exchanged if self[i] > self[i+1]. Otherwise, it returns self.

EXAMPLES:

```
sage: w = Word([1,3,2])
sage: w.swap_increase(1)
word: 123
sage: w.swap_increase(0)
word: 132
sage: w.swap_increase(0) is w
True
sage: Words("ab")("abba").swap_increase(0)
word: abba
sage: Words("ba")("abba").swap_increase(0)
word: baba
```

class InfiniteWord_over_Alphabet(*parent, *args, **kwds*)

class InfiniteWord_over_OrderedAlphabet(*parent, *args, **kwds*)

Word(*data=None, alphabet=None*)
Returns a word over the given alphabet.

INPUT:

- `data` - An iterable (list, string, iterator) or a function defined on nonnegative integers that yields the letters of the word.
- `alphabet` - An iterable yielding letters in their comparative order. If alphabet is None and data is a string or list, then the letters appearing in data, ordered by Python's builtin sorted function, is used as alphabet.

OUTPUT: A FiniteWord_over_OrderedAlphabet or InfiniteWord_over_OrderedAlphabet object.

NOTE: Essentially, this function just returns Words(alphabet)(data).

EXAMPLES: 0. The empty word.

```
sage: Word()
word:
```

1.Finite words from strings.

```
sage: Word("abbabaab")
word: abbabaab
```

```
sage: Word("abbabaab", alphabet="abc")
word: abbabaab
```

1.Finite words from lists.

```
sage: Word(["a", "b", "b", "a", "b", "a", "a", "b"])
word: abbabaab
```

```
sage: Word([0,1,1,0,1,0])
word: 011010
```

1.Finite words from functions.

```
sage: f = lambda n : n % 2
sage: Word(f, [0, 1])[:17]
word: 01010101010101010
```

1.Finite words from iterators.

```
sage: def tmword():
...     thuemorse = WordMorphism('a->ab,b->ba')
...     w = thuemorse('a')[:]
...     i = 0
...     while w:
...         for x in thuemorse(w[i]):
...             yield x
...         else:
...             w *= thuemorse(w[i+1])
...             i += 1
sage: Word(tmword(), alphabet="ab")[:32]
word: abbabaabbaababbabaababbaabbabaab
```

1.Infinite words from functions.

```
sage: f = lambda n : n % 2
sage: Word(f, alphabet=[0, 1])
Infinite word over [0, 1]
```

1.Infinite words from iterators.

```
sage: def tmword():
...     thuemorse = WordMorphism('a->ab,b->ba')
...     w = thuemorse('a')[:]
...     i = 0
...     while w:
...         for x in thuemorse(w[i]):
...             yield x
...         else:
...             w *= thuemorse(w[i+1])
...             i += 1
sage: Word(tmword(), alphabet="ab")
Infinite word over ['a', 'b']
```

TESTS:

```
sage: f = lambda n : n % 2
sage: Word(f)
Traceback (most recent call last):
  ...
TypeError: alphabet is required for words not defined by lists or strings
sage: Word(f, alphabet=[1])
Infinite word over [1]
```

WordOptions(***kwargs*)

Sets the global options for elements of the word class. The defaults are for words to be displayed in list notation.

INPUT:

- `display` - 'string' (default), or 'list', words are displayed in string or list notation.
- `truncate` - boolean (default: True), whether to truncate the string output of long words (see truncate_length below).
- `truncate_length` - integer (default: 40), if the length of the word is greater than this integer, then the word is truncated.
- `letter_separator` - (string, default: ",") if the string representation of letters have length greater than 1, then the letters are separated by this string in the string representation of the word.

If no parameters are set, then the function returns a copy of the options dictionary.

EXAMPLES:

```
sage: w = Word([2,1,3,12])
sage: u = Word("abba")
sage: WordOptions(display='list')
sage: w
word: [2, 1, 3, 12]
sage: u
word: ['a', 'b', 'b', 'a']
sage: WordOptions(display='string')
sage: w
word: 2,1,3,12
sage: u
word: abba
```

class **Word_over_Alphabet** (*parent, *args, **kwds*)

alphabet()

Returns the alphabet of the parent.

EXAMPLES:

```
sage: Words('abc')('abbabaab').alphabet()
Ordered Alphabet ['a', 'b', 'c']
```

class **Word_over_OrderedAlphabet** (*parent, *args, **kwds*)

is_FiniteWord(*obj*)

Returns True if obj is a finite word, and False otherwise.

EXAMPLES:

```
sage: from sage.combinat.words.word import is_FiniteWord
sage: is_FiniteWord(33)
False
sage: is_FiniteWord(Word('baab'))
True
```

is_Word(*obj*)

Returns True if obj is a word, and False otherwise.

EXAMPLES:

```
sage: from sage.combinat.words.word import is_Word
sage: is_Word(33)
False
sage: is_Word(Word('abba'))
True
```

17.40.6 Word contents

BuildWordContent(*obj, mapping=<function id_f at 0x2c83938>, format=None, part=slice(None, None, None)*)

Builds the content for a word.

INPUT:

- `obj` - a function, an iterator or a list, potentially with a length defined
- `mapping` - function, a map sending elements of the iterable to nonnegative integers.
- `format` - string (default None), the explicit type of obj. Can be either 'empty', 'list', 'function' or 'iterator'. If set to None (the default), the type will be guessed from the properties of obj.
- `part` - slice (default slice(None)), the portion of the object to use

OUTPUT:

- `word content` - an object respecting the word content protocol wrapping obj

TESTS:

```
sage: from sage.combinat.words.word_content import BuildWordContent
sage: from itertools import count, imap, repeat
sage: len(BuildWordContent(None))
0
sage: len(BuildWordContent(None, format='empty'))
0
sage: c = BuildWordContent(['a', 'b'], format='empty')
...
TypeError: trying to build an empty word with something other than None
sage: len(BuildWordContent(['0', '1', '1'], int))
3
sage: len(BuildWordContent(['0', '1', '1'], int, format='list'))
3
sage: BuildWordContent(10, format='list')
...
TypeError: trying to build a word backed by a list with a sequence not providing the required op
sage: c = BuildWordContent(lambda x: x%2)
sage: len(BuildWordContent(lambda x: x%3, part=slice(0,10)))
10
sage: c = BuildWordContent(repeat(1))
sage: len(BuildWordContent(count(), part=slice(10)))
10
sage: len(BuildWordContent(count(), part=slice(10, 0, -2)))
5
```

class ConcatenateContent(*l*)

Class that acts as a function to concatenate contents.

class WordContent()

concatenate(*other*)

Method to concatenate two contents together.

TESTS:

```
sage: from sage.combinat.words.word_content import BuildWordContent
sage: list(BuildWordContent([1]).concatenate(BuildWordContent([2, 3, 4])))
[1, 2, 3, 4]
```

class WordContentFromFunction(*func, trans=<function id_f at 0x2c83938>*)

class WordContentFromIterator(*it, trans=<function id_f at 0x2c83938>*)

class WordContentFromList(*l, trans=<function id_f at 0x2c83938>*)

is_WordContent(*obj*)

Returns True if obj is the content of a word.

EXAMPLES:

```
sage: from sage.combinat.words.word_content import BuildWordContent, is_WordContent
sage: is_WordContent(33)
False
sage: is_WordContent(BuildWordContent([0, 1, 0], sage.combinat.words.utils.id_f))
True
```

17.40.7 Word generators

class ChristoffelWord_Lower(*p, q, alphabet=(0, 1)*)

markoff_number()

Returns the Markoff number associated to the Christoffel word self.

The *Markoff number* of a Christoffel word w is $1/3traceM(w)$, where $M(w)$ is the 2×2 matrix obtained by applying the morphism: 0 - matrix(2,[2,1,1,1]) 1 - matrix(2,[5,2,2,1])

EXAMPLES:

```
sage: from sage.combinat.words.word_generators import ChristoffelWord_Lower
sage: w0 = ChristoffelWord_Lower(4,7)
sage: w1, w2 = w0.standard_factorization()
sage: (m0,m1,m2) = (w.markoff_number() for w in (w0,w1,w2))
sage: (m0,m1,m2)
(294685, 13, 7561)
sage: m0**2 + m1**2 + m2**2 == 3*m0*m1*m2
True
```

standard_factorization()

Returns the standard factorization of the Christoffel word self.

The *standard factorization* of a Christoffel word w is the unique factorization of w into two Christoffel words.

EXAMPLES:

```
sage: from sage.combinat.words.word_generators import ChristoffelWord_Lower
sage: w = ChristoffelWord_Lower(5,9)
sage: print w
word: 00100100100101
sage: w1, w2 = w.standard_factorization()
sage: print w1
word: 001
sage: print w2
word: 00100100101

sage: w = ChristoffelWord_Lower(51,37)
sage: w1, w2 = w.standard_factorization()
sage: w1
Lower Christoffel word of slope 11/8 over the alphabet [0, 1]
sage: w2
Lower Christoffel word of slope 40/29 over the alphabet [0, 1]
```

```
sage: w1 * w2 == w
True
```

class WordGenerator()

A class consisting of constructors for several famous words.

TESTS:

```
sage: type(loads(dumps(words)))
<class 'sage.combinat.words.word_generators.WordGenerator'>
```

CharacteristicSturmianWord(*cf, alphabet=(0, 1), repeat=True*)

Returns the characteristic Sturmian word over the given two-letter alphabet with slope given by the continued fraction [0, cf[1], cf[2], ...]. Here cf should be an iterator.

The *characteristic Sturmian word of slope* $\alpha = [0, d[1] + 1, d[2], d[3], \ldots]$ is the limit of the sequence: $s_0 = 1$, $s_1 = 0$, ..., $s_{n+1} = s_n^{d_n} s_{n-1}$ for $n > 0$.

Equivalently, the n-th term of the characteristic Sturmian word of slope α is $\lfloor \alpha(n+1) \rfloor - \lfloor \alpha n \rfloor + \lfloor \alpha \rfloor$.

INPUT:

- `alphabet` - any container of length two that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)
- `cf` - an iterator outputting integers (thought of as a continued fraction)

EXAMPLES:

```
sage: def cf():
...     yield 0
...     while True: yield 1
...
sage: F = words.CharacteristicSturmianWord(cf()); F
Characteristic Sturmian word over [0, 1], defined recursively
sage: Fib = words.FibonacciWord(); Fib
Fibonacci word over [0, 1], defined recursively
sage: F[:10000] == Fib[:10000]
True

sage: def cf():
...     yield 0
...     while True: yield 1
...
sage: G = words.CharacteristicSturmianWord(cf(),'rs'); G
Characteristic Sturmian word over ['r', 's'], defined recursively
sage: print G[:50]
word: rsrrsrsrrsrrsrsrrsrsrrsrrsrsrrsrrsrsrrsrsrrsrrsrsr
```

TESTS:

```
sage: words.CharacteristicSturmianWord([1,1,1],'xyz')
...
TypeError: alphabet does not contain two distinct elements
```

ChristoffelWord(*p, q, alphabet=(0, 1)*)

Returns the lower Christoffel word of slope p/q, where p and q are relatively prime non-negative integers, over the given two-letter alphabet.

The *Christoffel word of slope 'p/q'* is obtained from the Cayley graph of $\mathbf{Z}/(p+q)\mathbf{Z}$ with generator q as follows. If $u \to v$ is an edge in the Cayley graph, then $v = u + p \mod p + q$. Label the edge $u \to v$ by alphabet[1] if $u < v$ and alphabet[0] otherwise. The Christoffel word is the word obtained by reading the edge labels along the cycle beginning from 0.

INPUT:

- `alphabet` - any container of length two that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)

EXAMPLES:

```
sage: w = words.ChristoffelWord(1,0); w
Lower Christoffel word of slope 1/0 over the alphabet [0, 1]
sage: print w
word: 1

sage: w = words.ChristoffelWord(0,1); w
Lower Christoffel word of slope 0/1 over the alphabet [0, 1]
sage: print w
word: 0

sage: w = words.ChristoffelWord(1,1); w
Lower Christoffel word of slope 1/1 over the alphabet [0, 1]
sage: print w
word: 01

sage: w = words.ChristoffelWord(4,7); w
Lower Christoffel word of slope 4/7 over the alphabet [0, 1]
sage: print w
word: 00100100101
```

TESTS:

```
sage: words.ChristoffelWord(51,43,"abc")
...
TypeError: alphabet does not contain two distinct elements
```

CodingOfRotationWord(*alpha, beta, x=0, alphabet=(0, 1)*)

Returns the infinite word obtained from the coding of rotation of parameters (α, β, x) over the given two-letter alphabet.

The *coding of rotation* corresponding to the parameters (α, β, x) is the symbolic sequence $u = (u_n)_{n\geq 0}$ defined over the binary alphabet $\{0, 1\}$ by $u_n = 1$ if $x + n\alpha \in [0, \beta[$ and $u_n = 0$ otherwise. See [1].

EXAMPLES:

```
sage: alpha = 0.45
sage: beta = 0.48
sage: w = words.CodingOfRotationWord(0.45, 0.48); w
Coding of rotation with parameters (0.450000000000000, 0.480000000000000, 0)
sage: w[:100]
Finite word of length 100 over [0, 1]
sage: w[:30]
word: 110101010100101010101101010101

sage: u = words.CodingOfRotationWord(0.45, 0.48, alphabet='xy')
sage: u[:100]
Finite word of length 100 over ['x', 'y']
sage: u[:30]
word: yyxyxyxyxyxxyxyxyxyxyyxyxyxyxy
```

TESTS:

```
sage: words.CodingOfRotationWord(0.51,0.43,alphabet=[1,0,2])
...
TypeError: alphabet does not contain two distinct elements
```

REFERENCES:

- [1] B. Adamczewski, J. Cassaigne, On the transcendence of real numbers with a regular expansion, J.

Number Theory 103 (2003) 27-37.

FibonacciWord(*alphabet=(0, 1), construction_method='recursive'*)

Returns the Fibonacci word on the given two-letter alphabet.

INPUT:

- `alphabet` - any container of length two that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)
- `construction_method` - can be any of the following: "recursive", "fixed point" (see below for definitions).

Recursive construction: the Fibonacci word is the limit of the following sequence of words: $S_0 = 0$, $S_1 = 01$, $S_n = S_{n-1}S_{n-2}$ for $n \geq 2$.

Fixed point construction: the Fibonacci word is the fixed point of the morphism: $0 \mapsto 01$ and $1 \mapsto 0$. Hence, it can be constructed by the following read-write process:

beginning at the first letter of 01, if the next letter is 0, append 01 to the word; if the next letter is 1, append 1 to the word; move to the next letter of the word.

EXAMPLES:

```
sage: w = words.FibonacciWord(construction_method="recursive"); w
Fibonacci word over [0, 1], defined recursively
sage: print w[:99]
word: 010010100100101001010010010100100101001010010010100101001010010010100101001010010010100101001
```

```
sage: v = words.FibonacciWord(construction_method="recursive", alphabet='ab'); v
Fibonacci word over ['a', 'b'], defined recursively
sage: print v[:99]
word: abaababaabaababaababaabaababaabaababaababaabaababaabaababaabaababaababaabaababaabaababaababaabaab
```

```
sage: u = words.FibonacciWord(construction_method="fixed_point"); u
Fibonacci word over [0, 1], defined as the fixed point of a morphism
sage: print u[:99]
word: 010010100100101001010010010100100101001010010010100101001010010010100101001010010010100101001
```

```
sage: x = words.FibonacciWord(construction_method="fixed_point", alphabet=[4, 1]); x
Fibonacci word over [1, 4], defined as the fixed point of a morphism
sage: print x[:99]
word: 414414144144144144144144144144144144144144144144144144144144144144144144144144144144144144144
```

TESTS:

```
sage: from math import floor, sqrt
sage: golden_ratio = (1 + sqrt(5))/2.0
sage: a = golden_ratio / (1  + 2*golden_ratio)
sage: wn = lambda n : int(floor(a*(n+2)) - floor(a*(n+1)))
sage: f = Words([0,1])(wn); f
Infinite word over [0, 1]
sage: f[:10000] == w[:10000]
True
sage: f[:10000] == u[:10000] #long time
True
sage: words.FibonacciWord("abc")
...
TypeError: alphabet does not contain two distinct elements
```

FixedPointOfMorphism(*morphism, first_letter*)

Returns the fixed point of the morphism beginning with first_letter.

A *fixed point* of a morphism φ is a word w such that $\varphi(w) = w$.

INPUT:

- `morphism` - endomorphism prolongable on first_letter. It must be something that WordMorphism's constructor understands (dict, str, ...).
- `first_letter` - the first letter of the fixed point

OUTPUT:

- `word` - the fixed point of the morphism beginning with first_letter

EXAMPLES:

```
sage: mu = {0:[0,1], 1:[1,0]}
sage: tm = words.FixedPointOfMorphism(mu,0); tm
Fixed point beginning with 0 of the morphism WordMorphism: 0->01, 1->10
sage: TM = words.ThueMorseWord()
sage: tm[:1000] == TM[:1000]
True

sage: mu = {0:[0,1], 1:[0]}
sage: f = words.FixedPointOfMorphism(mu,0); f
Fixed point beginning with 0 of the morphism WordMorphism: 0->01, 1->0
sage: F = words.FibonacciWord(); F
Fibonacci word over [0, 1], defined recursively
sage: f[:1000] == F[:1000]
True

sage: fp = words.FixedPointOfMorphism('a->abc,b->,c->','a'); fp
Fixed point beginning with 'a' of the morphism WordMorphism: a->abc, b->, c->
sage: fp[:10]
word: abc
```

LowerChristoffelWord(*p, q, alphabet=(0, 1)*)

Returns the lower Christoffel word of slope p/q, where p and q are relatively prime non-negative integers, over the given two-letter alphabet.

The *Christoffel word of slope 'p/q'* is obtained from the Cayley graph of $\mathbf{Z}/(p+q)\mathbf{Z}$ with generator q as follows. If $u \to v$ is an edge in the Cayley graph, then $v = u + p \mod p+q$. Label the edge $u \to v$ by alphabet[1] if $u < v$ and alphabet[0] otherwise. The Christoffel word is the word obtained by reading the edge labels along the cycle beginning from 0.

INPUT:

- `alphabet` - any container of length two that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)

EXAMPLES:

```
sage: w = words.ChristoffelWord(1,0); w
Lower Christoffel word of slope 1/0 over the alphabet [0, 1]
sage: print w
word: 1

sage: w = words.ChristoffelWord(0,1); w
Lower Christoffel word of slope 0/1 over the alphabet [0, 1]
sage: print w
word: 0

sage: w = words.ChristoffelWord(1,1); w
Lower Christoffel word of slope 1/1 over the alphabet [0, 1]
sage: print w
word: 01

sage: w = words.ChristoffelWord(4,7); w
Lower Christoffel word of slope 4/7 over the alphabet [0, 1]
```

```
sage: print w
word: 00100100101
```

TESTS:

```
sage: words.ChristoffelWord(51,43,"abc")
...
TypeError: alphabet does not contain two distinct elements
```

MinimalSmoothPrefix(*n*)

This function finds and returns the minimal smooth prefix of length n.

See [1] for a definition.

INPUT:

- `n` - the desired length of the prefix

OUTPUT:

- `word` - the prefix

Be patient, this function can take a really long time if asked for a large prefix.

EXAMPLES:

```
sage: words.MinimalSmoothPrefix(10)
word: 1212212112
```

REFERENCES: [1] S. Brlek, G. Melançon, G. Paquin, Properties of the extremal infinite smooth words, Discrete Math. Theor. Comput. Sci. 9 (2007) 33-49.

RandomWord(*n, m=2, alphabet=None*)

Returns a random word of length n over the given m-letter alphabet (default alphabet is 0,1,...,m-1).

INPUT:

- `n` - integer, the length of the word
- `m` - integer (default 2), the size of the output alphabet
- `alphabet` - any container of length m that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)

EXAMPLES:

```
sage: words.RandomWord(10)          # random results
word: 0110100101
sage: words.RandomWord(10, 4)       # random results
word: 0322313320
sage: words.RandomWord(100, 7)
Finite word of length 100 over [0, 1, 2, 3, 4, 5, 6]
sage: words.RandomWord(100, 7, range(-3,4))
Finite word of length 100 over [-3, -2, -1, 0, 1, 2, 3]
sage: words.RandomWord(100, 5, "abcde")
Finite word of length 100 over ['a', 'b', 'c', 'd', 'e']
sage: words.RandomWord(17, 5, "abcde")      # random results
word: dcacbbecbddebaadd
```

TESTS:

```
sage: words.RandomWord(2,3,"abcd")
...
TypeError: alphabet does not contain 3 distinct elements
```

StandardEpisturmianWord(*directive_word*)

Returns the standard episturmian word (or epistandard word) directed by directive_word. Over a 2-letter alphabet, this function gives characteristic Sturmian words.

An infinite word w over a finite alphabet A is said to be *standard episturmian* (or *epistandard*) iff there exists an infinite word $x_1x_2x_3\cdots$ over A (called the *directive word* of w) such that w is the limit as n goes to infinity of $Pal(x_1\cdots x_n)$, where Pal is the iterated palindromic closure function.

Note that an infinite word is *episturmian* if it has the same set of factors as some epistandard word.

See for instance [1], [2], and [3].

INPUT:

- `directive word` - an infinite word or a period of a periodic infinite word

EXAMPLES:

```
sage: Fibonacci = words.StandardEpisturmianWord(Word('ab')); Fibonacci
Standard episturmian word over ['a', 'b']
sage: Fibonacci[:25]
word: abaababaabaababaababaaba
sage: Tribonacci = words.StandardEpisturmianWord(Word('abc')); Tribonacci
Standard episturmian word over ['a', 'b', 'c']
sage: Tribonacci[:25]
word: abacabaabacababacabaabaca
sage: S = words.StandardEpisturmianWord(Word('aabcabada')); S
Standard episturmian word over ['a', 'b', 'c', 'd']
sage: print S[:75]
word: aabaacaabaaabaacaabaabaacaabaaabaacaabaaabaacaabaabaacaabaaabaacaabaadaabaa
sage: S = words.StandardEpisturmianWord(Fibonacci); S
Standard episturmian word over ['a', 'b']
sage: S[:25]
word: abaabaababaabaabaababaaba
sage: S = words.StandardEpisturmianWord(Tribonacci); S
Standard episturmian word over ['a', 'b', 'c']
sage: S[:25]
word: abaabacabaabaabacabaababa
sage: words.StandardEpisturmianWord(123)
...
TypeError: directive_word is not a word, so it cannot be used to build an episturmian word
sage: words.StandardEpisturmianWord(Words('ab'))
...
TypeError: directive_word is not a word, so it cannot be used to build an episturmian word
```

REFERENCES:

- [1] X. Droubay, J. Justin, G. Pirillo, Episturmian words and some constructions of de Luca and Rauzy, Theoret. Comput. Sci. 255 (2001) 539-553.
- [2] J. Justin, G. Pirillo, Episturmian words and episturmian morphisms, Theoret. Comput. Sci. 276 (2002) 281-313.
- [3] A. Glen, J. Justin, Episturmian words: a survey, Preprint, 2007, arXiv:0801.1655.

ThueMorseWord(*alphabet=(0, 1)*)

Returns the Thue-Morse word over the given two-letter alphabet.

There are several ways to define the Thue-Morse word t. We use the following definition: $t[n]$ is the sum modulo 2 of the digits in the binary expansion of n.

INPUT:

- `alphabet` - any container of length two that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)

EXAMPLES:

```
sage: t = words.ThueMorseWord(); t
Thue-Morse word on the alphabet [0, 1]
sage: print t[:50]
word: 01101001100101101001011001101001100101100110100101
```

```
sage: t = words.ThueMorseWord('ab'); t
Thue-Morse word on the alphabet ['a', 'b']
sage: print t[:50]
word: abbabaabbaababbabaababbaabbabaabbaababbaabbabaabab

sage: t = words.ThueMorseWord(['L1', 'L2']); t
Thue-Morse word on the alphabet ['L1', 'L2']
sage: words.ThueMorseWord(['L1','L2'])[:8]
word: L1,L2,L2,L1,L2,L1,L1,L2
```

TESTS:

```
sage: words.ThueMorseWord("abc")
...
TypeError: alphabet does not contain two distinct elements
```

UpperChristoffelWord(*p, q, alphabet=(0, 1)*)

Returns the upper Christoffel word of slope p/q, where p and q are relatively prime non-negative integers, over the given alphabet.

The *upper Christoffel word of slope 'p/q'* is equal to the reversal of the lower Christoffel word of slope p/q. Equivalently, if xuy is the lower Christoffel word of slope p/q, where x and y are letters, then yux is the upper Christoffel word of slope p/q (because u is a palindrome).

INPUT:

- `alphabet` - any container of length two that is suitable to build an instance of OrderedAlphabet (list, tuple, str, ...)

EXAMPLES:

```
sage: w = words.UpperChristoffelWord(1,0); w
Upper Christoffel word of slope 1/0 over the alphabet [0, 1]
sage: print w
word: 1

sage: w = words.UpperChristoffelWord(0,1); w
Upper Christoffel word of slope 0/1 over the alphabet [0, 1]
sage: print w
word: 0

sage: w = words.UpperChristoffelWord(1,1); w
Upper Christoffel word of slope 1/1 over the alphabet [0, 1]
sage: print w
word: 10

sage: w = words.UpperChristoffelWord(4,7); w
Upper Christoffel word of slope 4/7 over the alphabet [0, 1]
sage: print w
word: 10100100100
```

TESTS:

```
sage: words.UpperChristoffelWord(51,43,"abc")
...
TypeError: alphabet does not contain two distinct elements
```

17.40.8 Words

class FiniteWords_length_k_over_OrderedAlphabet(*alphabet, length*)

cardinality()

Returns the number of words of length n from alphabet.

EXAMPLES:

```
sage: Words(['a','b','c'], 4).cardinality()
81
sage: Words(3, 4).cardinality()
81
sage: Words(0,0).cardinality()
1
sage: Words(5,0).cardinality()
1
sage: Words(['a','b','c'],0).cardinality()
1
sage: Words(0,1).cardinality()
0
sage: Words(5,1).cardinality()
5
sage: Words(['a','b','c'],1).cardinality()
3
sage: Words(7,13).cardinality()
96889010407
sage: Words(['a','b','c','d','e','f','g'],13).cardinality()
96889010407
```

iterate_by_length(*length*)

All words in this class are of the same length, so use iterator instead.

TESTS:

```
sage: W = Words(['a', 'b'], 2)
sage: list(W.iterate_by_length(2))
[word: aa, word: ab, word: ba, word: bb]
sage: list(W.iterate_by_length(1))
[]
```

class **FiniteWords_over_OrderedAlphabet**(*alphabet*)

class **InfiniteWords_over_OrderedAlphabet**(*alphabet*)

Words(*alphabet=None, length=None, finite=True, infinite=True*)

Returns the combinatorial class of words of length k over an ordered alphabet.

EXAMPLES:

```
sage: Words()
Words
sage: Words(length=7)
Finite Words of length 7
sage: Words(5)
Words over Ordered Alphabet [1, 2, 3, 4, 5]
sage: Words(5, 3)
Finite Words over Ordered Alphabet [1, 2, 3, 4, 5] of length 3
sage: Words(5, infinite=False)
Finite Words over Ordered Alphabet [1, 2, 3, 4, 5]
sage: Words(5, finite=False)
Infinite Words over Ordered Alphabet [1, 2, 3, 4, 5]
sage: Words('ab')
Words over Ordered Alphabet ['a', 'b']
sage: Words('ab', 2)
Finite Words over Ordered Alphabet ['a', 'b'] of length 2
```

```
sage: Words('ab', infinite=False)
Finite Words over Ordered Alphabet ['a', 'b']
sage: Words('ab', finite=False)
Infinite Words over Ordered Alphabet ['a', 'b']
sage: Words('positive integers', finite=False)
Infinite Words over Ordered Alphabet of Positive Integers
sage: Words('natural numbers')
Words over Ordered Alphabet of Natural Numbers
```

class Words_all()

TESTS:

```
sage: from sage.combinat.words.words import Words_all
sage: list(Words_all())
...
NotImplementedError
sage: Words_all().list()
...
NotImplementedError: infinite list
sage: Words_all().cardinality()
+Infinity
```

class Words_n(*n*)

class Words_over_Alphabet(*alphabet*)

alphabet()

EXAMPLES:

```
sage: from sage.combinat.words.words import Words_over_Alphabet
sage: W = Words_over_Alphabet([1,2,3])
sage: W.alphabet()
[1, 2, 3]
sage: from sage.combinat.words.words import OrderedAlphabet
sage: W = Words_over_Alphabet(OrderedAlphabet('ab'))
sage: W.alphabet()
Ordered Alphabet ['a', 'b']
```

size_of_alphabet()

Returns the size of the alphabet.

EXAMPLES:

```
sage: Words('abcdef').size_of_alphabet()
6
sage: Words('').size_of_alphabet()
0
```

class Words_over_OrderedAlphabet(*alphabet*)

iter_morphisms(*l*)

Iterate over all endomorphisms φ of self satisfying $|\varphi(a[i])| = l[i]$, where a[i] is the i-th letter of self.

INPUT:

- `l` - list of integers such that len(l) == self.size_of_alphabet()

OUTPUT:

- `iterator outputting WordMorphism` - outputs the endomorphisms

EXAMPLES:

```
sage: W = Words('ab')
sage: map(str, W.iter_morphisms([2, 1]))
['WordMorphism: a->aa, b->a',
 'WordMorphism: a->aa, b->b',
 'WordMorphism: a->ab, b->a',
 'WordMorphism: a->ab, b->b',
 'WordMorphism: a->ba, b->a',
 'WordMorphism: a->ba, b->b',
 'WordMorphism: a->bb, b->a',
 'WordMorphism: a->bb, b->b']
sage: map(str, W.iter_morphisms([2, 2]))
['WordMorphism: a->aa, b->aa',
 'WordMorphism: a->aa, b->ab',
 'WordMorphism: a->aa, b->ba',
 'WordMorphism: a->aa, b->bb',
 'WordMorphism: a->ab, b->aa',
 'WordMorphism: a->ab, b->ab',
 'WordMorphism: a->ab, b->ba',
 'WordMorphism: a->ab, b->bb',
 'WordMorphism: a->ba, b->aa',
 'WordMorphism: a->ba, b->ab',
 'WordMorphism: a->ba, b->ba',
 'WordMorphism: a->ba, b->bb',
 'WordMorphism: a->bb, b->aa',
 'WordMorphism: a->bb, b->ab',
 'WordMorphism: a->bb, b->ba',
 'WordMorphism: a->bb, b->bb']
sage: map(str, W.iter_morphisms([0, 0]))
['WordMorphism: a->, b->']
sage: map(str, W.iter_morphisms([0, 1]))
['WordMorphism: a->, b->a', 'WordMorphism: a->, b->b']
sage: list(W.iter_morphisms([1,0]))
[Morphism from Words over Ordered Alphabet ['a', 'b'] to Words over Ordered Alphabet ['a', '
```

TESTS:

```
sage: list(W.iter_morphisms([0, 1, 2]))
...
TypeError: l (=[0, 1, 2]) must be a list of 2 integers
sage: list(W.iter_morphisms([0, 'a']))
...
TypeError: l (=[0, 'a']) must be a list of 2 integers
```

iterate_by_length(*l=1*)

Returns an iterator over all the words of self of length l.

INPUT:

- `l` - integer (default: 1) the length of the desired words

EXAMPLES:

```
sage: W = Words('ab')
sage: list(W.iterate_by_length(1))
[word: a, word: b]
sage: list(W.iterate_by_length(2))
[word: aa, word: ab, word: ba, word: bb]
sage: list(W.iterate_by_length(3))
[word: aaa,
 word: aab,
 word: aba,
```

```
 word: abb,
 word: baa,
 word: bab,
 word: bba,
 word: bbb]
sage: list(W.iterate_by_length('a'))
...
TypeError: the parameter l ('a') must be an integer
```

is_Words(*obj*)

Returns True if obj is a word set and False otherwise.

EXAMPLES:

```
sage: from sage.combinat.words.words import is_Words
sage: is_Words(33)
False
sage: is_Words(Words('ab'))
True
```

17.40.9 Word utilities

class Factorization()

A list subclass having a nicer representation for factorization of words.

TESTS:

```
sage: f = sage.combinat.words.utils.Factorization()
sage: f == loads(dumps(f))
True
```

clamp(*x, min, max*)

Clamp a value between a maximum and a minimum.

EXAMPLES:

```
sage: from sage.combinat.words.utils import clamp
sage: clamp(0, -1, 1)
0
sage: clamp(-2, 0, 4)
0
sage: clamp(10, 0, 4)
4
```

copy_it(*it*)

Copy an iterator using its builtin __copy__ method if available, otherwise use itertools.tee(). Define __copy__ for your iterators. (See PEP 323)

TESTS:

```
sage: from sage.combinat.words.utils import copy_it
sage: it = iter([1, 2, 3, 4, 5])
sage: it, it2 = copy_it(it)
sage: list(it)
[1, 2, 3, 4, 5]
sage: it2.next()
1
sage: it2, it3 = copy_it(it2)
```

```
sage: list(it2)
[2, 3, 4, 5]
sage: it3.next()
2
```

haslen(*obj*)

Returns true if obj has a properly defined length

EXAMPLES:

```
sage: from sage.combinat.words.utils import haslen
sage: haslen([1, 2, 3])
True
sage: haslen(33)
False
```

TESTS:

```
sage: class test(object):
...       def __len__(self):
...           return -1
...
sage: haslen(test())
False
```

id_f(*x*)

Dummy identity function for when a function is required but none is desired.

TESTS:

```
sage: l = [1, 2, 3]
sage: l is sage.combinat.words.utils.id_f(l)
True
```

is_iterable(*obj*)

Returns true if the obj is iterable.

EXAMPLES:

```
sage: from sage.combinat.words.utils import is_iterable
sage: is_iterable('123')
True
sage: is_iterable([1, 2, 3])
True
sage: is_iterable(xrange(1, 4))
True
sage: is_iterable(33)
False
```

isint(*obj*)

Returns True if obj is an integer or a custom object representing an integer and False otherwise.

EXAMPLES:

```
sage: from sage.combinat.words.utils import isint
sage: isint(1)
True
sage: isint("2")
False
```

```
sage: isint(1.0)
False
```

len_it(*it*)

Returns the number of elements in it.

This function will modify the iterator, so if you want to access the elements later, make a copy of the iterator and pass it to this function.

EXAMPLES:

```
sage: from sage.combinat.words.utils import len_it
sage: len_it(iter([1, 2, 3]))
3
```

peek_it(*it*)

Returns the first element of an iterator and returns an iterator at the same position as the original iterator

EXAMPLES:

```
sage: from sage.combinat.words.utils import peek_it
sage: it = iter([1, 2, 3])
sage: it, n = peek_it(it); n
1
sage: it.next()
1
```

reverse_map(*d*)

Return a new dict with swapped keys and values

EXAMPLES:

```
sage: from sage.combinat.words.utils import reverse_map
sage: reverse_map({'a': 1, 'b': 2}) == {1: 'a', 2: 'b'}
True
```

slice_indices(*s, l*)

Implement slice.indices without bugs.

TESTS:

```
sage: from sage.combinat.words.utils import slice_indices
sage: slice_indices(slice(None), 8)
(0, 8, 1)
sage: slice_indices(slice(1), 8)
(0, 1, 1)
sage: slice_indices(slice(10), 8)
(0, 8, 1)
sage: slice_indices(slice(-4), 8)
(0, 4, 1)
sage: slice_indices(slice(-10), 8)
(0, 0, 1)
sage: slice_indices(slice(1, None), 8)
(1, 8, 1)
sage: slice_indices(slice(10, None), 8)
(8, 8, 1)
sage: slice_indices(slice(-4, None), 8)
(4, 8, 1)
sage: slice_indices(slice(-10, None), 8)
(0, 8, 1)
```

```
sage: slice_indices(slice(None, None, 2), 8)
(0, 8, 2)
sage: slice_indices(slice(None, None, -2), 8)
(7, -1, -2)
```

slice_it(*it, l, key*)

Slice an iterator, supporting negative step sizes by expliciting the elements if needed.

NOTE: The iterator returned depends on it. You must pass in a copy of your iterator if you intend to keep using the original iterator.

TESTS:

```
sage: from sage.combinat.words.utils import slice_it
sage: list(slice_it(iter(range(5)), 5, slice(None)))
[0, 1, 2, 3, 4]
sage: list(slice_it(iter(range(5)), 5, slice(3)))
[0, 1, 2]
sage: list(slice_it(iter(range(5)), 5, slice(-1)))
[0, 1, 2, 3]
sage: list(slice_it(iter(range(5)), 5, slice(2, 4)))
[2, 3]
sage: list(slice_it(iter(range(5)), 5, slice(3, 1, -1)))
[3, 2]
```

slice_ok(*part*)

Returns true if part is a slice and doesn't have funny values.

EXAMPLES:

```
sage: from sage.combinat.words.utils import slice_ok
sage: slice_ok(slice(None))
True
sage: slice_ok(slice(2))
True
sage: slice_ok(slice(1, 2, 3))
True
sage: slice_ok(slice(None, 2, 3))
True
sage: slice_ok(slice(1, None, 3))
True
sage: slice_ok(slice(1, 2, None))
True
sage: slice_ok(slice("a"))
False
sage: slice_ok(slice("a", 1))
False
sage: slice_ok(slice(1, 2, "a"))
False
sage: slice_ok(1)
False
```

sliceable(*obj*)

Returns true if obj is completely sliceable, including negative step sizes

EXAMPLES:

```
sage: from sage.combinat.words.utils import sliceable
sage: sliceable([1, 2, 3])
```

```
True
sage: sliceable(33)
False
```

TESTS:

```
sage: class test(object):
...       def __getitem__(self, key):
...           return islice(iter([1]), *key)
...
sage: sliceable(test())
False
```

17.41 Miscellaneous

class DoublyLinkedList(*l*)

A doubly linked list class that provides constant time hiding and unhiding of entries.

Note that this list's indexing is 1-based.

EXAMPLES:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3]); dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
sage: dll.hide(1); dll
Doubly linked list of [1, 2, 3]: [2, 3]
sage: dll.unhide(1); dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
sage: dll.hide(2); dll
Doubly linked list of [1, 2, 3]: [1, 3]
sage: dll.unhide(2); dll
Doubly linked list of [1, 2, 3]: [1, 2, 3]
```

head()

TESTS:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3])
sage: dll.head()
1
sage: dll.hide(1)
sage: dll.head()
2
```

hide(*i*)

TESTS:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3])
sage: dll.hide(1)
sage: list(dll)
[2, 3]
```

next(*j*)

TESTS:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3])
sage: dll.next(1)
2
sage: dll.hide(2)
```

```
sage: dll.next(1)
3
```

prev(*j*)

TESTS:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3])
sage: dll.prev(3)
2
sage: dll.hide(2)
sage: dll.prev(3)
1
```

unhide(*i*)

TESTS:

```
sage: dll = sage.combinat.misc.DoublyLinkedList([1,2,3])
sage: dll.hide(1); dll.unhide(1)
sage: list(dll)
[1, 2, 3]
```

class **IterableFunctionCall**(*f, *args, **kwargs*)

This class wraps functions with a yield statement (generators) by an object that can be iterated over. For example,

EXAMPLES:

```
sage: def f(): yield 'a'; yield 'b'
```

This does not work:

```
sage: for z in f: print z
...
TypeError: 'function' object is not iterable
```

Use IterableFunctionCall if you want something like the above to work:

```
sage: from sage.combinat.misc import IterableFunctionCall
sage: g = IterableFunctionCall(f)
sage: for z in g: print z
a
b
```

If your function takes arguments, just put them after the function name. You needn't enclose them in a tuple or anything, just put them there:

```
sage: def f(n, m): yield 'a' * n; yield 'b' * m; yield 'foo'
sage: g = IterableFunctionCall(f, 2, 3)
sage: for z in g: print z
aa
bbb
foo
```

check_integer_list_constraints(*l, **kwargs*)

EXAMPLES:

```
sage: from sage.combinat.misc import check_integer_list_constraints
sage: cilc = check_integer_list_constraints
sage: l = [[2,1,3],[1,2],[3,3],[4,1,1]]
sage: cilc(l, min_part=2)
```

```
[[3, 3]]
sage: cilc(l, max_part=2)
[[1, 2]]
sage: cilc(l, length=2)
[[1, 2], [3, 3]]
sage: cilc(l, max_length=2)
[[1, 2], [3, 3]]
sage: cilc(l, min_length=3)
[[2, 1, 3], [4, 1, 1]]
sage: cilc(l, max_slope=0)
[[3, 3], [4, 1, 1]]
sage: cilc(l, min_slope=1)
[[1, 2]]
sage: cilc(l, outer=[2,2])
[[1, 2]]
sage: cilc(l, inner=[2,2])
[[3, 3]]


sage: cilc([1,2,3], length=3, singleton=True)
[1, 2, 3]
sage: cilc([1,2,3], length=2, singleton=True) is None
True
```

umbral_operation(*poly*)

Returns the umbral operation $\downarrow$ applied to poly.

The umbral operation replaces each instance of $x_i^{a_i}$ with $x_i * (x_i - 1) * \cdots * (x_i - a_i + 1)$.

EXAMPLES:

```
sage: P = PolynomialRing(QQ, 2, 'x')
sage: x = P.gens()
sage: from sage.combinat.misc import umbral_operation
sage: umbral_operation(x[0]^3) == x[0]*(x[0]-1)*(x[0]-2)
True
sage: umbral_operation(x[0]*x[1])
x0*x1
sage: umbral_operation(x[0]+x[1])
x0 + x1
sage: umbral_operation(x[0]^2*x[1]^2) == x[0]*(x[0]-1)*x[1]*(x[1]-1)
True
```

CHAPTER

EIGHTEEN

NUMERICAL OPTIMIZATION

18.1 Knapsack Problems

This module implements a number of solutions to various knapsack problems, otherwise known as linear integer programming problems. Solutions to the following knapsack problems are implemented:

- Solving the subset sum problem for super-increasing sequences.

AUTHORS:

- Minh Van Nguyen (2009-04): initial version

EXAMPLES:

We can test for whether or not a sequence is super-increasing:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: seq = Superincreasing(L)
sage: seq
Super-increasing sequence of length 8
sage: seq.is_superincreasing()
True
sage: Superincreasing().is_superincreasing([1,3,5,7])
False
```

Solving the subset sum problem for a super-increasing sequence and target sum:

```
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).subset_sum(98)
[69, 21, 5, 2, 1]
```

class **Superincreasing**(*seq=None*)

A class for super-increasing sequences.

Let $L = (a_1, a_2, a_3, \ldots, a_n)$ be a non-empty sequence of non-negative integers. Then L is said to be super-increasing if each a_i is strictly greater than the sum of all previous values. That is, for each $a_i \in L$ the sequence L must satisfy the property

$$a_i > \sum_{k=1}^{i-1} a_k$$

in order to be called a super-increasing sequence, where $|L| \geq 2$. If L has only one element, it is also defined to be a super-increasing sequence.

If `seq` is `None`, then construct an empty sequence. By defintion, this empty sequence is not super-increasing.

INPUT:

- `seq` – (default: `None`) a non-empty sequence.

EXAMPLES:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
True
sage: Superincreasing().is_superincreasing([1,3,5,7])
False
sage: seq = Superincreasing(); seq
An empty sequence.
sage: seq = Superincreasing([1, 3, 6]); seq
Super-increasing sequence of length 3
sage: seq = Superincreasing(list([1, 2, 5, 21, 69, 189, 376, 919])); seq
Super-increasing sequence of length 8
```

is_superincreasing(*seq=None*)

Determine whether or not `seq` is super-increasing.

If `seq=None` then determine whether or not `self` is super-increasing.

Let $L = (a_1, a_2, a_3, \ldots, a_n)$ be a non-empty sequence of non-negative integers. Then L is said to be super-increasing if each a_i is strictly greater than the sum of all previous values. That is, for each $a_i \in L$ the sequence L must satisfy the property

$$a_i > \sum_{k=1}^{i-1} a_k$$

in order to be called a super-increasing sequence, where $|L| \geq 2$. If L has exactly one element, then it is also defined to be a super-increasing sequence.

INPUT:

- `seq` – (default: `None`) a sequence to test

OUTPUT:

- If `seq` is `None`, then test `self` to determine whether or not it is super-increasing. In that case, return `True` if `self` is super-increasing; `False` otherwise.
- If `seq` is not `None`, then test `seq` to determine whether or not it is super-increasing. Return `True` if `seq` is super-increasing; `False` otherwise.

EXAMPLES:

By definition, an empty sequence is not super-increasing:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: Superincreasing().is_superincreasing([])
False
sage: Superincreasing().is_superincreasing()
False
sage: Superincreasing().is_superincreasing(tuple())
False
sage: Superincreasing().is_superincreasing(())
False
```

But here is an example of a super-increasing sequence:

```
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
True
sage: L = (1, 2, 5, 21, 69, 189, 376, 919)
sage: Superincreasing(L).is_superincreasing()
True
```

A super-increasing sequence can have zero as one of its elements:

```
sage: L = [0, 1, 2, 4]
sage: Superincreasing(L).is_superincreasing()
True
```

A super-increasing sequence can be of length 1:

```
sage: Superincreasing([randint(0, 100)]).is_superincreasing()
True
```

TESTS:

The sequence must contain only integers:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1.0, 2.1, pi, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
...
TypeError: Element e (= 1.00000000000000) of seq must be a non-negative integer.
sage: L = [1, 2.1, pi, 21, 69, 189, 376, 919]
sage: Superincreasing(L).is_superincreasing()
...
TypeError: Element e (= 2.10000000000000) of seq must be a non-negative integer.
```

largest_less_than(*N*)

Return the largest integer in the sequence `self` that is less than or equal to `N`.

This function narrows down the candidate solution using a binary trim, similar to the way binary search halves the sequence at each iteration.

INPUT:

- `N` – integer; the target value to search for.
- `seq` – the non-empty sequence in which to search.

OUTPUT:

The largest integer in `self` that is less than or equal to `N`. If no solution exists, then return `None`.

EXAMPLES:

When a solution is found, return it:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [2, 3, 7, 25, 67, 179, 356, 819]
sage: Superincreasing(L).largest_less_than(207)
179
sage: L = (2, 3, 7, 25, 67, 179, 356, 819)
sage: Superincreasing(L).largest_less_than(2)
2
```

But if no solution exists, return `None`:

```
sage: L = [2, 3, 7, 25, 67, 179, 356, 819]
sage: Superincreasing(L).largest_less_than(-1) == None
True
```

TESTS:

The target `N` must be an integer:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [2, 3, 7, 25, 67, 179, 356, 819]
sage: Superincreasing(L).largest_less_than(2.30)
...
TypeError: N (= 2.30000000000000) must be an integer.
```

The sequence that `self` represents must also be non-empty:

```
sage: Superincreasing([]).largest_less_than(2)
...
ValueError: seq must be a super-increasing sequence
sage: Superincreasing(list()).largest_less_than(2)
...
ValueError: seq must be a super-increasing sequence
```

subset_sum(*N*)

Solving the subset sum problem for a super-increasing sequence.

Let $S = (s_1, s_2, s_3, \ldots, s_n)$ be a non-empty sequence of non-negative integers, and let $N \in \mathbf{Z}$ be non-negative. The subset sum problem asks for a subset $A \subseteq S$ all of whose elements sum to N. This method specializes the subset sum problem to the case of super-increasing sequences. If a solution exists, then it is also a super-increasing sequence.

Note: This method only solves the subset sum problem for super-increasing sequences. In general, solving the subset sum problem for an arbitrary sequence is known to be computationally hard.

INPUT:

- `N` – a non-negative integer.

OUTPUT:

- A non-empty subset of `self` whose elements sum to `N`. This subset is also a super-increasing sequence. If no such subset exists, then return the empty list.

ALGORITHMS:

The algorithm used is adapted from page 355 of [HPS08].

EXAMPLES:

Solving the subset sum problem for a super-increasing sequence and target sum:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [1, 2, 5, 21, 69, 189, 376, 919]
sage: Superincreasing(L).subset_sum(98)
[69, 21, 5, 2, 1]
```

TESTS:

The target `N` must be a non-negative integer:

```
sage: from sage.numerical.knapsack import Superincreasing
sage: L = [0, 1, 2, 4]
sage: Superincreasing(L).subset_sum(-6)
...
TypeError: N (= -6) must be a non-negative integer.
sage: Superincreasing(L).subset_sum(-6.2)
...
TypeError: N (= -6.20000000000000) must be a non-negative integer.
```

The sequence that `self` represents must only contain non-negative integers:

```
sage: L = [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1]
sage: Superincreasing(L).subset_sum(1)
...
TypeError: Element e (= -10) of seq must be a non-negative integer.
```

REFERENCES:

18.2 Numerical Root Finding and Optimization

AUTHOR:

- William Stein (2007): initial version

find_fit (*data, model, initial_guess=None, parameters=None, variables=None, solution_dict=False*)

Finds numerical estimates for the parameters of the function model to give a best fit to data.

INPUT:

- `data` – A two dimensional table of floating point numbers of the form $[[x_{1,1}, x_{1,2}, \ldots, x_{1,k}, f_1], [x_{2,1}, x_{2,2}, \ldots, x_{2,k}, f_2], \ldots, [x_{n,1}, x_{n,2}, \ldots, x_{n,k}, f_n]]$ given as either a list of lists, matrix, or numpy array.
- `model` – Either a symbolic expression, symbolic function, or a Python function. `model` has to be a function of the variables $(x_1, x_2, \ldots, x_k)$ and free parameters $(a_1, a_2, \ldots, a_l)$.
- `initial_guess` – (default: `None`) Initial estimate for the parameters $(a_1, a_2, \ldots, a_l)$, given as either a list, tuple, vector or numpy array. If `None`, the default estimate for each parameter is 1.
- `parameters` – (default: `None`) A list of the parameters $(a_1, a_2, \ldots, a_l)$. If model is a symbolic function it is ignored, and the free parameters of the symbolic function are used.
- `variables` – (default: `None`) A list of the variables $(x_1, x_2, \ldots, x_k)$. If model is a symbolic function it is ignored, and the variables of the symbolic function are used.
- `solution_dict` – (default: `False`) if `True`, return the solution as a dictionary rather than an equation.

EXAMPLES:

First we create some datapoints of a sine function with some random perturbations:

```
sage: data = [(i, 1.2 * sin(0.5*i-0.2) + 0.1 * normalvariate(0, 1)) for i in xsrange(0, 4*pi, 0.
sage: var('a, b, c, x')
(a, b, c, x)
```

We define a function with free parameters a, b and c:

```
sage: model(x) = a * sin(b * x - c)
```

We search for the parameters that give the best fit to the data:

```
sage: find_fit(data, model)
[a == 1.21..., b == 0.49..., c == 0.19...]
```

We can also use a Python function for the model:

```
sage: def f(x, a, b, c): return a * sin(b * x - c)
sage: fit = find_fit(data, f, parameters = [a, b, c], variables = [x], solution_dict = True)
sage: fit[a], fit[b], fit[c]
(1.21..., 0.49..., 0.19...)
```

We search for a formula for the n-th prime number:

```
sage: dataprime = [(i, nth_prime(i)) for i in xrange(1, 5000, 100)]
sage: find_fit(dataprime, a * x * log(b * x), parameters = [a, b], variables = [x])
[a == 1.11..., b == 1.24...]
```

ALGORITHM:

Uses `scipy.optimize.leastsq` which in turn uses MINPACK's lmdif and lmder algorithms.

find_maximum_on_interval (*f, a, b, tol=1.48e-08, maxfun=500*)

Numerically find the maximum of the expression f on the interval $[a, b]$ (or $[b, a]$) along with the point at which the maximum is attained.

See the documentation for `find_minimum_on_interval()` for more details.

EXAMPLES:

```
sage: f = lambda x: x*cos(x)
sage: find_maximum_on_interval(f, 0,5)
(0.561096338191..., 0.8603335890...)
sage: find_maximum_on_interval(f, 0,5, tol=0.1, maxfun=10)
(0.561090323458..., 0.857926501456...)
```

find_minimum_on_interval (*f, a, b, tol=1.48e-08, maxfun=500*)

Numerically find the minimum of the expression `self` on the interval $[a, b]$ (or $[b, a]$) and the point at which it attains that minimum. Note that `self` must be a function of (at most) one variable.

INPUT:

- `a`, `b` – endpoints of interval on which to minimize self.
- `tol` – the convergence tolerance
- `maxfun` – maximum function evaluations

OUTPUT:

- `minval` – (float) the minimum value that self takes on in the interval $[a, b]$
- `x` – (float) the point at which self takes on the minimum value

EXAMPLES:

```
sage: f = lambda x: x*cos(x)
sage: find_minimum_on_interval(f, 1, 5)
(-3.28837139559..., 3.4256184695...)
sage: find_minimum_on_interval(f, 1, 5, tol=1e-3)
(-3.28837136189098..., 3.42575079030572...)
sage: find_minimum_on_interval(f, 1, 5, tol=1e-2, maxfun=10)
(-3.28837084598..., 3.4250840220...)
sage: show(plot(f, 0, 20))
sage: find_minimum_on_interval(f, 1, 15)
(-9.4772942594..., 9.5293344109...)
```

ALGORITHM:

Uses scipy.optimize.fminbound which uses Brent's method.

AUTHOR:

- William Stein (2007-12-07)

find_root (*f, a, b, xtol=9.9999999999999998e-13, rtol=4.5000000000000002e-16, maxiter=100, full_output=False*)

Numerically find a root of `f` on the closed interval $[a, b]$ (or $[b, a]$) if possible, where `f` is a function in the one variable.

INPUT:

- `f` – a function of one variable or symbolic equality
- `a`, `b` – endpoints of the interval
- `xtol`, `rtol` – the routine converges when a root is known to lie within `xtol` of the value return. Should be ≥ 0. The routine modifies this to take into account the relative precision of doubles.

- `maxiter` – integer; if convergence is not achieved in `maxiter` iterations, an error is raised. Must be ≥ 0.
- `full_output` – bool (default: `False`), if `True`, also return object that contains information about convergence.

EXAMPLES:

An example involving an algebraic polynomial function:

```
sage: R.<x> = QQ[]
sage: f = (x+17)*(x-3)*(x-1/8)^3
sage: find_root(f, 0,4)
2.9999999999999951
sage: find_root(f, 0,1)  # note -- precision of answer isn't very good on some machines.
0.124999...
sage: find_root(f, -20,-10)
-17.0
```

In Pomerance book on primes he asserts that the famous Riemann Hypothesis is equivalent to the statement that the function $f(x)$ defined below is positive for all $x \geq 2.01$:

```
sage: def f(x):
...       return sqrt(x) * log(x) - abs(Li(x) - prime_pi(x))
```

We find where f equals, i.e., what value that is slightly smaller than 2.01 that could have been used in the formulation of the Riemann Hypothesis:

```
sage: find_root(f, 2, 4, rtol=0.0001)
2.0082590205656166
```

This agrees with the plot:

```
sage: plot(f,2,2.01)
```

linear_program(*c, G, h, A=None, b=None*)

Solves the dual linear programs:

- Minimize $c'x$ subject to $Gx + s = h$, $Ax = b$, and $s \geq 0$ where $'$ denotes transpose.
- Maximize $-h'z - b'y$ subject to $G'z + A'y + c = 0$ and $z \geq 0$.

INPUT:

- `c` – a vector
- `G` – a matrix
- `h` – a vector
- `A` – a matrix
- `b` — a vector

These can be over any field that can be turned into a floating point number.

OUTPUT:

A dictionary `sol` with keys `x`, `s`, `y`, `z` corresponding to the variables above:

- `sol['x']` – the solution to the linear program
- `sol['s']` – the slack variables for the solution
- `sol['z']`, `sol['y']` – solutions to the dual program

EXAMPLES:

First, we minimize $-4x_1 - 5x_2$ subject to $2x_1 + x_2 \leq 3$, $x_1 + 2x_2 \leq 3$, $x_1 \geq 0$, and $x_2 \geq 0$:

```
sage: c=vector(RDF,[-4,-5])
sage: G=matrix(RDF,[[2,1],[1,2],[-1,0],[0,-1]])
sage: h=vector(RDF,[3,3,0,0])
sage: sol=linear_program(c,G,h)
sage: sol['x']
(0.999..., 1.000...)
```

Next, we maximize $x + y - 50$ subject to $50x + 24y \leq 2400$, $30x + 33y \leq 2100$, $x \geq 45$, and $y \geq 5$:

```
sage: v=vector([-1.0,-1.0,-1.0])
sage: m=matrix([[50.0,24.0,0.0],[30.0,33.0,0.0],[-1.0,0.0,0.0],[0.0,-1.0,0.0],[0.0,0.0,1.0],[0.0
sage: h=vector([2400.0,2100.0,-45.0,-5.0,1.0,-1.0])
sage: sol=linear_program(v,m,h)
sage: sol['x']
(45.000000..., 6.2499999...3, 1.00000000...)
```

minimize (*func, x0, gradient=None, hessian=None, algorithm='default', **args*)

This function is an interface to a variety of algorithms for computing the minimum of a function of several variables.

INPUT:

- `func` – Either a symbolic function or a Python function whose argument is a tuple with n components
- `x0` – Initial point for finding minimum.
- `gradient` – Optional gradient function. This will be computed automatically for symbolic functions. For Python functions, it allows the use of algorithms requiring derivatives. It should accept a tuple of arguments and return a numpy array containing the partial derivatives at that point.
- `hessian` – Optional hessian function. This will be computed automatically for symbolic functions. For Python functions, it allows the use of algorithms requiring derivatives. It should accept a tuple of arguments and return a numpy array containing the second partial derivatives of the function.
- `algorithm` – String specifying algorithm to use. Options are `'default'` (for Python functions, the simplex method is the default) (for symbolic functions bfgs is the default):
  - `'simplex'`
  - `'powell'`
  - `'bfgs'` – (broyden-fletcher-goldfarb-shannon) requires `gradient`
  - `'cg'` – (conjugate-gradient) requires gradient
  - `'ncg'` – (newton-conjugate gradient) requires gradient and hessian

EXAMPLES:

```
sage: vars=var('x y z')
sage: f=100*(y-x^2)^2+(1-x)^2+100*(z-y^2)^2+(1-y)^2
sage: minimize(f,[.1,.3,.4],disp=0)
(1.00..., 1.00..., 1.00...)

sage: minimize(f,[.1,.3,.4],algorithm="ncg",disp=0)
(0.9999999..., 0.999999..., 0.999999...)
```

Same example with just Python functions:

```
sage: def rosen(x): # The Rosenbrock function
...       return sum(100.0r*(x[1r:]-x[:-1r]**2.0r)**2.0r + (1r-x[:-1r])**2.0r)
sage: minimize(rosen,[.1,.3,.4],disp=0)
(1.00..., 1.00..., 1.00...)
```

Same example with a pure Python function and a Python function to compute the gradient:

```
sage: def rosen(x): # The Rosenbrock function
...       return sum(100.0r*(x[1r:]-x[:-1r]**2.0r)**2.0r + (1r-x[:-1r])**2.0r)
sage: import numpy
sage: from numpy import zeros
sage: def rosen_der(x):
...       xm = x[1r:-1r]
...       xm_m1 = x[:-2r]
...       xm_p1 = x[2r:]
...       der = zeros(x.shape,dtype=float)
...       der[1r:-1r] = 200r*(xm-xm_m1**2r) - 400r*(xm_p1 - xm**2r)*xm - 2r*(1r-xm)
...       der[0] = -400r*x[0r]*(x[1r]-x[0r]**2r) - 2r*(1r-x[0])
...       der[-1] = 200r*(x[-1r]-x[-2r]**2r)
...       return der
sage: minimize(rosen,[.1,.3,.4],gradient=rosen_der,algorithm="bfgs",disp=0)
(1.00...,  1.00..., 1.00...)
```

minimize_constrained(*func, cons, x0, gradient=None, algorithm='default', **args*)

Minimize a function with constraints.

INPUT:

- `func` – Either a symbolic function, or a Python function whose argument is a tuple with n components
- `cons` – constraints. This should be either a function or list of functions that must be positive. Alternatively, the constraints can be specified as a list of intervals that define the region we are minimizing in. If the constraints are specified as functions, the functions should be functions of a tuple with n components (assuming n variables). If the constraints are specifed as a list of intervals and there are no constraints for a given variable, that component can be (`None`, `None`).
- `x0` – Initial point for finding minimum
- `algorithm` – Optional, specify the algorithm to use:
  - `'default'` – default choices
  - `'l-bfgs-b'` – only effective if you specify bound constraints. See [ZBN97].
- `gradient` – Optional gradient function. This will be computed automatically for symbolic functions. This is only used when the constraints are specified as a list of intervals.

EXAMPLES:

Let us maximize $x+y-50$ subject to the following constraints: $50x+24y \leq 2400$, $30x+33y \leq 2100$, $x \geq 45$, and $y \geq 5$:

```
sage: y = var('y')
sage: f = lambda p: -p[0]-p[1]+50
sage: c_1 = lambda p: p[0]-45
sage: c_2 = lambda p: p[1]-5
sage: c_3 = lambda p: -50*p[0]-24*p[1]+2400
sage: c_4 = lambda p: -30*p[0]-33*p[1]+2100
sage: a = minimize_constrained(f,[c_1,c_2,c_3,c_4],[2,3])
sage: a
(45.0, 6.25)
```

Let's find a minimum of $\sin(xy)$:

```
sage: x,y = var('x y')
sage: f = sin(x*y)
sage: minimize_constrained(f, [(None,None),(4,10)],[5,5])
(4.8..., 4.8...)
```

Check, if L-BFGS-B finds the same minimum:

```
sage: minimize_constrained(f, [(None,None),(4,10)],[5,5], algorithm='l-bfgs-b')
(4.7..., 4.9...)
```

Rosenbrock funcion, [http://en.wikipedia.org/wiki/Rosenbrock_function]:

```
sage: from scipy.optimize import rosen, rosen_der
sage: minimize_constrained(rosen, [(-50,-10),(5,10)],[1,1],gradient=rosen_der,algorithm='l-bfgs-
(-10.0, 10.0)
sage: minimize_constrained(rosen, [(-50,-10),(5,10)],[1,1],algorithm='l-bfgs-b')
(-10.0, 10.0)
```

REFERENCES:

CHAPTER

NINETEEN

PROBABILITY

19.1 Random variables and probability spaces

This introduces a class of random variables, with the focus on discrete random variables (i.e. on a discrete probability space). This avoids the problem of defining a measure space and measurable functions.

class **DiscreteProbabilitySpace** (*X, P, codomain=None, check=False*)

The discrete probability space

entropy ()

The entropy of the probability space.

set ()

The set of values of the probability space taking possibly nonzero probability (a subset of the domain).

class **DiscreteRandomVariable** (*X, f, codomain=None, check=False*)

A random variable on a discrete probability space.

correlation (*other*)

The correlation of the probability space X = self with Y = other.

covariance (*other*)

The covariance of the discrete random variable X = self with Y = other.

Let S be the probability space of X = self, with probability function p, and $E(X)$ be the expectation of X. Then the variance of X is:

$$\text{cov}(X,Y) = E((X - E(X) * (Y - E(Y)) = \sum_{x \in S} p(x)(X(x) - E(X))(Y(x) - E(Y))$$

expectation ()

The expectation of the discrete random variable, namely $\sum_{x \in S} p(x)X[x]$, where X = self and S is the probability space of X.

function ()

The function defining the random variable.

standard_deviation ()

The standard deviation of the discrete random variable.

Let S be the probability space of X = self, with probability function p, and $E(X)$ be the expectation of X. Then the standard deviation of X is defined to be

$$\sigma(X) = \sqrt{\sum_{x \in S} p(x)(X(x) - E(x)) * *2}$$

translation_correlation (*other, map*)

The correlation of the probability space X = self with image of Y = other under map.

translation_covariance(*other, map*)

The covariance of the probability space X = self with image of Y = other under the given map of the probability space.

Let S be the probability space of X = self, with probability function p, and $E(X)$ be the expectation of X. Then the variance of X is:

$$\mathrm{cov}(X,Y) = E((X - E(X) * (Y - E(Y)) = \sum_{x \in S} p(x)(X(x) - E(X))(Y(x) - E(Y))$$

translation_expectation(*map*)

The expectation of the discrete random variable, namely $\sum_{x \in S} p(x)X[e(x)]$, where X = self, S is the probability space of X, and e = map.

translation_standard_deviation(*map*)

The standard deviation of the translated discrete random variable $X \circ e$, where X = self and e = map.

Let S be the probability space of X = self, with probability function p, and $E(X)$ be the expectation of X. Then the standard deviation of X is defined to be

$$\sigma(X) = \sqrt{\sum_{x \in S} p(x)(X(x) - E(x)) * *2}$$

translation_variance(*map*)

The variance of the discrete random variable $X \circ e$, where X = self, and e = map.

Let S be the probability space of X = self, with probability function p, and $E(X)$ be the expectation of X. Then the variance of X is:

$$\mathrm{var}(X) = E((X - E(x))^2) = \sum_{x \in S} p(x)(X(x) - E(x))^2$$

variance()

The variance of the discrete random variable.

Let S be the probability space of X = self, with probability function p, and $E(X)$ be the expectation of X. Then the variance of X is:

$$\mathrm{var}(X) = E((X - E(x))^2) = \sum_{x \in S} p(x)(X(x) - E(x))^2$$

class **ProbabilitySpace_generic**(*domain, RR*)

A probability space.

domain()

class **RandomVariable_generic**(*X, RR*)

A random variable.

codomain()

domain()

field()

probability_space()

is_DiscreteProbabilitySpace(S)

is_DiscreteRandomVariable(X)

is_ProbabilitySpace(S)

is_RandomVariable(X)

CHAPTER

TWENTY

CATEGORY THEORY

20.1 Categories

AUTHORS:

- David Kohel and William Stein

Every Sage object lies in a category. Categories in Sage are modeled on the mathematical idea of category, and are distinct from Python classes, which are a programming construct.

In most cases, typing `x.category()` returns the category to which x belongs. If C is a category and x is any object, $C(x)$ tries to make an object in C from x.

EXAMPLES: We create a couple of categories.

```
sage: Sets()
Category of sets
sage: GSets(AbelianGroup([2,4,9]))
Category of G-sets for Multiplicative Abelian Group isomorphic to C2 x C4 x C9
sage: Semigroups()
Category of semigroups
sage: VectorSpaces(FiniteField(11))
Category of vector spaces over Finite Field of size 11
sage: Ideals(IntegerRing())
Category of ring ideals in Integer Ring
```

The default category for elements x of an objects O is the category of all objects of O. For example,

```
sage: V = VectorSpace(RationalField(), 3)
sage: x = V.gen(1)
sage: x.category()
Category of elements of Vector space of dimension 3 over Rational Field
```

class **Category**(*s=None*)

The base class for all categories.

category()

is_abelian()

is_subcategory(*c*)

Returns True if self is naturally embedded as a subcategory of c.

EXAMPLES:

```
sage: Rings  = Rings()
sage: AbGrps = AbelianGroups()
sage: Rings.is_subcategory(AbGrps)
True
sage: AbGrps.is_subcategory(Rings)
False
```

The `is_subcategory` function takes into account the base.

```
sage: M3 = VectorSpaces(FiniteField(3))
sage: M9 = VectorSpaces(FiniteField(9, 'a'))
sage: M3.is_subcategory(M9)
False
```

short_name()

class Category_uniq(*s=None*)

class Sets(*s=None*)

The category of sets.

EXAMPLES: sage: Sets() Category of sets

is_Category(*x*)

Returns True if x is a category.

class uniq()

class uniq1()

20.2 Homsets

AUTHORS:

- David Kohel and William Stein
- David Joyner (2005-12-17): added examples
- William Stein (2006-01-14): Changed from Homspace to Homset.

End(*X*, *cat=None*)

Create the set of endomorphisms of X in the category cat.

INPUT:

- `X` - anything
- `cat` - (optional) category in which to coerce X

OUTPUT: a set of endomorphisms in cat

EXAMPLES:

```
sage: V = VectorSpace(QQ, 3)
sage: End(V)
Set of Morphisms from Vector space of dimension 3 over Rational
Field to Vector space of dimension 3 over Rational Field in
Category of vector spaces over Rational Field
```

```
sage: G = SymmetricGroup(3)
sage: S = End(G); S
Set of Morphisms from SymmetricGroup(3) to SymmetricGroup(3) in Category of groups
sage: from sage.categories.homset import is_Endset
sage: is_Endset(S)
True
sage: S.domain()
Symmetric group of order 3! as a permutation group
```

Homsets are *not* objects in their category. They are currently sets.

```
sage: S.category()
Category of sets
sage: S.domain().category()
Category of groups
```

Hom(*X, Y, cat=None*)

Create the space of homomorphisms from X to Y in the category cat.

INPUT:

- `X` - anything
- `Y` - anything
- `cat` - (optional) category in which the morphisms must be

OUTPUT: a homset in cat

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: Hom(V, V)
Set of Morphisms from Vector space of dimension 3 over Rational
Field to Vector space of dimension 3 over Rational Field in
Category of vector spaces over Rational Field
sage: G = SymmetricGroup(3)
sage: Hom(G, G)
Set of Morphisms from SymmetricGroup(3) to SymmetricGroup(3) in Category of groups
sage: Hom(ZZ, QQ, Sets())
Set of Morphisms from Integer Ring to Rational Field in Category of sets
```

class **Homset**(*X, Y, cat=None, check=True*)

The class for collections of morphisms in a category.

EXAMPLES:

```
sage: H = Hom(QQ^2, QQ^3)
sage: loads(H.dumps()) == H
True
sage: E = End(AffineSpace(2, names='x,y'))
sage: loads(E.dumps()) == E
True
```

codomain()

coerce_map_from_c(*R*)

domain()

get_action_c(*R, op, self_on_left*)

homset_category()
Return the category that this is a Hom in, i.e., this is typically the category of the domain or codomain object.

EXAMPLES:

```
sage: H = Hom(SymmetricGroup(4), SymmetricGroup(7))
sage: H.homset_category()
Category of groups
```

identity()

is_endomorphism_set()
Return True if the domain and codomain of self are the same object.

natural_map()

reversed()
Return the corresponding homset, but with the domain and codomain reversed.

EXAMPLES:

```
sage: H = Hom(ZZ^2, ZZ^3); H
Set of Morphisms from Ambient free module of rank 2 over the principal ideal domain Integer
sage: type(H)
<class 'sage.modules.free_module_homspace.FreeModuleHomspace'>
sage: H.reversed()
Set of Morphisms from Ambient free module of rank 3 over the principal ideal domain Integer
sage: type(H.reversed())
<class 'sage.modules.free_module_homspace.FreeModuleHomspace'>
```

class HomsetWithBase(*X, Y, cat=None, check=True, base=None*)

end(*X, f*)
Return End(X)(f), where f is data that defines an element of End(X).

EXAMPLES:

```
sage: R, x = PolynomialRing(QQ,'x').objgen()
sage: phi = end(R, [x + 1])
sage: phi
Ring endomorphism of Univariate Polynomial Ring in x over Rational Field
  Defn: x |--> x + 1
sage: phi(x^2 + 5)
x^2 + 2*x + 6
```

hom(*X, Y, f*)
Return Hom(X,Y)(f), where f is data that defines an element of Hom(X,Y).

EXAMPLES:

```
sage: R, x = PolynomialRing(QQ,'x').objgen()
sage: phi = hom(R, QQ, [2])
sage: phi(x^2 + 3)
7
```

is_Endset(*x*)
Return True if x is a set of endomorphisms in a category.

is_Homset(*x*)
Return True if x is a set of homomorphisms in a category.

20.3 Morphisms

AUTHORS:

- William Stein: initial version
- David Joyner (12-17-2005): added examples
- Robert Bradshaw (2007-06-25) Pyrexification

class CallMorphism()

class FormalCoercionMorphism()

class IdentityMorphism()

class Morphism()

category()

is_endomorphism()

pushforward()

class SetMorphism()

is_Morphism()

make_morphism()

20.4 Functors

AUTHORS:

- David Kohel and William Stein
- David Joyner (2005-12-17): examples
- Robert Bradshaw (2007-06-23): Pyrexify

ForgetfulFunctor()
Construct the forgetful function from one category to another.

EXAMPLES:

```
sage: rings = Rings()
sage: abgrps = AbelianGroups()
sage: F = ForgetfulFunctor(rings, abgrps)
sage: F
The forgetful functor from Rings to AbelianGroups
```

class ForgetfulFunctor_generic()

class Functor()
EXAMPLES:

```
sage: rings  = Rings()
sage: abgrps = AbelianGroups()
sage: F = ForgetfulFunctor(rings, abgrps)
sage: F.domain()
Category of rings
sage: F.codomain()
Category of abelian groups
sage: from sage.categories.functor import is_Functor
sage: is_Functor(F)
True
sage: I = IdentityFunctor(abgrps)
sage: I
The identity functor on AbelianGroups
sage: I.domain()
Category of abelian groups
sage: is_Functor(I)
True
```

codomain()

domain()

IdentityFunctor()

class **IdentityFunctor_generic**()

is_Functor()

CHAPTER

TWENTYONE

MONOIDS

Sage supports free monoids and free abelian monoids in any finite number of indeterminates.

21.1 Free Monoids

AUTHORS:

- David Kohel (2005-09)

Sage supports free monoids on any prescribed finite number $n \geq 0$ of generators. Use the `FreeMonoid` function to create a free monoid, and the `gen` and `gens` functions to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional `names` argument to the `FreeMonoid` function.

class FreeMonoidFactory()

Returns a free monoid on n generators.

INPUT:

- `n` - integer
- `names` - names of generators

OUTPUT: free abelian monoid

EXAMPLES:

```
sage: FreeMonoid(0,'')
Free monoid on 0 generators ()
sage: F.<a,b,c,d,e> = FreeMonoid(5); F
Free monoid on 5 generators (a, b, c, d, e)
sage: F(1)
1
sage: mul([ a, b, a, c, b, d, c, d ], F(1))
a*b*a*c*b*d*c*d
```

create_key(*n, names*)

create_object(*version, key*)

class FreeMonoid_class(*n, names=None*)

The free monoid on n generators.

gen(*i=0*)

The i-th generator of the monoid.

INPUT:

- i - integer (default: 0)

EXAMPLES:

```
sage: F = FreeMonoid(3, 'a')
sage: F.gen(1)
a1
sage: F.gen(2)
a2
sage: F.gen(5)
...
IndexError: Argument i (= 5) must be between 0 and 2.
```

ngens()

The number of free generators of the monoid.

EXAMPLES:

```
sage: F = FreeMonoid(2005, 'a')
sage: F.ngens()
2005
```

is_FreeMonoid(*x*)

Return True if *x* is a free monoid.

EXAMPLES:

```
sage: from sage.monoids.free_monoid import is_FreeMonoid
sage: is_FreeMonoid(5)
False
sage: is_FreeMonoid(FreeMonoid(7,'a'))
True
sage: is_FreeMonoid(FreeAbelianMonoid(7,'a'))
False
sage: is_FreeMonoid(FreeAbelianMonoid(0,''))
False
```

21.2 Monoid Elements

AUTHORS:

- David Kohel (2005-09-29)

Elements of free monoids are represented internally as lists of pairs of integers.

class **FreeMonoidElement**(*F, x, check=True*)

Element of a free monoid.

EXAMPLES:

```
sage: a = FreeMonoid(5, 'a').gens()
sage: x = a[0]*a[1]*a[4]**3
sage: x**3
a0*a1*a4^3*a0*a1*a4^3*a0*a1*a4^3
sage: x**0
1
sage: x**(-1)
...
TypeError: bad operand type for unary ~: 'FreeMonoidElement'
```

is_FreeMonoidElement(x)

21.3 Free abelian monoids

AUTHORS:

- David Kohel (2005-09)

Sage supports free abelian monoids on any prescribed finite number $n \geq 0$ of generators. Use the `FreeAbelianMonoid` function to create a free abelian monoid, and the `gen` and `gens` functions to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional `names` argument to the `FreeAbelianMonoid` function.

EXAMPLE 1: It is possible to create an abelian monoid in zero or more variables; the syntax T(1) creates the monoid indentity element even in the rank zero case.

```
sage: T = FreeAbelianMonoid(0, '')
sage: T
Free abelian monoid on 0 generators ()
sage: T.gens()
()
sage: T(1)
1
```

EXAMPLE 2: A free abelian monoid uses a multiplicative representation of elements, but the underlying representation is lists of integer exponents.

```
sage: F = FreeAbelianMonoid(5,names='a,b,c,d,e')
sage: (a,b,c,d,e) = F.gens()
sage: a*b^2*e*d
a*b^2*d*e
sage: x = b^2*e*d*a^7
sage: x
a^7*b^2*d*e
sage: x.list()
[7, 2, 0, 1, 1]
```

class FreeAbelianMonoidFactory()

Create the free abelian monoid in n generators.

INPUT:

- `n` - integer
- `names` - names of generators

OUTPUT: free abelian monoid

EXAMPLES:

```
sage: FreeAbelianMonoid(0, '')
Free abelian monoid on 0 generators ()
sage: F = FreeAbelianMonoid(5,names = list("abcde"))
sage: F
Free abelian monoid on 5 generators (a, b, c, d, e)
sage: F(1)
1
```

```
sage: (a, b, c, d, e) = F.gens()
sage: mul([ a, b, a, c, b, d, c, d ], F(1))
a^2*b^2*c^2*d^2
sage: a**2 * b**3 * a**2 * b**4
a^4*b^7

sage: loads(dumps(F)) is F
True
```

create_key(*n, names*)

create_object(*version, key*)

class FreeAbelianMonoid_class(*n, names*)

Free abelian monoid on n generators.

gen(*i=0*)

The i-th generator of the abelian monoid.

EXAMPLES:

```
sage: F = FreeAbelianMonoid(5,'a')
sage: F.gen(0)
a0
sage: F.gen(2)
a2
```

ngens()

The number of free generators of the abelian monoid.

EXAMPLES:

```
sage: F = FreeAbelianMonoid(3000, 'a')
sage: F.ngens()
3000
```

is_FreeAbelianMonoid(*x*)

Return True if x is a free abelian monoid.

EXAMPLES:

```
sage: from sage.monoids.free_abelian_monoid import is_FreeAbelianMonoid
sage: is_FreeAbelianMonoid(5)
False
sage: is_FreeAbelianMonoid(FreeAbelianMonoid(7,'a'))
True
sage: is_FreeAbelianMonoid(FreeMonoid(7,'a'))
False
sage: is_FreeAbelianMonoid(FreeMonoid(0,''))
False
```

21.4 Abelian monoid elements

AUTHORS:

- David Kohel (2005-09)

EXAMPLES: Recall the example from abelian monoids.

```
sage: F = FreeAbelianMonoid(5,names = list("abcde"))
sage: (a,b,c,d,e) = F.gens()
sage: a*b^2*e*d
a*b^2*d*e
sage: x = b^2*e*d*a^7
sage: x
a^7*b^2*d*e
sage: x.list()
[7, 2, 0, 1, 1]
```

It is important to note that lists are mutable and the returned list is not a copy. As a result, reassignment of an element of the list changes the object.

```
sage: x.list()[0] = 0
sage: x
b^2*d*e
```

class FreeAbelianMonoidElement (*F*, *x*)

list ()

Return (a reference to) the underlying list used to represent this element. If this is a monoid in an abelian monoid on n generators, then this is a list of nonnegative integers of length n.

EXAMPLES:

```
sage: F = FreeAbelianMonoid(5, 'abcde')
sage: (a, b, c, d, e) = F.gens()
sage: a.list()
[1, 0, 0, 0, 0]
```

is_FreeAbelianMonoidElement (*x*)

CHAPTER

TWENTYTWO

GROUPS

22.1 Base class for groups

class AbelianGroup()
Generic abelian group.

is_abelian()
Return True.

class AlgebraicGroup()
Generic algebraic group.

class FiniteGroup()
Generic finite group.

cayley_graph()
Returns the cayley graph for this finite group, as a Sage DiGraph object. To plot the graph with with different colors

EXAMPLES:

```
sage: D4 = DihedralGroup(4); D4
Dihedral group of order 8 as a permutation group
sage: G = D4.cayley_graph()
sage: show(G, color_by_label=True, edge_labels=True)
sage: A5 = AlternatingGroup(5); A5
Alternating group of order 5!/2 as a permutation group
sage: G = A5.cayley_graph()
sage: G.show3d(color_by_label=True, edge_size=0.01, edge_size2=0.02, vertex_size=0.03)
sage: G.show3d(vertex_size=0.03, edge_size=0.01, edge_size2=0.02, vertex_colors={(1,1,1):G.v

sage: s1 = SymmetricGroup(1); s = s1.cayley_graph(); s.vertices()
[()]
```

AUTHORS:

- Bobby Moretti (2007-08-10)
- Robert Miller (2008-05-01): editing

is_finite()
Return True.

class Group()
Generic group class

category()
The category of all groups

is_abelian()
Return True if this group is abelian.

is_atomic_repr()
True if the elements of this group have atomic string representations. For example, integers are atomic but polynomials are not.

is_finite()
Returns True if this group is finite.

is_multiplicative()
Returns True if the group operation is given by * (rather than +).

Override for additive groups.

order()
Returns the number of elements of this group, which is either a positive integer or infinity.

quotient()
Return the quotient of this group by the normal subgroup H.

random_element()
Return a random element of this group.

22.2 Multiplicative Abelian Groups

AUTHORS:

- William Stein, David Joyner (2008-12): added (user requested) is_cyclic, fixed elementary_divisors.
- David Joyner (2006-03): (based on free abelian monoids by David Kohel)
- David Joyner (2006-05) several significant bug fixes
- David Joyner (2006-08) trivial changes to docs, added random, fixed bug in how invariants are recorded
- David Joyner (2006-10) added dual_group method
- David Joyner (2008-02) fixed serious bug in word_problem
- David Joyner (2008-03) fixed bug in trivial group case
- David Loeffler (2009-05) added subgroups method

TODO:

- additive abelian groups should also be supported

Background on invariant factors and the Smith normal form (according to section 4.1 of [C1]): An abelian group is a group A for which there exists an exact sequence $\mathbf{Z}^k \to \mathbf{Z}^\ell \to A \to 1$, for some positive integers k, ℓ with $k \leq \ell$. For example, a finite abelian group has a decomposition

$$A = \langle a_1 \rangle \times \cdots \times \langle a_\ell \rangle,$$

where $ord(a_i) = p_i^{c_i}$, for some primes p_i and some positive integers c_i, $i = 1, ..., \ell$. GAP calls the list (ordered by size) of the $p_i^{c_i}$ the *abelian invariants*. In Sage they will be called *invariants*. In this situation, $k = \ell$ and $\phi : \mathbf{Z}^\ell \to A$ is the map $\phi(x_1, ..., x_\ell) = a_1^{x_1} ... a_\ell^{x_\ell}$, for $(x_1, ..., x_\ell) \in \mathbf{Z}^\ell$. The matrix of relations $M : \mathbf{Z}^k \to \mathbf{Z}^\ell$ is the matrix whose rows generate the kernel of ϕ as a $\mathbf{Z}$-module. In other words, $M = (M_{ij})$ is a $\ell \times \ell$ diagonal matrix with $M_{ii} = p_i^{c_i}$.

Consider now the subgroup $B \subset A$ generated by $b_1 = a_1^{f_{1,1}}...a_\ell^{f_{\ell,1}}$, ..., $b_m = a_1^{f_{1,m}}...a_\ell^{f_{\ell,m}}$. The kernel of the map $\phi_B : \mathbf{Z}^m \to B$ defined by $\phi_B(y_1, ..., y_m) = b_1^{y_1}...b_m^{y_m}$, for $(y_1, ..., y_m) \in \mathbf{Z}^m$, is the kernel of the matrix

$$F = \begin{pmatrix} f_{11} & f_{12} & \cdots & f_{1m} \\ f_{21} & f_{22} & \cdots & f_{2m} \\ \vdots & & \ddots & \vdots \\ f_{\ell,1} & f_{\ell,2} & \cdots & f_{\ell,m} \end{pmatrix},$$

regarded as a map $\mathbf{Z}^m \to (\mathbf{Z}/p_1^{c_1}\mathbf{Z}) \times ... \times (\mathbf{Z}/p_\ell^{c_\ell}\mathbf{Z})$. In particular, $B \cong \mathbf{Z}^m/ker(F)$. If $B = A$ then the Smith normal form (SNF) of a generator matrix of $ker(F)$ and the SNF of M are the same. The diagonal entries s_i of the SNF $S = diag[s_1, s_2, s_3, ...s_r, 0, 0, ...0]$, are called *determinantal divisors* of F. where r is the rank. The {it invariant factors} of A are:

$$s_1, s_2/s_1, s_3/s_2, ...s_r/s_{r-1}.$$

Sage supports multiplicative abelian groups on any prescribed finite number $n \geq 0$ of generators. Use the `AbelianGroup` function to create an abelian group, and the `gen` and `gens` functions to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional `names` argument to the `AbelianGroup` function.

EXAMPLE 1:

We create an abelian group in zero or more variables; the syntax T(1) creates the identity element even in the rank zero case.

```
sage: T = AbelianGroup(0,[])
sage: T
Trivial Abelian Group
sage: T.gens()
()
sage: T(1)
1
```

EXAMPLE 2: An abelian group uses a multiplicative representation of elements, but the underlying representation is lists of integer exponents.

```
sage: F = AbelianGroup(5,[3,4,5,5,7],names = list("abcde"))
sage: F
Multiplicative Abelian Group isomorphic to C3 x C4 x C5 x C5 x C7
sage: (a,b,c,d,e) = F.gens()
sage: a*b^2*e*d
a*b^2*d*e
sage: x = b^2*e*d*a^7
sage: x
a*b^2*d*e
sage: x.list()
[1, 2, 0, 1, 1]
```

REFERENCES:

- [C1] H. Cohen Advanced topics in computational number theory, Springer, 2000.
- [C2] —-, A course in computational algebraic number theory, Springer, 1996.
- [R] J. Rotman, An introduction to the theory of groups, 4th ed, Springer, 1995.

Warning: Many basic properties for infinite abelian groups are not implemented.

AbelianGroup(*n, invfac=None, names='f'*)

Create the multiplicative abelian group in n generators with given invariants (which need not be prime powers).

INPUT:

- `n` - integer
- `invfac` - (the"invariant factors") a list of non-negative integers in the form [a0, a1,...,a(n-1)], typically written in increasing order. This list is padded with zeros if it has length less than n.
- `names` - (optional) names of generators

  Alternatively, you can also give input in the following form:

  `AbelianGroup(invfac, names="f"),`

  where names must be explicitly named.

OUTPUT: Abelian group with generators and invariant type. The default name for generator A.i is fi, as in GAP.

EXAMPLES:

```
sage: F = AbelianGroup(5, [5,5,7,8,9], names='abcde')
sage: F(1)
1
sage: (a, b, c, d, e) = F.gens()
sage: mul([ a, b, a, c, b, d, c, d ], F(1))
a^2*b^2*c^2*d^2
sage: d * b**2 * c**3
b^2*c^3*d
sage: F = AbelianGroup(3,[2]*3); F
Multiplicative Abelian Group isomorphic to C2 x C2 x C2
sage: H = AbelianGroup([2,3], names="xy"); H
Multiplicative Abelian Group isomorphic to C2 x C3
sage: AbelianGroup(5)
Multiplicative Abelian Group isomorphic to Z x Z x Z x Z x Z
sage: AbelianGroup(5).order()
+Infinity
```

Notice how 0's are padded on.

```
sage: AbelianGroup(5, [2,3,4])
Multiplicative Abelian Group isomorphic to Z x Z x C2 x C3 x C4
```

The invariant list can't be longer than the number of generators.

```
sage: AbelianGroup(2, [2,3,4])
...
ValueError: invfac (=[2, 3, 4]) must have length n (=2)
```

class **AbelianGroup_class**(*n, invfac, names='f'*)

Abelian group on n generators. The invariant factors $[a_1, a_2, ..., a_k]$ need not be prime powers. divisors will be).

EXAMPLES:

```
sage: F = AbelianGroup(5,[5,5,7,8,9],names = list("abcde")); F
Multiplicative Abelian Group isomorphic to C5 x C5 x C7 x C8 x C9
sage: F = AbelianGroup(5,[2, 4, 12, 24, 120],names = list("abcde")); F
Multiplicative Abelian Group isomorphic to C2 x C4 x C12 x C24 x C120
sage: F.elementary_divisors()
[2, 4, 12, 24, 120]
```

The entry 1 in the list of invariants is ignored:

```
sage: F = AbelianGroup(3,[1,2,3],names='a')
sage: F.invariants()
[2, 3]
sage: F.gens()
(a0, a1)
sage: F.ngens()
2
sage: (F.0).order()
2
sage: (F.1).order()
3
sage: AbelianGroup(1, [1], names='e')
Multiplicative Abelian Group isomorphic to C1
sage: AbelianGroup(1, [1], names='e').gens()
(e,)
sage: AbelianGroup(1, [1], names='e').list()
[1]
sage: AbelianGroup(3, [2, 1, 2], names=list('abc')).list()
[1, b, a, a*b]
```

dual_group()

Returns the dual group.

EXAMPLES:

elementary_divisors()

This returns the elementary divisors of the group, using Pari.

Note: Here is another algorithm for computing the elementary divisors $d_1, d_2, d_3, \ldots$, of a finite abelian group (where $d_1|d_2|d_3|\ldots$ are composed of prime powers dividing the invariants of the group in a way described below). Just factor the invariants a_i that define the abelian group. Then the biggest d_i is the product of the maximum prime powers dividing some a_j. In other words, the largest d_i is the product of p^v, where $v = max(ord_p(a_j)\text{forall}j)$. Now divide out all those p^v's into the list of invariants a_i, and get a new list of "smaller invariants"". Repeat the above procedure on these ""smaller invariants"" to compute d_{i-1}, and so on. (Thanks to Robert Miller for communicating this algorithm.)

TODO: When somebody wants to speed this code up, please implement he above algorithm.

EXAMPLES:

```
sage: G = AbelianGroup(2,[2,3])
sage: G.elementary_divisors()
[6]
sage: G = AbelianGroup(1, [6])
sage: G.elementary_divisors()
[6]
sage: G = AbelianGroup(2,[2,6])
sage: G
Multiplicative Abelian Group isomorphic to C2 x C6
sage: G.invariants()
[2, 6]
sage: G.elementary_divisors()
[2, 6]
sage: J = AbelianGroup([1,3,5,12])
sage: J.elementary_divisors()
[3, 60]
sage: G = AbelianGroup(2,[0,6])
sage: G.elementary_divisors()
[6, 0]
```

exponent()

Return the exponent of this abelian group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,7]); G
Multiplicative Abelian Group isomorphic to C2 x C3 x C7
sage: G.exponent()
42
```

gen(*i=0*)

The i-th generator of the abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(5,[],names='a')
sage: F.0
a0
sage: F.2
a2
sage: F.invariants()
[0, 0, 0, 0, 0]
```

invariants()

Return a copy of the list of invariants of this group.

It is safe to modify the returned list.

EXAMPLES:

```
sage: J = AbelianGroup([2,3])
sage: J.invariants()
[2, 3]
sage: v = J.invariants(); v
[2, 3]
sage: v[0] = 5
sage: J.invariants()
[2, 3]
sage: J.invariants() is J.invariants()
False
```

is_commutative()

Return True since this group is commutative.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9, 0])
sage: G.is_commutative()
True
sage: G.is_abelian()
True
```

is_cyclic()

Return True if the group is a cyclic group.

EXAMPLES: sage: J = AbelianGroup([2,3]) sage: J.invariants() [2, 3] sage: J.elementary_divisors() [6] sage: J.is_cyclic() True sage: G = AbelianGroup([6]) sage: G.invariants() [6] sage: G.is_cyclic() True sage: H = AbelianGroup([2,2]) sage: H.invariants() [2, 2] sage: H.is_cyclic() False sage: H = AbelianGroup([2,4]) sage: H.elementary_divisors() [2, 4] sage: H.is_cyclic() False sage: H.permutation_group().is_cyclic() False sage: T = AbelianGroup([]) sage: T.is_cyclic() True sage: T = AbelianGroup(1,[0]); T Multiplicative Abelian Group isomorphic to Z sage: T.is_cyclic() True

list()

Return list of all elements of this group.

EXAMPLES:

```
sage: G = AbelianGroup([2,3], names = "ab")
sage: G.list()
[1, b, b^2, a, a*b, a*b^2]

sage: G = AbelianGroup([]); G
Trivial Abelian Group
sage: G.list()
[1]
```

ngens()

The number of free generators of the abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(10000)
sage: F.ngens()
10000
```

order()

Return the order of this group.

EXAMPLES:

```
sage: G = AbelianGroup(2,[2,3])
sage: G.order()
6
sage: G = AbelianGroup(3,[2,3,0])
sage: G.order()
+Infinity
```

permutation_group()

Return the permutation group isomorphic to this abelian group.

If the invariants are $q_1, \ldots, q_n$ then the generators of the permutation will be of order $q_1, \ldots, q_n$, respectively.

EXAMPLES:

```
sage: G = AbelianGroup(2,[2,3]); G
Multiplicative Abelian Group isomorphic to C2 x C3
sage: G.permutation_group()
Permutation Group with generators [(1,4)(2,5)(3,6), (1,2,3)(4,5,6)]
```

random_element()

Return a random element of this group. (Renamed random to random_element.)

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: G.random_element()
f0*f1^2*f2
```

subgroup(*gensH*, *names='f'*)

Create a subgroup of this group. The "big" group must be defined using "named" generators.

INPUT:

- `gensH` - list of elements which are products of the generators of the ambient abelian group G = self

EXAMPLES:

```
sage: G.<a,b,c> = AbelianGroup(3, [2,3,4]); G
Multiplicative Abelian Group isomorphic to C2 x C3 x C4
sage: H = G.subgroup([a*b,a]); H
Multiplicative Abelian Group isomorphic to C2 x C3, which is the subgroup of
Multiplicative Abelian Group isomorphic to C2 x C3 x C4
```

```
generated by [a*b, a]
sage: H < G
True
sage: F = G.subgroup([a,b^2])
sage: F
Multiplicative Abelian Group isomorphic to C2 x C3, which is the subgroup of
Multiplicative Abelian Group isomorphic to C2 x C3 x C4
generated by [a, b^2]
sage: F.gens()
[a, b^2]
sage: F = AbelianGroup(5,[30,64,729],names = list("abcde"))
sage: a,b,c,d,e = F.gens()
sage: F.subgroup([a,b])
Multiplicative Abelian Group isomorphic to Z x Z, which is
the subgroup of Multiplicative Abelian Group isomorphic to
Z x Z x C30 x C64 x C729 generated by [a, b]
sage: F.subgroup([c,e])
Multiplicative Abelian Group isomorphic to C2 x C3 x C5 x
C729, which is the subgroup of Multiplicative Abelian
Group isomorphic to Z x Z x C30 x C64 x C729 generated by
[c, e]
```

subgroup_reduced(*elts, verbose=False*)

Given a list of lists of integers (corresponding to elements of self), find a set of independent generators for the subgroup generated by these elements, and return the subgroup with these as generators, forgetting the original generators.

This is used by the `subgroups` routine.

An error will be raised if the elements given are not linearly independent over QQ.

EXAMPLE:

```
sage: G = AbelianGroup([4,4])
sage: G.subgroup( [ G([1,0]), G([1,2]) ])
Multiplicative Abelian Group isomorphic to C2 x C4, which is the subgroup of
Multiplicative Abelian Group isomorphic to C4 x C4
generated by [f0, f0*f1^2]
sage: AbelianGroup([4,4]).subgroup_reduced( [ [1,0], [1,2] ])
Multiplicative Abelian Group isomorphic to C2 x C4, which is the subgroup of
Multiplicative Abelian Group isomorphic to C4 x C4
generated by [f1^2, f0]
```

subgroups(*check=False*)

Compute all the subgroups of this abelian group (which must be finite).

TODO: This is *many orders of magnitude* slower than Magma.

INPUT:

- check: if True, performs the same computation in GAP and checks that the number of subgroups generated is the same. (I don't know how to convert GAP's output back into Sage, so we don't actually compare the subgroups).

ALGORITHM:

If the group is cyclic, the problem is easy. Otherwise, write it as a direct product A x B, where B is cyclic. Compute the subgroups of A (by recursion).
Now, for every subgroup C of A x B, let G be its *projection onto* A and H its *intersection with* B. Then there is a well-defined homomorphism f: G -> B/H that sends a in G to the class mod H of b, where (a,b) is any element of C lifting a; and every subgroup C arises from a unique triple (G, H, f).

EXAMPLES:

```
sage: AbelianGroup([2,3]).subgroups()
[Multiplicative Abelian Group isomorphic to C2 x C3, which is the subgroup of
Multiplicative Abelian Group isomorphic to C2 x C3
generated by [f0*f1^2],
 Multiplicative Abelian Group isomorphic to C2, which is the subgroup of
Multiplicative Abelian Group isomorphic to C2 x C3
generated by [f0],
 Multiplicative Abelian Group isomorphic to C3, which is the subgroup of
Multiplicative Abelian Group isomorphic to C2 x C3
generated by [f1],
 Trivial Abelian Group, which is the subgroup of
Multiplicative Abelian Group isomorphic to C2 x C3
generated by []]

sage: len(AbelianGroup([2,4,8]).subgroups())
81
```

class AbelianGroup_subgroup(*ambient, gens, names='f'*)

Subgroup subclass of AbelianGroup_class, so instance methods are inherited.

TODO:

- There should be a way to coerce an element of a subgroup into the ambient group.

ambient_group()

Return the ambient group related to self.

gen(*n*)

Return the nth generator of this subgroup.

EXAMPLE:

```
sage: G.<a,b> = AbelianGroup(2)
sage: A = G.subgroup([a])
sage: A.gen(0)
a
```

gens()

Return the generators for this subgroup.

invs()

Return the invariants for this subgroup.

is_AbelianGroup(*x*)

Return True if x is an abelian group.

EXAMPLES:

```
sage: from sage.groups.abelian_gps.abelian_group import is_AbelianGroup
sage: F = AbelianGroup(5,[5,5,7,8,9],names = list("abcde")); F
Multiplicative Abelian Group isomorphic to C5 x C5 x C7 x C8 x C9
sage: is_AbelianGroup(F)
True
sage: is_AbelianGroup(AbelianGroup(7, [3]*7))
True
```

word_problem(*words, g, verbose=False*)

G and H are abelian, g in G, H is a subgroup of G generated by a list (words) of elements of G. If g is in H, return the expression for g as a word in the elements of (words).

The 'word problem' for a finite abelian group G boils down to the following matrix-vector analog of the Chinese remainder theorem.

Problem: Fix integers $1 < n_1 \leq n_2 \leq ... \leq n_k$ (indeed, these n_i will all be prime powers), fix a generating set $g_i = (a_{i1}, ..., a_{ik})$ (with $a_{ij} \in \mathrm{Z}/n_j\mathrm{Z}$), for $1 \leq i \leq \ell$, for the group G, and let $d = (d_1, ..., d_k)$ be an element of the direct product $\mathrm{Z}/n_1\mathrm{Z} \times ... \times \mathrm{Z}/n_k\mathrm{Z}$. Find, if they exist, integers $c_1, ..., c_\ell$ such that $c_1 g_1 + ... + c_\ell g_\ell = d$. In other words, solve the equation $cA = d$ for $c \in \mathrm{Z}^\ell$, where A is the matrix whose rows are the g_i's. Of course, it suffices to restrict the c_i's to the range $0 \leq c_i \leq N - 1$, where N denotes the least common multiple of the integers $n_1, ..., n_k$.

This function does not solve this directly, as perhaps it should. Rather (for both speed and as a model for a similar function valid for more general groups), it pushes it over to GAP, which has optimized (non-deterministic) algorithms for the word problem. Essentially, this function is a wrapper for the GAP function 'Factorization'.

EXAMPLE:

```
sage: G.<a,b,c> = AbelianGroup(3,[2,3,4]); G
Multiplicative Abelian Group isomorphic to C2 x C3 x C4
sage: word_problem([a*b,a*c], b*c)
[[a*b, 1], [a*c, 1]]
sage: word_problem([a*c,c],a)
[[a*c, 1], [c, -1]]
sage: word_problem([a*c,c],a,verbose=True)
a = (a*c)^1*(c)^-1
[[a*c, 1], [c, -1]]


sage: A.<a,b,c,d,e> = AbelianGroup(5,[4, 5, 5, 7, 8])
sage: b1 = a^3*b*c*d^2*e^5
sage: b2 = a^2*b*c^2*d^3*e^3
sage: b3 = a^7*b^3*c^5*d^4*e^4
sage: b4 = a^3*b^2*c^2*d^3*e^5
sage: b5 = a^2*b^4*c^2*d^4*e^5
sage: word_problem([b1,b2,b3,b4,b5],e)
[[a^3*b*c*d^2*e^5, 1], [a^2*b*c^2*d^3*e^3, 1], [a^3*b^3*d^4*e^4, 3], [a^2*b^4*c^2*d^4*e^5, 1]]
sage: word_problem([a,b,c,d,e],e)
[[e, 1]]
sage: word_problem([a,b,c,d,e],b)
[[b, 1]]
```

Warning:

1. Might have unpleasant effect when the word problem cannot be solved.

2. Uses permutation groups, so may be slow when group is large. The instance method word_problem of the class AbelianGroupElement is implemented differently (wrapping GAP's 'EpimorphismFromFreeGroup' and 'PreImagesRepresentative') and may be faster.

22.3 Abelian group elements

AUTHORS:

- David Joyner (2006-02); based on free_abelian_monoid_element.py, written by David Kohel.
- David Joyner (2006-05); bug fix in order
- David Joyner (2006-08); bug fix+new method in pow for negatives+fixed corresponding examples.
- David Joyner (2009-02): Fixed bug in order.

EXAMPLES: Recall an example from abelian groups.

```
sage: F = AbelianGroup(5,[4,5,5,7,8],names = list("abcde"))
sage: (a,b,c,d,e) = F.gens()
sage: x = a*b^2*e*d^20*e^12
sage: x
a*b^2*d^6*e^5
sage: x = a^10*b^12*c^13*d^20*e^12
sage: x
a^2*b^2*c^3*d^6*e^4
sage: y = a^13*b^19*c^23*d^27*e^72
sage: y
a*b^4*c^3*d^6
sage: x*y
a^3*b*c*d^5*e^4
sage: x.list()
[2, 2, 3, 6, 4]
```

It is important to note that lists are mutable and the returned list is not a copy. As a result, reassignment of an element of the list changes the object.

```
sage: x.list()[0] = 3
sage: x.list()
[3, 2, 3, 6, 4]
sage: x
a^3*b^2*c^3*d^6*e^4
```

class AbelianGroupElement(*F*, *x*)

as_permutation()

Return the element of the permutation group G (isomorphic to the abelian group A) associated to a in A.

EXAMPLES:

```
sage: G = AbelianGroup(3,[2,3,4],names="abc"); G
Multiplicative Abelian Group isomorphic to C2 x C3 x C4
sage: a,b,c=G.gens()
sage: Gp = G.permutation_group(); Gp
Permutation Group with generators [(1,13)(2,14)(3,15)(4,16)(5,17)(6,18)(7,19)(8,20)(9,21)(10
sage: a.as_permutation()
(1,13)(2,14)(3,15)(4,16)(5,17)(6,18)(7,19)(8,20)(9,21)(10,22)(11,23)(12,24)
sage: ap = a.as_permutation(); ap
(1,13)(2,14)(3,15)(4,16)(5,17)(6,18)(7,19)(8,20)(9,21)(10,22)(11,23)(12,24)
sage: ap in Gp
True
```

inverse()

Returns the inverse element.

EXAMPLE:

```
sage: G.<a,b> = AbelianGroup(2)
sage: a^-1
a^-1
sage: a.list()
[1, 0]
sage: a.inverse().list()
[-1, 0]
sage: a.inverse()
a^-1
```

list()

Return (a reference to) the underlying list used to represent this element. If this is a word in an abelian group on n generators, then this is a list of nonnegative integers of length n.

EXAMPLES:

```
sage: F = AbelianGroup(5, [3,4,5,8,7], 'abcde')
sage: (a, b, c, d, e) = F.gens()
sage: a.list()
[1, 0, 0, 0, 0]
```

order()

Returns the (finite) order of this element or Infinity if this element does not have finite order.

EXAMPLES:

```
sage: F = AbelianGroup(3,[7,8,9]); F
Multiplicative Abelian Group isomorphic to C7 x C8 x C9
sage: F.gens()[2].order()
9
sage: a,b,c = F.gens()
sage: (b*c).order()
72
sage: G = AbelianGroup(3,[7,8,9])
sage: type((G.0 * G.1).order())==Integer
True
```

random_element()

Return a random element of this dual group.

word_problem(*words*)

TODO - this needs a rewrite - see stuff in the matrix_grp directory.

G and H are abelian groups, g in G, H is a subgroup of G generated by a list (words) of elements of G. If self is in H, return the expression for self as a word in the elements of (words).

This function does not solve the word problem in Sage. Rather it pushes it over to GAP, which has optimized (non-deterministic) algorithms for the word problem.

Warning: Don't use E (or other GAP-reserved letters) as a generator name.

EXAMPLE:

```
sage: G = AbelianGroup(2,[2,3], names="xy")
sage: x,y = G.gens()
sage: x.word_problem([x,y])
[[x, 1]]
sage: y.word_problem([x,y])
[[y, 1]]
sage: (y*x).word_problem([x,y])
[[x, 1], [y, 1]]
```

is_AbelianGroupElement(*x*)

Return true if x is an abelian group element, i.e., an element of type AbelianGroupElement.

EXAMPLES: Though the integer 3 is in the integers, and the integers have an abelian group structure, 3 is not an AbelianGroupElement:

```
sage: from sage.groups.abelian_gps.abelian_group_element import is_AbelianGroupElement
sage: is_AbelianGroupElement(3)
False
sage: F = AbelianGroup(5, [3,4,5,8,7], 'abcde')
sage: is_AbelianGroupElement(F.0)
True
```

22.4 Homomorphisms of abelian groups

TODO:

- there must be a homspace first
- there should be hom and Hom methods in abelian group

AUTHORS:

- David Joyner (2006-03-03): initial version

class AbelianGroupMap (*parent*)
A set-theoretic map between AbelianGroups.

class AbelianGroupMorphism (*G, H, genss, imgss*)
Some python code for wrapping GAP's GroupHomomorphismByImages function for abelian groups. Returns "fail" if gens does not generate self or if the map does not extend to a group homomorphism, self - other.

EXAMPLES:

```
sage: G = AbelianGroup(3,[2,3,4],names="abc"); G
Multiplicative Abelian Group isomorphic to C2 x C3 x C4
sage: a,b,c = G.gens()
sage: H = AbelianGroup(2,[2,3],names="xy"); H
Multiplicative Abelian Group isomorphic to C2 x C3
sage: x,y = H.gens()


sage: from sage.groups.abelian_gps.abelian_group_morphism import AbelianGroupMorphism
sage: phi = AbelianGroupMorphism(H,G,[x,y],[a,b])
```

AUTHORS:

•David Joyner (2006-02)

codomain ()

domain ()

image (*J*)
Only works for finite groups.

J must be a subgroup of G. Computes the subgroup of H which is the image of J.

EXAMPLES:

```
sage: G = AbelianGroup(2,[2,3],names="xy")
sage: x,y = G.gens()
sage: H = AbelianGroup(3,[2,3,4],names="abc")
sage: a,b,c = H.gens()
sage: phi = AbelianGroupMorphism(G,H,[x,y],[a,b])
```

kernel ()
Only works for finite groups.

TODO: not done yet; returns a gap object but should return a Sage group.

EXAMPLES:

```
sage: H = AbelianGroup(3,[2,3,4],names="abc"); H
Multiplicative Abelian Group isomorphic to C2 x C3 x C4
sage: a,b,c = H.gens()
sage: G = AbelianGroup(2,[2,3],names="xy"); G
Multiplicative Abelian Group isomorphic to C2 x C3
sage: x,y = G.gens()
sage: phi = AbelianGroupMorphism(G,H,[x,y],[a,b])
sage: phi.kernel()
'Group([  ])'
```

range()

class **AbelianGroupMorphism_id**(*X*)

Return the identity homomorphism from X to itself.

EXAMPLES:

is_AbelianGroupMorphism(*f*)

22.5 Basic functionality for dual groups of finite multiplicative Abelian groups

AUTHORS:

- David Joyner (2006-08) (based on abelian_groups)
- David Joyner (2006-10) modifications suggested by William Stein

TODO:

- additive abelian groups should also be supported.

The basic idea is very simple. Let G be an abelian group and G^* its dual (i.e., the group of homomorphisms from G to $\mathbf{C}^\times$). Let g_j, $j = 1,..,n$, denote generators of G - say g_j is of order $m_j > 1$. There are generators X_j, $j = 1,..,n$, of G^* for which $X_j(g_j) = \exp(2\pi i/m_j)$ and $X_i(g_j) = 1$ if $i \neq j$. These are used to construct G^* in the DualAbelianGroup class below.

Sage supports multiplicative abelian groups on any prescribed finite number $n > 0$ of generators. Use the `AbelianGroup` function to create an abelian group, the `DualAbelianGroup` function to create its dual, and then the `gen` and `gens` functions to obtain the corresponding generators. You can print the generators as arbitrary strings using the optional `names` argument to the `DualAbelianGroup` function.

DualAbelianGroup(*G, names='X', base_ring=Complex Field with 53 bits of precision*)

Create the dual group of the multiplicative abelian group G.

INPUT:

- `G` - a finite abelian group
- `names` - (optional) names of generators

OUTPUT: The dual group of G.

EXAMPLES:

```
sage: F = AbelianGroup(5, [2,5,7,8,9], names='abcde')
sage: (a, b, c, d, e) = F.gens()
sage: Fd = DualAbelianGroup(F,names='ABCDE')
```

```
sage: A,B,C,D,E = Fd.gens()
sage: A(a)     ## random
-1.0000000000000000 + 0.00000000000000013834419720915037*I
sage: A(b); A(c); A(d); A(e)
1.00000000000000
1.00000000000000
1.00000000000000
1.00000000000000
```

class **DualAbelianGroup_class**(*G, names='X', bse_ring=None*)

Dual of abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(5,[3,5,7,8,9],names = list("abcde"))
sage: DualAbelianGroup(F)
Dual of Abelian Group isomorphic to Z/3Z x Z/5Z x Z/7Z x Z/8Z x Z/9Z  over Complex Field with 53
sage: F = AbelianGroup(4,[15,7,8,9],names = list("abcd"))
sage: DualAbelianGroup(F)
Dual of Abelian Group isomorphic to Z/15Z x Z/7Z x Z/8Z x Z/9Z  over Complex Field with 53 bits
```

base_ring()

gen(*i=0*)

The i-th generator of the abelian group.

EXAMPLES:

```
sage: F = AbelianGroup(3,[1,2,3],names='a')
sage: Fd = DualAbelianGroup(F, names="A")
sage: Fd.0
A0
sage: Fd.1
A1
sage: Fd.invariants()
[2, 3]
```

group()

invariants()

The invariants of the dual group.

EXAMPLES:

```
sage: F = AbelianGroup(1000)
sage: Fd = DualAbelianGroup(F)
sage: invs = Fd.invariants(); len(invs)
1000
```

This can be slow for 10000 or more generators.

is_commutative()

Return True since this group is commutative.

EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: Gd = DualAbelianGroup(G)
sage: Gd.is_commutative()
True
sage: Gd.is_abelian()
True
```

list()
Return list of all elements of this group.
EXAMPLES:

```
sage: G = AbelianGroup([2,3], names = "ab")
sage: Gd = DualAbelianGroup(G, names = "AB")
sage: Gd.list()
[1, B, B^2, A, A*B, A*B^2]
```

ngens()
The number of generators of the dual group.
EXAMPLES:

```
sage: F = AbelianGroup(1000)
sage: Fd = DualAbelianGroup(F)
sage: Fd.ngens()
1000
```

This can be slow for 10000 or more generators.

order()
Return the order of this group.
EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: Gd = DualAbelianGroup(G)
sage: Gd.order()
54
```

random()
Deprecated. Use self.random_element() instead.

random_element()
Return a random element of this dual group.
EXAMPLES:

```
sage: G = AbelianGroup([2,3,9])
sage: Gd = DualAbelianGroup(G)
sage: Gd.random_element()
X0*X1^2*X2
sage: N = 43^2-1
sage: G = AbelianGroup([N],names="a")
sage: Gd = DualAbelianGroup(G,names="A")
sage: a, = G.gens()
sage: A, = Gd.gens()
sage: x = a^(N/4); y = a^(N/3); z = a^(N/14)
sage: X = Gd.random_element(); X
A^615
sage: len([a for a in [x,y,z] if abs(X(a)-1)>10^(-8)])
2
```

is_DualAbelianGroup(x)
Return True if x is the dual group of an abelian group.
EXAMPLES:

```
sage: from sage.groups.abelian_gps.dual_abelian_group import is_DualAbelianGroup
sage: F = AbelianGroup(5,[3,5,7,8,9],names = list("abcde"))
sage: Fd = DualAbelianGroup(F)
sage: is_DualAbelianGroup(Fd)
True
```

```
sage: F = AbelianGroup(3,[1,2,3],names='a')
sage: Fd = DualAbelianGroup(F)
sage: Fd.gens()
(X0, X1)
sage: F.gens()
(a0, a1)
```

22.6 Permutation groups

A permutation group is a finite group G whose elements are permutations of a given finite set X (i.e., bijections $X \longrightarrow X$) and whose group operation is the composition of permutations. The number of elements of X is called the degree of G.

In Sage, a permutation is represented as either a string that defines a permutation using disjoint cycle notation, or a list of tuples, which represent disjoint cycles. That is:

```
(a,...,b)(c,...,d)...(e,...,f)  <--> [(a,...,b), (c,...,d),..., (e,...,f)]
                    () = identity <--> []
```

You can make the "named" permutation groups (see `permgp_named.py`) and use the following constructions:

- permutation group generated by elements,
- `direct_product_permgroups`, which takes a list of permutation groups and returns their direct product.

JOKE: Q: What's hot, chunky, and acts on a polygon? A: Dihedral soup. Renteln, P. and Dundes, A. "Foolproof: A Sampling of Mathematical Folk Humor." Notices Amer. Math. Soc. 52, 24-34, 2005.

AUTHORS:

- David Joyner (2005-10-14): first version
- David Joyner (2005-11-17)
- William Stein (2005-11-26): rewrite to better wrap Gap
- David Joyner (2005-12-21)
- William Stein and David Joyner (2006-01-04): added conjugacy_class_representatives
- David Joyner (2006-03): reorganization into subdirectory perm_gps; added __contains__, has_element; fixed _cmp_; added subgroup class+methods, PGL,PSL,PSp, PSU classes,
- David Joyner (2006-06): added PGU, functionality to SymmetricGroup, AlternatingGroup, direct_product_permgroups
- David Joyner (2006-08): added degree, ramification_module_decomposition_modular_curve and ramification_module_decomposition_hurwitz_curve methods to PSL(2,q), MathieuGroup, is_isomorphic
- Bobby Moretti (2006)-10): Added KleinFourGroup, fixed bug in DihedralGroup
- David Joyner (2006-10): added is_subgroup (fixing a bug found by Kiran Kedlaya), is_solvable, normalizer, is_normal_subgroup, Suzuki
- David Kohel (2007-02): fixed __contains__ to not enumerate group elements, following the convention for __call__

- David Harvey, Mike Hansen, Nick Alexander, William Stein (2007-02,03,04,05): Various patches
- Nathan Dunfield (2007-05): added orbits
- David Joyner (2007-06): added subgroup method (suggested by David Kohel), composition_series, lower_central_series, upper_central_series, cayley_table, quotient_group, sylow_subgroup, is_cyclic, homology, homology_part, cohomology, cohomology_part, poincare_series, molien_series, is_simple, is_monomial, is_supersolvable, is_nilpotent, is_perfect, is_polycyclic, is_elementary_abelian, is_pgroup, gens_small, isomorphism_type_info_simple_group. moved all the"named" groups to a new file.
- Nick Alexander (2007-07): move is_isomorphic to isomorphism_to, add from_gap_list
- William Stein (2007-07): put is_isomorphic back (and make it better)
- David Joyner (2007-08): fixed bugs in composition_series, upper/lower_central_series, derived_series,
- David Joyner (2008-06): modified is_normal (reported by W. J. Palenstijn), and added normalizes
- David Joyner (2008-08): Added example to docstring of cohomology.
- Simon King (2009-04): __cmp__ methods for PermutationGroup_generic and PermutationGroup_subgroup

REFERENCES:

- Cameron, P., Permutation Groups. New York: Cambridge University Press, 1999.
- Wielandt, H., Finite Permutation Groups. New York: Academic Press, 1964.
- Dixon, J. and Mortimer, B., Permutation Groups, Springer-Verlag, Berlin/New York, 1996.

Note: Though Suzuki groups are okay, Ree groups should *not* be wrapped as permutation groups - the construction is too slow - unless (for small values or the parameter) they are made using explicit generators.

PermutationGroup(*gens=None, gap_group=None, canonicalize=True*)

Return the permutation group associated to x (typically a list of generators).

INPUT:

- `gens` - list of generators (default: `None`)
- `gap_group` - a gap permutation group (default: `None`)
- `canonicalize` - bool (default: `True`); if `True`, sort generators and remove duplicates

OUTPUT:

- A permutation group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
```

We can also make permutation groups from PARI groups:

```
sage: H = pari('x^4 - 2*x^3 - 2*x + 1').polgalois()
sage: G = PariGroup(H, 4); G
PARI group [8, -1, 3, "D(4)"] of degree 4
sage: H = PermutationGroup(G); H          # optional - database_gap
Transitive group number 3 of degree 4
sage: H.gens()                            # optional - database_gap
[(1,2,3,4), (1,3)]
```

We can also create permutation groups whose generators are Gap permutation objects:

```
sage: p = gap('(1,2)(3,7)(4,6)(5,8)'); p
(1,2)(3,7)(4,6)(5,8)
sage: PermutationGroup([p])
Permutation Group with generators [(1,2)(3,7)(4,6)(5,8)]
```

There is an underlying gap object that implements each permutation group:

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: G._gap_()
Group( [ (1,2,3,4) ] )
sage: gap(G)
Group( [ (1,2,3,4) ] )
sage: gap(G) is G._gap_()
True
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: current_randstate().set_seed_gap()
sage: G._gap_().DerivedSeries()
[ Group( [ (3,4), (1,2,3)(4,5) ] ), Group( [ (1,5)(3,4), (1,5)(2,4), (1,5,3) ] ) ]
```

TESTS:

```
sage: PermutationGroup(SymmetricGroup(5))
...
TypeError: gens must be a tuple, list, or GapElement
```

class **PermutationGroup_generic**(*gens=None, gap_group=None, canonicalize=True*)

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G
Permutation Group with generators [(3,4), (1,2,3)(4,5)]
sage: G.center()
Permutation Group with generators [()]
sage: G.group_id()          # optional - database_gap
[120, 34]
sage: n = G.order(); n
120
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: loads(G.dumps()) == G
True
```

cayley_table(*names='x'*)

Returns the multiplication table, or Cayley table, of the finite group G in the form of a matrix with symbolic coefficients. This function is useful for learning, teaching, and exploring elementary group theory. Of course, G must be a group of low order.

EXAMPLES:

As the last line below illustrates, the ordering used here in the first row is the same as in `G.list()`:

```
sage: G = PermutationGroup(['(1,2,3)', '(2,3)'])
sage: G.cayley_table()
[x0 x1 x2 x3 x4 x5]
[x1 x0 x3 x2 x5 x4]
[x2 x4 x0 x5 x1 x3]
[x3 x5 x1 x4 x0 x2]
[x4 x2 x5 x0 x3 x1]
[x5 x3 x4 x1 x2 x0]
```

```
sage: G.list()[3]*G.list()[3] == G.list()[4]
True
sage: G.cayley_table("y")
[y0 y1 y2 y3 y4 y5]
[y1 y0 y3 y2 y5 y4]
[y2 y4 y0 y5 y1 y3]
[y3 y5 y1 y4 y0 y2]
[y4 y2 y5 y0 y3 y1]
[y5 y3 y4 y1 y2 y0]
sage: G.cayley_table(names="abcdef")
[a b c d e f]
[b a d c f e]
[c e a f b d]
[d f b e a c]
[e c f a d b]
[f d e b c a]
```

center()

Return the subgroup of elements that commute with every element of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)]])
sage: G.center()
Permutation Group with generators [(1,2,3,4)]
sage: G = PermutationGroup([[(1,2,3,4)], [(1,2)]])
sage: G.center()
Permutation Group with generators [()]
```

centralizer(*g*)

Returns the centralizer of `g` in `self`.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: g = G([(1,3)])
sage: G.centralizer(g)
Permutation Group with generators [(2,4), (1,3)]
sage: g = G([(1,2,3,4)])
sage: G.centralizer(g)
Permutation Group with generators [(1,2,3,4)]
sage: H = G.subgroup([G([(1,2,3,4)])])
sage: G.centralizer(H)
Permutation Group with generators [(1,2,3,4)]
```

character(*values*)

Returns a group character from `values`, where `values` is a list of the values of the character evaluated on the conjugacy classes.

EXAMPLES:

```
sage: G = AlternatingGroup(4)
sage: n = len(G.conjugacy_classes_representatives())
sage: G.character([1]*n)
Character of Alternating group of order 4!/2 as a permutation group
```

character_table()

Returns the matrix of values of the irreducible characters of a permutation group G at the conjugacy classes of G. The columns represent the conjugacy classes of G and the rows represent the different irreducible characters in the ordering given by GAP.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3)]])
sage: G.order()
12
sage: G.character_table()
[          1           1           1           1]
[          1           1 -zeta3 - 1      zeta3]
[          1           1      zeta3 -zeta3 - 1]
[          3          -1           0           0]
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3)]])
sage: CT = gap(G).CharacterTable()
```

Type `print gap.eval("Display(%s)"%CT.name())` to display this nicely.

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: G.order()
8
sage: G.character_table()
[ 1  1  1  1  1]
[ 1 -1 -1  1  1]
[ 1 -1  1 -1  1]
[ 1  1 -1 -1  1]
[ 2  0  0  0 -2]
sage: CT = gap(G).CharacterTable()
```

Again, type `print gap.eval("Display(%s)"%CT.name())` to display this nicely.

```
sage: SymmetricGroup(2).character_table()
[ 1 -1]
[ 1  1]
sage: SymmetricGroup(3).character_table()
[ 1 -1  1]
[ 2  0 -1]
[ 1  1  1]
sage: SymmetricGroup(5).character_table()
[ 1 -1  1  1 -1 -1  1]
[ 4 -2  0  1  1  0 -1]
[ 5 -1  1 -1 -1  1  0]
[ 6  0 -2  0  0  0  1]
[ 5  1  1 -1  1 -1  0]
[ 4  2  0  1 -1  0 -1]
[ 1  1  1  1  1  1  1]
sage: list(AlternatingGroup(6).character_table())
[(1, 1, 1, 1, 1, 1, 1), (5, 1, 2, -1, -1, 0, 0), (5, 1, -1, 2, -1, 0, 0), (8, 0, -1, -1, 0,
```

Suppose that you have a class function $f(g)$ on G and you know the values $v_1, \ldots, v_n$ on the conjugacy class elements in `conjugacy_classes_representatives(G)` = $[g_1, \ldots, g_n]$. Since the irreducible characters $\rho_1, \ldots, \rho_n$ of G form an E-basis of the space of all class functions (E a "sufficiently large" cyclotomic field), such a class function is a linear combination of these basis elements, $f = c_1\rho_1 + \cdots + c_n\rho_n$. To find the coefficients c_i, you simply solve the linear system `character_table_values(G)` $[v_1, ..., v_n] = [c_1, ..., c_n]$, where $[v_1, \ldots, v_n]$ = `character_table_values(G)` $^{(-1)}[c_1, ..., c_n]$.

AUTHORS:

- David Joyner and William Stein (2006-01-04)

cohomology (*n, p=0*)

Computes the group cohomology $H^n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's `GroupHomology` function, written by Graham Ellis.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: G.cohomology(1,2)                                  # optional - gap_packages
Multiplicative Abelian Group isomorphic to C2
sage: G = SymmetricGroup(3)
sage: G.cohomology(5)                                    # optional - gap_packages
Trivial Abelian Group
sage: G.cohomology(5,2)                                  # optional - gap_packages
Multiplicative Abelian Group isomorphic to C2
sage: G.homology(5,3)                                    # optional - gap_packages
Trivial Abelian Group
sage: G.homology(5,4)                                    # optional - gap_packages
...
ValueError: p must be 0 or prime
```

This computes $H^4(S_3, \mathbf{Z})$ and $H^4(S_3, \mathbf{Z}/2\mathbf{Z})$, respectively.

AUTHORS:

- David Joyner and Graham Ellis

REFERENCES:

- G. Ellis, 'Computing group resolutions', J. Symbolic Computation. Vol.38, (2004)1077-1118 (Available at http://hamilton.nuigalway.ie/).
- D. Joyner, 'A primer on computational group homology and cohomology', http://front.math.ucdavis.edu/0706.0549.

cohomology_part(*n, p=0*)

Computes the p-part of the group cohomology $H^n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's Homology function, written by Graham Ellis, applied to the p-Sylow subgroup of G.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.cohomology_part(7,2)                      # optional - gap_packages
Multiplicative Abelian Group isomorphic to C2 x C2 x C2
sage: G = SymmetricGroup(3)
sage: G.cohomology_part(2,3)                      # optional - gap_packages
Multiplicative Abelian Group isomorphic to C3
```

AUTHORS:

- David Joyner and Graham Ellis

composition_series()

Return the composition series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing.

```
sage: set_random_seed(0)
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.composition_series()  # random output
[Permutation Group with generators [(1,2,3)(4,5), (3,4)], Permutation Group with generators
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: CS = G.composition_series()
sage: CS[3]
Permutation Group with generators [()]
```

conjugacy_classes_representatives()

Returns a complete list of representatives of conjugacy classes in a permutation group G. The ordering is that given by GAP.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: cl = G.conjugacy_classes_representatives(); cl
[(), (2,4), (1,2)(3,4), (1,2,3,4), (1,3)(2,4)]
sage: cl[3] in G
True

sage: G = SymmetricGroup(5)
sage: G.conjugacy_classes_representatives ()
[(), (1,2), (1,2)(3,4), (1,2,3), (1,2,3)(4,5), (1,2,3,4), (1,2,3,4,5)]
```

AUTHORS:

- David Joyner and William Stein (2006-01-04)

conjugacy_classes_subgroups()

Returns a complete list of representatives of conjugacy classes of subgroups in a permutation group G. The ordering is that given by GAP.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: cl = G.conjugacy_classes_subgroups()
sage: cl
[Permutation Group with generators [()],
 Permutation Group with generators [(1,2)(3,4)],
 Permutation Group with generators [(1,3)(2,4)],
 Permutation Group with generators [(2,4)],
 Permutation Group with generators [(1,4)(2,3), (1,2)(3,4)],
 Permutation Group with generators [(1,3)(2,4), (2,4)],
 Permutation Group with generators [(1,3)(2,4), (1,2,3,4)],
 Permutation Group with generators [(1,3)(2,4), (1,2)(3,4), (1,2,3,4)]]

sage: G = SymmetricGroup(3)
sage: G.conjugacy_classes_subgroups()
[Permutation Group with generators [()],
 Permutation Group with generators [(2,3)],
 Permutation Group with generators [(1,2,3)],
 Permutation Group with generators [(1,3,2), (1,2)]]
```

AUTHORS:

- David Joyner (2006-10)

degree()

Return the largest point moved by a permutation in this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: G.largest_moved_point()
4
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.largest_moved_point()
10

sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.degree()
5
```

derived_series()

Return the derived series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing.

```
sage: set_random_seed(0)
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.derived_series()  # random output
[Permutation Group with generators [(1,2,3)(4,5), (3,4)], Permutation Group with generators
```

direct_product (*other, maps=True*)

Wraps GAP's `DirectProduct`, `Embedding`, and `Projection`.

Sage calls GAP's `DirectProduct`, which chooses an efficient representation for the direct product. The direct product of permutation groups will be a permutation group again. For a direct product `D`, the GAP operation `Embedding(D,i)` returns the homomorphism embedding the i-th factor into `D`. The GAP operation `Projection(D,i)` gives the projection of `D` onto the i-th factor. This method returns a 5-tuple: a permutation group and 4 morphisms.

INPUT:

- `self, other` - permutation groups

OUTPUT:

- `D` - a direct product of the inputs, returned as a permutation group as well
- `iota1` - an embedding of `self` into `D`
- `iota2` - an embedding of `other` into `D`
- `pr1` - the projection of `D` onto `self` (giving a splitting 1 - other - D - self - 1)
- `pr2` - the projection of `D` onto `other` (giving a splitting 1 - self - D - other - 1)

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: D = G.direct_product(G,False)
sage: D
Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
sage: D,iota1,iota2,pr1,pr2 = G.direct_product(G)
sage: D; iota1; iota2; pr1; pr2
Permutation Group with generators [(1,2,3,4), (5,6,7,8)]
Homomorphism : Cyclic group of order 4 as a permutation group --> Permutation Group with gen
Homomorphism : Cyclic group of order 4 as a permutation group --> Permutation Group with gen
Homomorphism : Permutation Group with generators [(1,2,3,4), (5,6,7,8)] --> Cyclic group of
Homomorphism : Permutation Group with generators [(1,2,3,4), (5,6,7,8)] --> Cyclic group of

sage: g=D([(1,3),(2,4)]); g
(1,3)(2,4)
sage: d=D([(1,4,3,2),(5,7),(6,8)]); d
(1,4,3,2)(5,7)(6,8)
sage: iota1(g); iota2(g); pr1(d); pr2(d)
(1,3)(2,4)
(5,7)(6,8)
(1,4,3,2)
(1,3)(2,4)
```

exponent ()

Computes the exponent of the group. The exponent e of a group G is the LCM of the orders of its elements, that is, e is the smallest integer such that $g^e = 1$ for all $g \in G$.

EXAMPLES:

```
sage: G = AlternatingGroup(4)
sage: G.exponent()
6
```

gen (*i*)

Returns the i-th generator of `self`; that is, the i-th element of the list `self.gens()`.

EXAMPLES:

We explicitly construct the alternating group on four elements:

```
sage: A4 = PermutationGroup([[(1,2,3)],[(2,3,4)]]); A4
Permutation Group with generators [(2,3,4), (1,2,3)]
sage: A4.gens()
[(2,3,4), (1,2,3)]
sage: A4.gen(0)
(2,3,4)
sage: A4.gen(1)
(1,2,3)
sage: A4.gens()[0]; A4.gens()[1]
(2,3,4)
(1,2,3)
```

gens()

Return tuple of generators of this group. These need not be minimal, as they are the generators used in defining this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3)], [(1,2)]])
sage: G.gens()
[(1,2), (1,2,3)]
```

Note that the generators need not be minimal, though duplicates are removed:

```
sage: G = PermutationGroup([[(1,2)], [(1,3)], [(2,3)], [(1,2)]])
sage: G.gens()
[(2,3), (1,2), (1,3)]
```

We can use index notation to access the generators returned by `self.gens`:

```
sage: G = PermutationGroup([[(1,2,3,4), (5,6)], [(1,2)]])
sage: g = G.gens()
sage: g[0]
(1,2)
sage: g[1]
(1,2,3,4)(5,6)
```

TESTS:

We make sure that the trivial group gets handled correctly:

```
sage: SymmetricGroup(1).gens()
[()]
```

gens_small()

For this group, returns a generating set which has few elements. As neither irredundancy nor minimal length is proven, it is fast.

EXAMPLES:

```
sage: R = "(25,27,32,30)(26,29,31,28)( 3,38,43,19)( 5,36,45,21)( 8,33,48,24)" ## R = right
sage: U = "( 1, 3, 8, 6)( 2, 5, 7, 4)( 9,33,25,17)(10,34,26,18)(11,35,27,19)" ## U = top
sage: L = "( 9,11,16,14)(10,13,15,12)( 1,17,41,40)( 4,20,44,37)( 6,22,46,35)" ## L = left
sage: F = "(17,19,24,22)(18,21,23,20)( 6,25,43,16)( 7,28,42,13)( 8,30,41,11)" ## F = front
sage: B = "(33,35,40,38)(34,37,39,36)( 3, 9,46,32)( 2,12,47,29)( 1,14,48,27)" ## B = back or
sage: D = "(41,43,48,46)(42,45,47,44)(14,22,30,38)(15,23,31,39)(16,24,32,40)" ## D = down or
sage: G = PermutationGroup([R,L,U,F,B,D])
sage: len(G.gens_small())
2
```

group_id()

Return the ID code of this group, which is a list of two integers. Requires "optional" database_gap-4.4.x package.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.group_id()    # optional - database_gap
[12, 4]
```

has_element (*item*)

Returns boolean value of `item in self` - however *ignores* parentage.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)]); g
(1,2,3,4)
sage: G.has_element(g)
True
sage: h = H([(1,2),(3,4)]); h
(1,2)(3,4)
sage: G.has_element(h)
False
```

homology (*n, p=0*)

Computes the group homology $H_n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's `GroupHomology` function, written by Graham Ellis.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

AUTHORS:

- David Joyner and Graham Ellis

The example below computes $H_7(S_5, \mathbf{Z})$, $H_7(S_5, \mathbf{Z}/2\mathbf{Z})$, $H_7(S_5, \mathbf{Z}/3\mathbf{Z})$, and $H_7(S_5, \mathbf{Z}/5\mathbf{Z})$, respectively. To compute the 2-part of $H_7(S_5, \mathbf{Z})$, use the `homology_part` function.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.homology(7)                                  # optional - gap_packages
Multiplicative Abelian Group isomorphic to C2 x C2 x C4 x C3 x C5
sage: G.homology(7,2)                                # optional - gap_packages
Multiplicative Abelian Group isomorphic to C2 x C2 x C2 x C2 x C2
sage: G.homology(7,3)                                # optional - gap_packages
Multiplicative Abelian Group isomorphic to C3
sage: G.homology(7,5)                                # optional - gap_packages
Multiplicative Abelian Group isomorphic to C5
```

REFERENCES:

- G. Ellis, "Computing group resolutions", J. Symbolic Computation. Vol.38, (2004)1077-1118 (Available at http://hamilton.nuigalway.ie/.
- D. Joyner, "A primer on computational group homology and cohomology", http://front.math.ucdavis.edu/0706.0549

homology_part (*n, p=0*)

Computes the p-part of the group homology $H_n(G, F)$, where $F = \mathbf{Z}$ if $p = 0$ and $F = \mathbf{Z}/p\mathbf{Z}$ if $p > 0$ is a prime. Wraps HAP's `Homology` function, written by Graham Ellis, applied to the p-Sylow subgroup of G.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.homology_part(7,2)                              # optional - gap_packages
Multiplicative Abelian Group isomorphic to C2 x C2 x C2 x C2 x C4
```

AUTHORS:

- David Joyner and Graham Ellis

id()

(Same as `self.group_id()`.) Return the ID code of this group, which is a list of two integers. Requires "optional" database_gap-4.4.x package.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.group_id()     # optional - database_gap
[12, 4]
```

identity()

Return the identity element of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)]])
sage: e = G.identity()
sage: e
()
sage: g = G.gen(0)
sage: g*e
(1,2,3)(4,5)
sage: e*g
(1,2,3)(4,5)
```

irreducible_characters()

Returns a list of the irreducible characters of `self`.

EXAMPLES:

```
sage: irr = SymmetricGroup(3).irreducible_characters()
sage: [x.values() for x in irr]
[[1, -1, 1], [2, 0, -1], [1, 1, 1]]
```

is_abelian()

Return `True` if this group is abelian.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_abelian()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_abelian()
True
```

is_commutative()

Return `True` if this group is commutative.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_commutative()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_commutative()
True
```

is_cyclic()
Return True if this group is cyclic.
EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_cyclic()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_cyclic()
True
```

is_elementary_abelian()
Return True if this group is elementary abelian. An elementary abelian group is a finite abelian group, where every nontrivial element has order p, where p is a prime.
EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_elementary_abelian()
False
sage: G = PermutationGroup(['(1,2,3)','(4,5,6)'])
sage: G.is_elementary_abelian()
True
```

is_isomorphic(*right*)
Return True if the groups are isomorphic. If mode="verbose" then an isomorphism is printed.
INPUT:

- self - this group
- right - a permutation group

OUTPUT:

- boolean; True if self and right are isomorphic groups; False otherwise.

EXAMPLES:

```
sage: v = ['(1,2,3)(4,5)', '(1,2,3,4,5)']
sage: G = PermutationGroup(v)
sage: H = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_isomorphic(H)
False
sage: G.is_isomorphic(G)
True
sage: G.is_isomorphic(PermutationGroup(list(reversed(v))))
True
```

is_monomial()
Returns True if the group is monomial. A finite group is monomial if every irreducible complex character is induced from a linear character of a subgroup.
EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_monomial()
True
```

is_nilpotent()
Return True if this group is nilpotent.
EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_nilpotent()
```

```
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_nilpotent()
True
```

is_normal(*other*)

Return `True` if this group is a normal subgroup of `other`.

EXAMPLES:

```
sage: AlternatingGroup(4).is_normal(SymmetricGroup(4))
True
sage: H = PermutationGroup(['(1,2,3)(4,5)'])
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: H.is_normal(G)
False
```

is_perfect()

Return `True` if this group is perfect. A group is perfect if it equals its derived subgroup.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_perfect()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_perfect()
False
```

is_pgroup()

Returns `True` if this group is a p-group. A finite group is a p-group if its order is of the form p^n for a prime integer p and a nonnegative integer n.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3,4,5)'])
sage: G.is_pgroup()
True
```

is_polycyclic()

Return `True` if this group is polycyclic. A group is polycyclic if it has a subnormal series with cyclic factors. (For finite groups, this is the same as if the group is solvable - see `is_solvable`.)

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: G.is_polycyclic()
False
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_polycyclic()
True
```

is_simple()

Returns `True` if the group is simple. A group is simple if it has no proper normal subgroups.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_simple()
False
```

is_solvable()

Returns `True` if the group is solvable.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_solvable()
True
```

is_subgroup(*other*)

Returns `True` if `self` is a subgroup of `other`.

EXAMPLES:

```
sage: G = AlternatingGroup(5)
sage: H = SymmetricGroup(5)
sage: G.is_subgroup(H)
True
```

is_supersolvable()

Returns `True` if the group is supersolvable. A finite group is supersolvable if it has a normal series with cyclic factors.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.is_supersolvable()
True
```

is_transitive()

Return `True` if `self` is a transitive group, i.e., if the action of self on [1..n] is transitive.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.is_transitive()
True
sage: G = PermutationGroup(['(1,2)(3,4)(5,6)'])
sage: G.is_transitive()
False
```

Note that this differs from the definition in GAP, where `IsTransitive` returns whether the group is transitive on the set of points moved by the group.

```
sage: G = PermutationGroup([(2,3)])
sage: G.is_transitive()
False
sage: gap(G).IsTransitive()
true
```

isomorphism_to(*right*)

Return an isomorphism from `self` to `right` if the groups are isomorphic, otherwise `None`.

INPUT:

- `self` - this group
- `right` - a permutation group

OUTPUT:

- `None` or a morphism of permutation groups.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: H = PermutationGroup(['(1,2,3)(4,5)'])
sage: G.isomorphism_to(H) is None
True
sage: G = PermutationGroup([(1,2,3), (2,3)])
sage: H = PermutationGroup([(1,2,4), (1,4)])
```

```
sage: G.isomorphism_to(H)
Homomorphism : Permutation Group with generators [(2,3), (1,2,3)] --> Permutation Group with
```

isomorphism_type_info_simple_group()

If the group is simple, then this returns the name of the group.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(5)
sage: G.isomorphism_type_info_simple_group()
rec( series := "Z", parameter := 5, name := "Z(5)" )
```

largest_moved_point()

Return the largest point moved by a permutation in this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: G.largest_moved_point()
4
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.largest_moved_point()
10

sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.degree()
5
```

list()

Return list of all elements of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3,4)], [(1,2)]])
sage: G.list()
[(), (3,4), (2,3), (2,3,4), (2,4,3), (2,4), (1,2), (1,2)(3,4), (1,2,3), (1,2,3,4), (1,2,4,3)
```

lower_central_series()

Return the lower central series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing.

```
sage: set_random_seed(0)
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.lower_central_series()  # random output
[Permutation Group with generators [(1,2,3)(4,5), (3,4)], Permutation Group with generators
```

molien_series()

Returns the Molien series of a transitive permutation group. The function

$$M(x) = (1/|G|) \sum_{g \in G} \det(1 - x * g)^{-1}$$

is sometimes called the "Molien series" of G. GAP's `MolienSeries` is associated to a character of a group G. How are these related? A group G, given as a permutation group on n points, has a "natural" representation of dimension n, given by permutation matrices. The Molien series of G is the one associated to that permutation representation of G using the above formula. Character values then count fixed points of the corresponding permutations.

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.molien_series()
1/(-x^15 + x^14 + x^13 - x^10 - x^9 - x^8 + x^7 + x^6 + x^5 - x^2 - x + 1)
sage: G = SymmetricGroup(3)
sage: G.molien_series()
1/(-x^6 + x^5 + x^4 - x^2 - x + 1)
```

multiplication_table(*names='x'*)

Returns the multiplication table, or Cayley table, of the finite group G in the form of a matrix with symbolic coefficients. This function is useful for learning, teaching, and exploring elementary group theory. Of course, G must be a group of low order.

EXAMPLES:

As the last line below illustrates, the ordering used here in the first row is the same as in `G.list()`:

```
sage: G = PermutationGroup(['(1,2,3)', '(2,3)'])
sage: G.cayley_table()
[x0 x1 x2 x3 x4 x5]
[x1 x0 x3 x2 x5 x4]
[x2 x4 x0 x5 x1 x3]
[x3 x5 x1 x4 x0 x2]
[x4 x2 x5 x0 x3 x1]
[x5 x3 x4 x1 x2 x0]
sage: G.list()[3]*G.list()[3] == G.list()[4]
True
sage: G.cayley_table("y")
[y0 y1 y2 y3 y4 y5]
[y1 y0 y3 y2 y5 y4]
[y2 y4 y0 y5 y1 y3]
[y3 y5 y1 y4 y0 y2]
[y4 y2 y5 y0 y3 y1]
[y5 y3 y4 y1 y2 y0]
sage: G.cayley_table(names="abcdef")
[a b c d e f]
[b a d c f e]
[c e a f b d]
[d f b e a c]
[e c f a d b]
[f d e b c a]
```

normal_subgroups()

Return the normal subgroups of this group as a (sorted in increasing order) list of permutation groups.

The normal subgroups of $H = PSL(2,7) \times PSL(2,7)$ are 1, two copies of $PSL(2,7)$ and H itself, as the following example shows.

EXAMPLES:

```
sage: G = PSL(2,7)
sage: D = G.direct_product(G)
sage: H = D[0]
sage: NH = H.normal_subgroups()
sage: len(NH)
4
sage: NH[1].is_isomorphic(G)
True
sage: NH[2].is_isomorphic(G)
True
```

normalizer(*g*)

Returns the normalizer of `g` in `self`.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: g = G([(1,3)])
sage: G.normalizer(g)
Permutation Group with generators [(2,4), (1,3)]
sage: g = G([(1,2,3,4)])
sage: G.normalizer(g)
Permutation Group with generators [(2,4), (1,2,3,4), (1,3)(2,4)]
sage: H = G.subgroup([G([(1,2,3,4)])])
sage: G.normalizer(H)
Permutation Group with generators [(2,4), (1,2,3,4), (1,3)(2,4)]
```

normalizes(*other*)

Returns `True` if the group `other` is normalized by `self`. Wraps GAP's `IsNormal` function.

A group G normalizes a group U if and only if for every $g \in G$ and $u \in U$ the element u^g is a member of U. Note that U need not be a subgroup of G.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: H = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: H.normalizes(G)
False
sage: G = SymmetricGroup(3)
sage: H = PermutationGroup( [ (4,5,6) ] )
sage: G.normalizes(H)
True
sage: H.normalizes(G)
True
```

In the last example, G and H are disjoint, so each normalizes the other.

orbits()

Returns the orbits of [1,2,...,degree] under the group action.

EXAMPLES:

```
sage: G = PermutationGroup([ [(3,4)], [(1,3)] ])
sage: G.orbits()
[[1, 3, 4], [2]]
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.orbits()
[[1, 2, 3, 4, 10], [5], [6], [7], [8], [9]]
```

The answer is cached:

```
sage: G.orbits() is G.orbits()
True
```

AUTHORS:

- Nathan Dunfield

order()

Return the number of elements of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.order()
12
sage: G = PermutationGroup([()])
sage: G.order()
1
```

```
sage: G = PermutationGroup([])
sage: G.order()
1
```

poincare_series (*p=2, n=10*)

Returns the Poincare series of $G \mod p$ ($p \geq 2$ must be a prime), for n large. In other words, if you input a finite group G, a prime p, and a positive integer n, it returns a quotient of polynomials $f(x) = P(x)/Q(x)$ whose coefficient of x^k equals the rank of the vector space $H_k(G, \mathbf{Z}/p\mathbf{Z})$, for all k in the range $1 \leq k \leq n$.

REQUIRES: GAP package HAP (in gap_packages-*.spkg).

EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: G.poincare_series(2,10)                                   # optional - gap_packages
(x^2 + 1)/(x^4 - x^3 - x + 1)
sage: G = SymmetricGroup(3)
sage: G.poincare_series(2,10)                                   # optional - gap_packages
1/(-x + 1)
```

AUTHORS:

- David Joyner and Graham Ellis

quotient_group (*N*)

Returns the quotient group permgp/N, where N is a normal subgroup. Wraps the GAP operator "/".

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3), (2,3)])
sage: N = PermutationGroup([(1,2,3)])
sage: G.quotient_group(N)
Permutation Group with generators [(1,2)]
```

random_element ()

Return a random element of this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)], [(1,2)]])
sage: G.random_element()
(1,2)(4,5)
```

smallest_moved_point ()

Return the smallest point moved by a permutation in this group.

EXAMPLES:

```
sage: G = PermutationGroup([[(3,4)], [(2,3,4)]])
sage: G.smallest_moved_point()
2
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4,10)]])
sage: G.smallest_moved_point()
1
```

subgroup (*gens*)

Wraps the `PermutationGroup_subgroup` constructor. The argument `gens` is a list of elements of `self`.

EXAMPLES:

```
sage: G = PermutationGroup([(1,2,3),(3,4,5)])
sage: g = G((1,2,3))
sage: G.subgroup([g])
Subgroup of Permutation Group with generators [(3,4,5), (1,2,3)] generated by [(1,2,3)]
```

sylow_subgroup(*p*)
Returns a Sylow p-subgroup of the finite group G, where p is a prime. This is a p-subgroup of G whose index in G is coprime to p. Wraps the GAP function `SylowSubgroup`.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)', '(2,3)'])
sage: G.sylow_subgroup(2)
Permutation Group with generators [(2,3)]
sage: G.sylow_subgroup(5)
Permutation Group with generators [()]
```

trivial_character()
Returns the trivial character of `self`.

EXAMPLES:

```
sage: SymmetricGroup(3).trivial_character()
Character of Symmetric group of order 3! as a permutation group
```

upper_central_series()
Return the upper central series of this group as a list of permutation groups.

EXAMPLES:

These computations use pseudo-random numbers, so we set the seed for reproducible testing.

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: G.upper_central_series()
[Permutation Group with generators [()]]
```

class PermutationGroup_subgroup(*ambient, gens, from_group=False, check=True, canonicalize=True*)
Subgroup subclass of `PermutationGroup_generic`, so instance methods are inherited.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: PermutationGroup_subgroup(H,list(gens))
Subgroup of Dihedral group of order 8 as a permutation group generated by [(1,2,3,4)]
sage: K=PermutationGroup_subgroup(H,list(gens))
sage: K.list()
[(), (1,2,3,4), (1,3)(2,4), (1,4,3,2)]
sage: K.ambient_group()
Dihedral group of order 8 as a permutation group
sage: K.gens()
[(1,2,3,4)]
```

ambient_group()
Return the ambient group related to `self`.

EXAMPLES:

An example involving the dihedral group on four elements, D_8:

```
sage: G = DihedralGroup(4)
sage: H = CyclicPermutationGroup(4)
sage: gens = H.gens()
sage: S = PermutationGroup_subgroup(G, list(gens))
sage: S.ambient_group()
Dihedral group of order 8 as a permutation group
sage: S.ambient_group() == G
True
```

gens()

Return the generators for this subgroup.

EXAMPLES:

An example involving the dihedral group on four elements, D_8:

```
sage: G = DihedralGroup(4)
sage: H = CyclicPermutationGroup(4)
sage: gens = H.gens()
sage: S = PermutationGroup_subgroup(G, list(gens))
sage: S.gens()
[(1,2,3,4)]
sage: S.gens() == list(H.gens())
True
```

direct_product_permgroups(*P*)

Takes the direct product of the permutation groups listed in `P`.

EXAMPLES:

```
sage: G1 = AlternatingGroup([1,2,4,5])
sage: G2 = AlternatingGroup([3,4,6,7])
sage: D = direct_product_permgroups([G1,G2,G1])
sage: D.order()
1728
sage: D = direct_product_permgroups([G1])
sage: D==G1
True
sage: direct_product_permgroups([])
Symmetric group of order 1! as a permutation group
```

from_gap_list(*G*, *src*)

Convert a string giving a list of GAP permutations into a list of elements of `G`.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup import from_gap_list
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]])
sage: L = from_gap_list(G, "[(1,2,3)(4,5), (3,4)]"); L
[(1,2,3)(4,5), (3,4)]
sage: L[0].parent() is G
True
sage: L[1].parent() is G
True
```

load_hap()

Load the GAP hap package into the default GAP interpreter interface. If this fails, try one more time to load it.

EXAMPLES:

```
sage: sage.groups.perm_gps.permgroup.load_hap()
```

22.7 Permutation group elements

AUTHORS:

- David Joyner (2006-02)

- David Joyner (2006-03): word problem method and reorganization
- Robert Bradshaw (2007-11): convert to Cython

EXAMPLES: The Rubik's cube group:

```
sage: f= [(17,19,24,22),(18,21,23,20),(6,25,43,16),(7,28,42,13),(8,30,41,11)]
sage: b=[(33,35,40,38),(34,37,39,36),( 3, 9,46,32),( 2,12,47,29),( 1,14,48,27)]
sage: l=[( 9,11,16,14),(10,13,15,12),( 1,17,41,40),( 4,20,44,37),( 6,22,46,35)]
sage: r=[(25,27,32,30),(26,29,31,28),( 3,38,43,19),( 5,36,45,21),( 8,33,48,24)]
sage: u=[( 1, 3, 8, 6),( 2, 5, 7, 4),( 9,33,25,17),(10,34,26,18),(11,35,27,19)]
sage: d=[(41,43,48,46),(42,45,47,44),(14,22,30,38),(15,23,31,39),(16,24,32,40)]
sage: cube = PermutationGroup([f,b,l,r,u,d])
sage: F=cube.gens()[0]
sage: B=cube.gens()[1]
sage: L=cube.gens()[2]
sage: R=cube.gens()[3]
sage: U=cube.gens()[4]
sage: D=cube.gens()[5]
sage: cube.order()
43252003274489856000
sage: F.order()
4
```

The interested user may wish to explore the following commands: move = cube.random_element() and time word_problem([F,B,L,R,U,D], move, False). This typically takes about 5 minutes (on a 2 Ghz machine) and outputs a word ('solving' the cube in the position move) with about 60 terms or so.

OTHER EXAMPLES: We create element of a permutation group of large degree.

```
sage: G = SymmetricGroup(30)
sage: s = G(srange(30,0,-1)); s
(1,30)(2,29)(3,28)(4,27)(5,26)(6,25)(7,24)(8,23)(9,22)(10,21)(11,20)(12,19)(13,18)(14,17)(15,16)
```

class PermutationGroupElement()

An element of a permutation group.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: G
Permutation Group with generators [(1,2,3)(4,5)]
sage: g = G.random_element()
sage: g in G
True
sage: g = G.gen(0); g
(1,2,3)(4,5)
sage: print g
(1,2,3)(4,5)
sage: g*g
(1,3,2)
sage: g**(-1)
(1,3,2)(4,5)
sage: g**2
(1,3,2)
sage: G = PermutationGroup([(1,2,3)])
sage: g = G.gen(0); g
(1,2,3)
```

```
sage: g.order()
3
```

This example illustrates how permutations act on multivariate polynomials.

```
sage: R = PolynomialRing(RationalField(), 5, ["x","y","z","u","v"])
sage: x, y, z, u, v = R.gens()
sage: f = x**2 - y**2 + 3*z**2
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: sigma = G.gen(0)
sage: f * sigma
3*x^2 + y^2 - z^2
```

cycle_tuples()
Return self as a list of disjoint cycles, represented as tuples rather than permutation group elements.

cycles()
Return self as a list of disjoint cycles.
EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5,6,7)'])
sage: g = G.0
sage: g.cycles()
[(1,2,3), (4,5,6,7)]
sage: a, b = g.cycles()
sage: a(1), b(1)
(2, 1)
```

dict()
Returns list of the images of the integers from 1 to n under this permutation as a list of Python ints.
Note: `list()` returns a zero-indexed list. `dict()` return the actual mapping of the permutation, which will be indexed starting with 1.
EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: g = G((1,2,3,4)); g
(1,2,3,4)
sage: v = g.dict(); v
{1: 2, 2: 3, 3: 4, 4: 1}
sage: type(v[1])
<type 'int'>
sage: x = G([2,1]); x
(1,2)
sage: x.dict()
{1: 2, 2: 1, 3: 3, 4: 4}
```

list()
Returns list of the images of the integers from 1 to n under this permutation as a list of Python ints.
EXAMPLES:

```
sage: G = SymmetricGroup(4)
sage: x = G([2,1,4,3]); x
(1,2)(3,4)
sage: v = x.list(); v
[2, 1, 4, 3]
sage: type(v[0])
<type 'int'>
sage: x = G([2,1]); x
```

```
(1,2)
sage: x.list()
[2, 1, 3, 4]
```

matrix()

Returns deg x deg permutation matrix associated to the permutation self

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: g = G.gen(0)
sage: g.matrix()
[0 1 0 0 0]
[0 0 1 0 0]
[1 0 0 0 0]
[0 0 0 0 1]
[0 0 0 1 0]
```

orbit()

Returns the orbit of the integer n under this group element, as a sorted list of integers.

EXAMPLES:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)'])
sage: g = G.gen(0)
sage: g.orbit(4)
[4, 5]
sage: g.orbit(3)
[1, 2, 3]
sage: g.orbit(10)
[10]
```

order()

Return the order of this group element, which is the smallest positive integer n for which $g^n = 1$.

EXAMPLES:

```
sage: s = PermutationGroupElement('(1,2)(3,5,6)')
sage: s.order()
6
```

TESTS:

```
sage: prod(primes(150))
149218235093927932005887573661584106854758386332686453041O
sage: L = [tuple(range(sum(primes(p))+1, sum(primes(p))+1+p)) for p in primes(150)]
sage: PermutationGroupElement(L).order()
149218235093927932005887573661584106854758386332686453041O
```

sign()

Returns the sign of self, which is $(-1)^s$, where s is the number of swaps.

EXAMPLES:

```
sage: s = PermutationGroupElement('(1,2)(3,5,6)')
sage: s.sign()
-1
```

ALGORITHM: Only even cycles contribute to the sign, thus

$$sign(sigma) = (-1)^{\sum_c len(c)-1}$$

where the sum is over cycles in self.

tuple()
Return tuple of images of integers under self.
EXAMPLES:

```
sage: G = SymmetricGroup(5)
sage: s = G([2,1,5,3,4])
sage: s.tuple()
(2, 1, 5, 3, 4)
```

word_problem()
G and H are permutation groups, g in G, H is a subgroup of G generated by a list (words) of elements of G. If g is in H, return the expression for g as a word in the elements of (words).
This function does not solve the word problem in Sage. Rather it pushes it over to GAP, which has optimized algorithms for the word problem. Essentially, this function is a wrapper for the GAP functions "EpimorphismFromFreeGroup" and "PreImagesRepresentative".
EXAMPLE:

```
sage: G = PermutationGroup([[(1,2,3),(4,5)],[(3,4)]], canonicalize=False)
sage: g1 = G.gens()[0]
sage: g2 = G.gens()[1]
sage: h = g1^2*g2*g1
sage: h.word_problem([g1,g2], False)
('x1^2*x2^-1*x1', '(1,2,3)(4,5)^2*(3,4)^-1*(1,2,3)(4,5)')
sage: h.word_problem([g1,g2])
   x1^2*x2^-1*x1
   [['(1,2,3)(4,5)', 2], ['(3,4)', -1], ['(1,2,3)(4,5)', 1]]
('x1^2*x2^-1*x1', '(1,2,3)(4,5)^2*(3,4)^-1*(1,2,3)(4,5)')
```

gap_format()
Put a permutation in Gap format, as a string.

is_PermutationGroupElement()

make_permgroup_element()

string_to_tuples()
EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_element import string_to_tuples
sage: string_to_tuples('(1,2,3)')
[(1, 2, 3)]
sage: string_to_tuples('(1,2,3)(4,5)')
[(1, 2, 3), (4, 5)]
sage: string_to_tuples(' (1,2, 3) (4,5)')
[(1, 2, 3), (4, 5)]
sage: string_to_tuples('(1,2)(3)')
[(1, 2), (3,)]
```

22.8 Permutation group homomorphisms

AUTHORS:

- David Joyner (2006-03-21): first version
- David Joyner (2008-06): fixed kernel and image to return a group, instead of a string.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)])
sage: phi = PermutationGroupMorphism_im_gens( G, H, gens, gens)
sage: phi.image(G)
Permutation Group with generators [(1,2,3,4)]
sage: phi.kernel()
Permutation Group with generators [()]
sage: phi.image(g)
(1,2,3,4)
sage: phi(g)
(1,2,3,4)
sage: phi.range()
Dihedral group of order 8 as a permutation group
sage: phi.codomain()
Dihedral group of order 8 as a permutation group
sage: phi.domain()
Cyclic group of order 4 as a permutation group
```

class PermutationGroupMap(*parent*)

A set-theoretic map between PermutationGroups.

class PermutationGroupMorphism(*G, H, gensG, imgsH*)

Some python code for wrapping GAP's GroupHomomorphismByImages function but only for permutation groups. Can be expensive if G is large. Returns "fail" if gens does not generate self or if the map does not extend to a group homomorphism, self - other.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: g = G([(1,3),(2,4)]); g
(1,3)(2,4)
sage: phi = PermutationGroupMorphism_im_gens( G, H, gens, gens)
sage: phi
Homomorphism : Cyclic group of order 4 as a permutation group --> Dihedral group of order 8 as a
sage: phi(g)
(1,3)(2,4)
sage: gens1 = G.gens()
sage: gens2 = ((4,3,2,1),)
sage: phi = PermutationGroupMorphism_im_gens( G, G, gens1, gens2)
sage: g = G([(1,2,3,4)]); g
(1,2,3,4)
sage: phi(g)
(1,4,3,2)
```

AUTHORS:

- David Joyner (2006-02)

codomain()

domain()

image(*J*)

J must be a subgroup of G. Computes the subgroup of H which is the image of J.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)])
sage: phi = PermutationGroupMorphism_im_gens( G, H, gens, gens)
sage: phi.image(G)
Permutation Group with generators [(1,2,3,4)]
sage: phi.image(g)
(1,2,3,4)
```

kernel()

range()

class PermutationGroupMorphism_from_gap(*G, H, gap_hom_str, name='phi'*)

This is a Python trick to allow Sage programmers to create a group homomorphism using GAP using very general constructions. An example of its usage is in the direct_product instance method of the PermutationGroup_generic class in permgroup.py.

Basic syntax:

PermutationGroupMorphism_from_gap(domain_group, range_group,'phi:=gap_hom_command;','phi') And don't forget the line: from sage.groups.perm_gps.permgroup_morphism import PermutationGroupMorphism_from_gap in your program.

EXAMPLES:

```
sage: from sage.groups.perm_gps.permgroup_morphism import PermutationGroupMorphism_from_gap
sage: G = PermutationGroup([[(1,2),(3,4)], [(1,2,3,4)]])
sage: H = G.subgroup([G([(1,2,3,4)])])
sage: PermutationGroupMorphism_from_gap(H, G, 'phi:=Identity','phi')
Homomorphism : Subgroup of Permutation Group with generators [(1,2)(3,4), (1,2,3,4)] generated b
```

codomain()

domain()

image(*J*)

J must be an element or a subgroup of G. Computes the subgroup of H which is the image of J.

EXAMPLES:

```
sage: G = PSL(2,7)
sage: D = G.direct_product(G)
sage: H = D[0]
sage: pr1 = D[3]
sage: pr1.image(G)
Permutation Group with generators [(3,7,5)(4,8,6), (1,2,6)(3,4,8)]
sage: G.is_isomorphic(pr1.image(G))
True
```

kernel()

Computes the subgroup of the domain group which is the kernel of self.

EXAMPLES:

```
sage: G = PSL(2,7)
sage: D = G.direct_product(G)
sage: H = D[0]
sage: pr1 = D[3]
sage: G.is_isomorphic(pr1.kernel())
True
```

range()

class PermutationGroupMorphism_id(*X*)

TODO: NOT FINISHED YET!! Return the identity homomorphism from X to itself.

EXAMPLES:

class PermutationGroupMorphism_im_gens(*G, H, gensG, imgsH*)

Some python code for wrapping GAP's GroupHomomorphismByImages function but only for permutation groups. Can be expensive if G is large. Returns "fail" if gens does not generate self or if the map does not extend to a group homomorphism, self - other.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: g = G([(1,3),(2,4)]); g
(1,3)(2,4)
sage: phi = PermutationGroupMorphism_im_gens( G, H, gens, gens)
sage: phi
Homomorphism : Cyclic group of order 4 as a permutation group --> Dihedral group of order 8 as a
sage: phi(g)
(1,3)(2,4)
sage: gens1 = G.gens()
sage: gens2 = ((4,3,2,1),)
sage: phi = PermutationGroupMorphism_im_gens( G, G, gens1, gens2)
sage: g = G([(1,2,3,4)]); g
(1,2,3,4)
sage: phi(g)
(1,4,3,2)
```

AUTHORS:

- David Joyner (2006-02)

codomain()

domain()

image(*J*)

J must be a subgroup of G. Computes the subgroup of H which is the image of J.

EXAMPLES:

```
sage: G = CyclicPermutationGroup(4)
sage: gens = G.gens()
sage: H = DihedralGroup(4)
sage: g = G([(1,2,3,4)])
sage: phi = PermutationGroupMorphism_im_gens( G, H, gens, gens)
sage: phi.image(G)
Permutation Group with generators [(1,2,3,4)]
sage: phi.image(g)
(1,2,3,4)
```

kernel()

range()

gap_format(*x*)

Put a permutation in Gap format, as a string.

is_PermutationGroupMorphism(*f*)

22.9 Rubik's cube group functions

Note: "Rubik's cube" is trademarked. We shall omit the trademark symbol below for simplcity.

NOTATION: B denotes a clockwise quarter turn of the back face D denotes a clockwise quarter turn of the down face and similarly for F (front), L (left), R (right), U (up) Products of moves are read right to left, so for example, R*U means move U first and then R.

See `CubeGroup.parse()` for all possible input notations.

The "Singmaster notation":

- moves: U, D, R, L, F, B as in the diagram below,
- corners: xyz means the facet is on face x (in R,F,L,U,D,B) and the clockwise rotation of the corner sends x-y-z
- edges: xy means the facet is on face x and a flip of the edge sends x-y.

```
sage: rubik = CubeGroup()
sage: rubik.display2d("")
              +--------------+
              |  1    2    3 |
              |  4   top   5 |
              |  6    7    8 |
+-------------+--------------+-------------+------------+
|  9  10  11 | 17   18   19 | 25   26  27 | 33  34  35 |
| 12 left 13 | 20  front 21 | 28 right 29 | 36 rear 37 |
| 14  15  16 | 22   23   24 | 30   31  32 | 38  39  40 |
+-------------+--------------+-------------+------------+
              | 41   42   43 |
              | 44 bottom 45 |
              | 46   47   48 |
              +--------------+
```

AUTHORS:

- David Joyner (2006-10-21): first version
- David Joyner (2007-05): changed faces, added legal and solve
- David Joyner(2007-06): added plotting functions
- David Joyner (2007, 2008): colors corrected, "solve" rewritten (again),typos fixed.
- Robert Miller (2007, 2008): editing, cleaned up display2d
- Robert Bradshaw (2007, 2008): RubiksCube object, 3d plotting.
- David Joyner (2007-09): rewrote docstring for CubeGroup's "solve".
- Robert Bradshaw (2007-09): Versatile parse function for all input types.
- Robert Bradshaw (2007-11): Cleanup.

REFERENCES:

- Cameron, P., Permutation Groups. New York: Cambridge University Press, 1999.
- Wielandt, H., Finite Permutation Groups. New York: Academic Press, 1964.

- Dixon, J. and Mortimer, B., Permutation Groups, Springer-Verlag, Berlin/New York, 1996.
- Joyner,D., Adventures in Group Theory, Johns Hopkins Univ Press, 2002.

class CubeGroup()

A python class to help compute Rubik's cube group actions.

EXAMPLES: If G denotes the cube group then it may be regarded as a subgroup of SymmetricGroup(48), where the 48 facets are labeled as follows.

```
sage: rubik = CubeGroup()
sage: rubik.display2d("")
             +--------------+
             |  1    2    3 |
             |  4   top   5 |
             |  6    7    8 |
+------------+--------------+-------------+------------+
|  9  10  11 | 17   18   19 | 25   26  27 | 33  34  35 |
| 12 left 13 | 20  front 21 | 28 right 29 | 36 rear 37 |
| 14  15  16 | 22   23   24 | 30   31  32 | 38  39  40 |
+------------+--------------+-------------+------------+
             | 41   42   43 |
             | 44 bottom 45 |
             | 46   47   48 |
             +--------------+


sage: rubik
The PermutationGroup of all legal moves of the Rubik's cube.
sage: print rubik
The Rubik's cube group with genrators R,L,F,B,U,D in SymmetricGroup(48).
```

B()

D()

F()

L()

R()

U()

display2d(*mv*)

faces(*mv*)

Returns the dictionary of faces created by the effect of the move mv, which is a string of the form $X^a * Y^b * ...$, where X, Y, ... are in $\{R, L, F, B, U, D\}$ and $a, b, ...$ are integers. We call this ordering of the faces the "BDFLRU, L2R, T2B ordering".

EXAMPLES:

```
sage: rubik = CubeGroup()
```

Now type `rubik.faces("")` for the dictionary of the solved state and `rubik.faces("R*L")` for the dictionary of the state obtained after making the move R followed by L.

facets(*g=None*)

Returns the set of facets on which the group acts. This function is a "constant".

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.facets()==range(1,49)
True
```

gen_names()

gens()

group()

legal(*state, mode='quiet'*)

Returns 1 (true) if the dictionary `state` (in the same format as returned by the faces method) represents a legal position (or state) of the Rubik's cube. Returns 0 (false) otherwise.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: G = rubik.group()
sage: r0 = rubik.faces("")
sage: r1 = {'back': [[33, 34, 35], [36, 0, 37], [38, 39, 40]], 'down': [[41, 42, 43], [44, 0
sage: rubik.legal(r0)
1
sage: rubik.legal(r0,"verbose")
(1, ())
sage: rubik.legal(r1)
0
```

move(*mv*)

Returns the group element and the reordered list of facets, as moved by the list mv (read left-to-right)

INPUT: mv is a string of the form `Xa*Yb*...`, where X, Y, ... are in R,L,F,B,U,D and a,b, ... are integers.

EXAMPLES:

```
  sage: rubik = CubeGroup()
sage: rubik.move("")[0]
()
sage: rubik.move("R")[0]
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
sage: rubik.R()
(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)
```

parse(*mv*)

This function allows one to create the permutation group element from a variety of formats.

INPUT: one of the following

- `list` - list of facets (as returned by self.facets())
- `dict` - list of faces (as returned by self.faces())
- `str` - either cycle notation (passed to GAP) or a product of generators or Singmaster notation
- `perm_group element` - returned as an element of self.group()

EXAMPLES:

```
sage: C = CubeGroup()
sage: C.parse(range(1,49))
()
sage: g = C.parse("L"); g
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
sage: C.parse(str(g)) == g
True
sage: facets = C.facets(g); facets
[17, 2, 3, 20, 5, 22, 7, 8, 11, 13, 16, 10, 15, 9, 12, 14, 41, 18, 19, 44, 21, 46, 23, 24, 2
sage: C.parse(facets)
(1,17,41,40)(4,20,44,37)(6,22,46,35)(9,11,16,14)(10,13,15,12)
sage: C.parse(facets) == g
True
sage: faces = C.faces("L"); faces
{'right': [[25, 26, 27], [28, 0, 29], [30, 31, 32]], 'up': [[17, 2, 3], [20, 0, 5], [22, 7,
```

```
sage: C.parse(faces) == C.parse("L")
True
sage: C.parse("L' R2") == C.parse("L^(-1)*R^2")
True
sage: C.parse("L' R2")
(1,40,41,17)(3,43)(4,37,44,20)(5,45)(6,35,46,22)(8,48)(9,14,16,11)(10,12,15,13)(19,38)(21,36
sage: C.parse("L^4")
()
sage: C.parse("L^(-1)*R")
(1,40,41,17)(3,38,43,19)(4,37,44,20)(5,36,45,21)(6,35,46,22)(8,33,48,24)(9,14,16,11)(10,12,1
```

plot3d_cube(*mv, title=True*)

Displays F,U,R faces of the cube after the given move mv, where mv is a string in the Singmaster notation. Mostly included for the purpose of drawing pictures and checking moves.

The first one below is "superflip+4 spot" (in 26q* moves) and the second one is the superflip (in 20f* moves). Type show(P) to view them.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: P = rubik.plot3d_cube("U^2*F*U^2*L*R^(-1)*F^2*U*F^3*B^3*R*L*U^2*R*D^3*U*L^3*R*D*R^3*L^
sage: P = rubik.plot3d_cube("R*L*D^2*B^3*L^2*F^2*R^2*U^3*D*R^3*D^2*F^3*B^3*D^3*F^2*D^3*R^2*U
```

plot_cube(*mv, title=True, colors=, [(1, 0.63, 1), (1, 1, 0), (1, 0, 0), (0, 1, 0), (1, 0.59999999999999998, 0.29999999999999999), (0, 0, 1)]*)

Input the move mv, as a string in the Singmaster notation, and output the 2-d plot of the cube in that state.

Type P.show() to display any of the plots below.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: P = rubik.plot_cube("R^2*U^2*R^2*U^2*R^2*U^2", title = False)
sage: # (R^2U^2)^3  permutes 2 pairs of edges (uf,ub)(fr,br)
sage: P = rubik.plot_cube("R*L*D^2*B^3*L^2*F^2*R^2*U^3*D*R^3*D^2*F^3*B^3*D^3*F^2*D^3*R^2*U^3
sage: # the superflip (in 20f* moves)
sage: P = rubik.plot_cube("U^2*F*U^2*L*R^(-1)*F^2*U*F^3*B^3*R*L*U^2*R*D^3*U*L^3*R*D*R^3*L^3*
sage: # "superflip+4 spot" (in 26q* moves)
```

repr2d(*mv*)

Displays a 2d map of the Rubik's cube after the move mv has been made. Nothing is returned.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: rubik.display2d("")
             +-------------+
             |  1    2    3 |
             |  4   top   5 |
             |  6    7    8 |
+------------+-------------+-------------+------------+
|  9  10  11 | 17   18   19 | 25   26  27 | 33  34  35 |
| 12 left 13 | 20  front 21 | 28 right 29 | 36 rear 37 |
| 14  15  16 | 22   23   24 | 30   31  32 | 38  39  40 |
+------------+-------------+-------------+------------+
             | 41   42   43 |
             | 44 bottom 45 |
             | 46   47   48 |
             +-------------+

sage: rubik.display2d("R")
             +-------------+
```

```
                |  1    2   38 |
                |  4   top  36 |
                |  6    7   33 |
    +-----------+--------------+-------------+-----------+
    |  9  10  11 | 17   18    3 | 27   29  32 | 48  34  35 |
    | 12 left 13 | 20  front  5 | 26 right 31 | 45 rear 37 |
    | 14  15  16 | 22   23    8 | 25   28  30 | 43  39  40 |
    +-----------+--------------+-------------+-----------+
                | 41   42   19 |
                | 44 bottom 21 |
                | 46   47   24 |
                +--------------+
```

You can see the right face has been rotated but not the left face.

solve (*state, algorithm='default'*)

Solves the cube in the `state`, given as a dictionary as in `legal`. See the `solve` method of the RubiksCube class for more details.

This may use GAP's `EpimorphismFromFreeGroup` and `PreImagesRepresentative` as explained below, if 'gap' is passed in as the algorithm.

This algorithm

1. constructs the free group on 6 generators then computes a reasonable set of relations which they satisfy
2. computes a homomorphism from the cube group to this free group quotient
3. takes the cube position, regarded as a group element, and maps it over to the free group quotient
4. using those relations and tricks from combinatorial group theory (stabilizer chains), solves the "word problem" for that element.
5. uses python string parsing to rewrite that in cube notation.

The Rubik's cube group has about 4.3×10^{19} elements, so this process is time-consuming. See http://www.gap-system.org/Doc/Examples/rubik.html for an interesting discussion of some GAP code analyzing the Rubik's cube.

EXAMPLES:

```
sage: rubik = CubeGroup()
sage: state = rubik.faces("R")
sage: rubik.solve(state)
'R'
sage: state = rubik.faces("R*U")
sage: rubik.solve(state, algorithm='gap')        # long time
'R*U'
```

You can also check this another (but similar) way using the `word_problem` method (eg, G = rubik.group(); g = G("(3,38,43,19)(5,36,45,21)(8,33,48,24)(25,27,32,30)(26,29,31,28)"); g.word_problem([b,d,f,l,r,u]), though the output will be less intuitive).

class RubiksCube (*state=None, history=, [], colors=, [(1, 0.63, 1), (1, 1, 0), (1, 0, 0), (0, 1, 0), (1, 0.5999999999999998, 0.2999999999999999), (0, 0, 1)]*)

```
sage: C = RubiksCube().move("R U R'")
sage: C.show3d()
```

```
sage: C = RubiksCube("R*L"); C
                +--------------+
                | 17    2   38 |
                | 20   top  36 |
                | 22    7   33 |
    +-----------+--------------+-------------+-----------+
```

```
| 11  13  16 | 41   18    3 | 27   29  32 | 48  34   6 |
| 10 left 15 | 44  front  5 | 26 right 31 | 45 rear  4 |
|  9  12  14 | 46   23    8 | 25   28  30 | 43  39   1 |
+------------+--------------+-------------+------------+
             | 40   42   19 |
             | 37 bottom 21 |
             | 35   47   24 |
             +--------------+
sage: C.show()
sage: C.solve(algorithm='gap')  # long time
'L R'
sage: C == RubiksCube("L*R")
True
```

cubie(*size, gap, x, y, z, colors, stickers=True*)

facets()

move(*g*)

plot()

plot3d(*stickers=True*)

```
sage: C = RubiksCube().move("R*U")
sage: C.plot3d()
sage: C.plot()
```

scramble(*moves=30*)

show()

show3d()

solve(*algorithm='hybrid', timeout=15*)

Algorithm must be one of : hybrid - try kociemba for timeout seconds, then dietz kociemba - Use Dik T. Winter's program (reasonable speed, few moves) dietz - Use Eric Dietz's cubex program (fast but lots of moves) optimal - Use Michael Reid's optimal program (may take a long time) gap - Use GAP word solution (can be slow)

EXAMPLE:

```
sage: C = RubiksCube("R U F L B D")
sage: C.solve()
'R U F L B D'
```

Dietz's program is much faster, but may give highly non-optimal solutions.

```
sage: s = C.solve('dietz'); s
"U' L' L' U L U' L U D L L D' L' D L' D' L D L' U' L D' L' U L' B' U' L' U B L D L D' U' L'
sage: C2 = RubiksCube(s)
sage: C == C2
True
```

undo()

color_of_square(*facet, colors=, ['lpurple', 'yellow', 'red', 'green', 'orange', 'blue']*)

Returns the color the facet has in the solved state.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import *
sage: color_of_square(41)
'blue'
```

create_poly(*face, color*)

cubie_centers(*label*)

cubie_colors(*label, state0*)

cubie_faces()

This provides a map from the 6 faces of the 27 cubies to the 48 facets of the larger cube.

-1,-1,-1 is left, top, front

index2singmaster(*facet*)

Translates index used (eg, 43) to Singmaster facet notation (eg, fdr).

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import *
sage: index2singmaster(41)
'dlf'
```

inv_list(*lst*)

Input a list of ints 1, ..., m (in any order), outputs inverse perm.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import *
sage: L = [2,3,1]
sage: inv_list(L)
[3, 1, 2]
```

plot3d_cubie(*cnt, clrs*)

Plots the front, up and right face of a cubie centered at cnt and rgbcolors given by clrs (in the order FUR).

Type P.show() to view.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import *
sage: clrF = blue; clrU = red; clrR = green
sage: P = plot3d_cubie([1/2,1/2,1/2],[clrF,clrU,clrR])
```

polygon_plot3d(*points, tilt=30, turn=30, **kwargs*)

Plots a polygon viewed from an angle determined by tilt, turn, and vertices points.

> **Warning:** The ordering of the points is important to get "correct" and if you add several of these plots together, the one added first is also drawn first (ie, addition of Graphics objects is not commutative).

The following example produced a green-colored square with vertices at the points indicated.

EXAMPLES:

```
sage: from sage.groups.perm_gps.cubegroup import *
sage: P = polygon_plot3d([[1,3,1],[2,3,1],[2,3,2],[1,3,2],[1,3,1]],rgbcolor=green)
```

rotation_list(*tilt, turn*)

xproj(*x, y, z, r*)

yproj(*x, y, z, r*)

22.10 Matrix Groups

AUTHORS:

- William Stein: initial version
- David Joyner (2006-03-15): degree, base_ring, _contains_, list, random, order methods; examples
- William Stein (2006-12): rewrite
- David Joyner (2007-12): Added invariant_generators (with Martin Albrecht and Simon King)
- David Joyner (2008-08): Added module_composition_factors (interface to GAP's MeatAxe implementation) and as_permutation_group (returns isomorphic PermutationGroup).

This class is designed for computing with matrix groups defined by a (relatively small) finite set of generating matrices.

EXAMPLES:

```
sage: F = GF(3)
sage: gens = [matrix(F,2, [1,0, -1,1]), matrix(F, 2, [1,1,0,1])]
sage: G = MatrixGroup(gens)
sage: G.conjugacy_class_representatives()
[
[1 0]
[0 1],
[0 1]
[2 1],
[0 1]
[2 2],
[0 2]
[1 1],
[0 2]
[1 2],
[0 1]
[2 0],
[2 0]
[0 2]
]
```

Loading and saving work:

```
sage: G = GL(2,5); G
General Linear Group of degree 2 over Finite Field of size 5
sage: loads(dumps(G)) == G
True
sage: g = G.1; g
[4 1]
[4 0]
sage: loads(dumps(g)) == g
True
```

MatrixGroup(*gens*)

Return the matrix group with given generators.

INPUT:

•`gens` - list of matrices in a matrix space or matrix group

EXAMPLES:

```
sage: F = GF(5)
sage: gens = [matrix(F,2,[1,2, -1, 1]), matrix(F,2, [1,1, 0,1])]
sage: G = MatrixGroup(gens); G
Matrix group over Finite Field of size 5 with 2 generators:
[[[1, 2], [4, 1]], [[1, 1], [0, 1]]]
```

In the second example, the generators are a matrix over $\mathbf{Z}$, a matrix over a finite field, and the integer 2. Sage determines that they both canonically map to matrices over the finite field, so creates that matrix group there.

```
sage: gens = [matrix(2,[1,2, -1, 1]), matrix(GF(7), 2, [1,1, 0,1]), 2]
sage: G = MatrixGroup(gens); G
Matrix group over Finite Field of size 7 with 3 generators:
[[[1, 2], [6, 1]], [[1, 1], [0, 1]], [[2, 0], [0, 2]]]
```

Each generator must be invertible:

```
sage: G = MatrixGroup([matrix(ZZ,2,[1,2,3,4])])
...
ValueError: each generator must be an invertible matrix but one is not:
[1 2]
[3 4]
```

Some groups aren't supported:

```
sage: SL(2, CC).gens()
...
NotImplementedError: Matrix group over Complex Field with 53 bits of precision not implemented.
sage: G = SL(0, QQ)
...
ValueError: The degree must be at least 1
```

class MatrixGroup_gap(*n, R, var='a'*)

as_matrix_group()

Return this group, but as a general matrix group, i.e., throw away the extra structure of general unitary group.

EXAMPLES:

```
sage: G = SU(4,GF(5))
sage: G.as_matrix_group()
Matrix group over Finite Field in a of size 5^2 with 2 generators:
[[[a, 0, 0, 0], [0, 2*a + 3, 0, 0], [0, 0, 4*a + 1, 0], [0, 0, 0, 3*a]], [[1, 0, 4*a + 3, 0]

sage: G = GO(3,GF(5))
sage: G.as_matrix_group()
Matrix group over Finite Field of size 5 with 2 generators:
[[[2, 0, 0], [0, 3, 0], [0, 0, 1]], [[0, 1, 0], [1, 4, 4], [0, 2, 1]]]
```

base_field()

Return the base ring of this matrix group.

EXAMPLES:

```
sage: GL(2,GF(3)).base_ring()
Finite Field of size 3
sage: G = SU(3,GF(5))
sage: G.base_ring()
```

```
Finite Field of size 5
sage: G.field_of_definition()
Finite Field in a of size 5^2
```

base_ring()
Return the base ring of this matrix group.
EXAMPLES:

```
sage: GL(2,GF(3)).base_ring()
Finite Field of size 3
sage: G = SU(3,GF(5))
sage: G.base_ring()
Finite Field of size 5
sage: G.field_of_definition()
Finite Field in a of size 5^2
```

degree()
Return the degree of this matrix group.
EXAMPLES:

```
sage: SU(5,5).degree()
5
```

field_of_definition(*var='a'*)
Return a field that contains all the matrices in this matrix group.
EXAMPLES:

```
sage: G = SU(3,GF(5))
sage: G.base_ring()
Finite Field of size 5
sage: G.field_of_definition()
Finite Field in a of size 5^2
sage: G = GO(4,GF(7),1)
sage: G.field_of_definition()
Finite Field of size 7
sage: G.base_ring()
Finite Field of size 7
```

gen(*n*)
Return the n-th generator.
EXAMPLES:

```
sage: G = GU(4,GF(5), var='beta')
sage: G.gen(0)
[  beta      0      0      0]
[     0      1      0      0]
[     0      0      1      0]
[     0      0      0 3*beta]
```

gens()
Return generators for this matrix group.
EXAMPLES:

```
sage: G = GO(3,GF(5))
sage: G.gens()
[
[2 0 0]
[0 3 0]
[0 0 1],
```

```
[0 1 0]
[1 4 4]
[0 2 1]
]
```

hom(*x*)

irreducible_characters()

Returns the list of irreducible characters of the group.

EXAMPLES:

```
sage: G = GL(2,2)
sage: G.irreducible_characters()
[Character of General Linear Group of degree 2 over Finite Field of size 2,
 Character of General Linear Group of degree 2 over Finite Field of size 2,
 Character of General Linear Group of degree 2 over Finite Field of size 2]
```

is_finite()

Return True if this matrix group is finite.

EXAMPLES:

```
sage: G = GL(2,GF(3))
sage: G.is_finite()
True
sage: SL(2,ZZ).is_finite()
False
```

list()

Return list of all elements of this group.

Always returns a new list, so it is safe to change the returned list.

EXAMPLES:

```
sage: F = GF(3)
sage: gens = [matrix(F,2, [1,0, -1,1]), matrix(F, 2, [1,1,0,1])]
sage: G = MatrixGroup(gens)
sage: G.order()
24
sage: v = G.list()
sage: len(v)
24
sage: v[:2]
[[0 1]
[2 0], [0 1]
[2 1]]
sage: G.list()[0] in G
True
```

An example over a ring (see trac 5241):

```
sage: M1 = matrix(ZZ,2,[[-1,0],[0,1]])
sage: M2 = matrix(ZZ,2,[[1,0],[0,-1]])
sage: M3 = matrix(ZZ,2,[[-1,0],[0,-1]])
sage: MG = MatrixGroup([M1, M2, M3])
sage: MG.list()
[[-1  0]
[ 0 -1], [-1  0]
[ 0  1], [ 1  0]
[ 0 -1], [1 0]
[0 1]]
sage: MG.list()[1]
```

```
[-1  0]
[ 0  1]
sage: MG.list()[1].parent()
Matrix group over Integer Ring with 3 generators:
[[[-1, 0], [0, 1]], [[1, 0], [0, -1]], [[-1, 0], [0, -1]]]

sage: GL(2,ZZ).list()
...
ValueError: group must be finite
```

matrix_space()

Return the matrix space corresponding to this matrix group.

This is a matrix space over the field of definition of this matrix group.

EXAMPLES:

```
sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: G = MatrixGroup([MS(1), MS([1,2,3,4])])
sage: G.matrix_space()
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 5
```

ngens()

Return the number of generators of this linear group.

EXAMPLES:

```
sage: G = GO(3,GF(5))
sage: G.ngens()
2
```

order()

EXAMPLES:

```
sage: G = Sp(4,GF(3))
sage: G.order()
51840
sage: G = SL(4,GF(3))
sage: G.order()
12130560
sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.order()
480
sage: G = MatrixGroup([matrix(ZZ,2,[1,1,0,1])])
sage: G.order()
+Infinity
```

class **MatrixGroup_gap_finite_field**(*n, R, var='a'*)

Python class for matrix groups over a finite field.

center()

Return the center of this linear group as a matrix group.

EXAMPLES:

```
sage: G = SU(3,GF(2))
sage: G.center()
Matrix group over Finite Field in a of size 2^2 with 1 generators:
 [[[a, 0, 0], [0, a, 0], [0, 0, a]]]
sage: GL(2,GF(3)).center()
Matrix group over Finite Field of size 3 with 1 generators:
```

```
 [[[2, 0], [0, 2]]]
sage: GL(3,GF(3)).center()
Matrix group over Finite Field of size 3 with 1 generators:
 [[[2, 0, 0], [0, 2, 0], [0, 0, 2]]]
sage: GU(3,GF(2)).center()
Matrix group over Finite Field in a of size 2^2 with 1 generators:
 [[[a + 1, 0, 0], [0, a + 1, 0], [0, 0, a + 1]]]
```

conjugacy_class_representatives()

Return a set of representatives for each of the conjugacy classes of the group.

EXAMPLES:

```
sage: G = SU(3,GF(2))
sage: len(G.conjugacy_class_representatives())
16
sage: len(GL(2,GF(3)).conjugacy_class_representatives())
8
sage: len(GU(2,GF(5)).conjugacy_class_representatives())
36
```

order()

EXAMPLES:

```
sage: G = Sp(4,GF(3))
sage: G.order()
51840
sage: G = SL(4,GF(3))
sage: G.order()
12130560
sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.order()
480
sage: G = MatrixGroup([matrix(ZZ,2,[1,1,0,1])])
sage: G.order()
+Infinity
```

random()

Deprecated. Use self.random_element() instead.

random_element()

Return a random element of this group.

EXAMPLES:

```
sage: G = Sp(4,GF(3))
sage: G.random_element()
[2 1 1 1]
[1 0 2 1]
[0 1 1 0]
[1 0 0 1]

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.random_element()
[1 3]
[0 3]
sage: G.random_element()
```

```
[2 2]
[1 0]
sage: G.random_element()
[4 0]
[1 4]
```

class MatrixGroup_generic()

class MatrixGroup_gens(*gensG*)

EXAMPLES:

A ValueError is raised if one of the generators is not invertible.

```
sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: G = MatrixGroup([MS.0])
...
ValueError: each generator must be an invertible matrix but one is not:
[1 0]
[0 0]
```

as_permutation_group(*method=None*)

This returns a permutation group representation for the group. In most cases occurring in practice, this is a permutation group of minimal degree (the degree begin determined from orbits under the group action). When these orbits are hard to compute, the procedure can be time-consuming and the degree may not be minimal. The "method=smaller" option tries return an isomorphic group of lower degree.

EXAMPLES:

```
sage: MS = MatrixSpace( GF(2), 5, 5)
sage: A = MS([[0,0,0,0,1],[0,0,0,1,0],[0,0,1,0,0],[0,1,0,0,0],[1,0,0,0,0]])
sage: G = MatrixGroup([A])
sage: G.as_permutation_group()
Permutation Group with generators [(1,2)]
sage: MS = MatrixSpace( GF(7), 12, 12)
sage: GG = gap("ImfMatrixGroup( 12, 3 )")
sage: GG.GeneratorsOfGroup().Length()
3
sage: g1 = MS(eval(str(GG.GeneratorsOfGroup()[1]).replace("\n","")))
sage: g2 = MS(eval(str(GG.GeneratorsOfGroup()[2]).replace("\n","")))
sage: g3 = MS(eval(str(GG.GeneratorsOfGroup()[3]).replace("\n","")))
sage: G = MatrixGroup([g1, g2, g3])
sage: G.order()
21499084800
sage: set_random_seed(0); current_randstate().set_seed_gap()
sage: G.as_permutation_group()
Permutation Group with generators [(2,3,5,11,20,38,14,26,43,66)(4,8,16,29,47,63,48,74,56,86)
sage: set_random_seed(3); current_randstate().set_seed_gap()
sage: G.as_permutation_group(method="smaller")
Permutation Group with generators [(1,2)(3,7,13,25,45,5,10,19,35,60)(8,16,30,41,69,11,22,40,
```

In this case, the "smaller" option returned an isomorphic group of lower degree. The above example used GAP's library of irreducible maximal finite ("imf") integer matrix groups to construct the MatrixGroup G over GF(7). The section "Irreducible Maximal Finite Integral Matrix Groups" in the GAP reference manual has more details.

gens()

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]
```

```
sage: G = MatrixGroup(gens)
sage: gens[0] in G
True
sage: gens = G.gens()
sage: gens[0] in G
True
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]

sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: G = MatrixGroup([MS(1), MS([1,2,3,4])])
sage: G
Matrix group over Finite Field of size 5 with 2 generators:
 [[[1, 0], [0, 1]], [[1, 2], [3, 4]]]
sage: G.gens()
[[1 0]
[0 1], [1 2]
[3 4]]
```

invariant_generators()

Wraps Singular's invariant_algebra_reynolds and invariant_ring in finvar.lib, with help from Simon King and Martin Albrecht. Computes generators for the polynomial ring $F[x_1, \ldots, x_n]^G$, where G in GL(n,F) is a finite matrix group.

In the "good characteristic" case the polynomials returned form a minimal generating set for the algebra of G-invariant polynomials. In the "bad" case, the polynomials returned are primary and secondary invariants, forming a not necessarily minimal generating set for the algebra of G-invariant polynomials.

EXAMPLES:

```
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[0,1],[-1,0]]),MS([[1,1],[2,3]])]
sage: G = MatrixGroup(gens)
sage: G.invariant_generators()
[x1^7*x2 - x1*x2^7, x1^12 - 2*x1^9*x2^3 - x1^6*x2^6 + 2*x1^3*x2^9 + x2^12, x1^18 + 2*x1^15*x
sage: q = 4; a = 2
sage: MS = MatrixSpace(QQ, 2, 2)
sage: gen1 = [[1/a,(q-1)/a],[1/a, -1/a]]; gen2 = [[1,0],[0,-1]]; gen3 = [[-1,0],[0,1]]
sage: G = MatrixGroup([MS(gen1),MS(gen2),MS(gen3)])
sage: G.order()
12
sage: G.invariant_generators()
[x1^2 + 3*x2^2, x1^6 + 15*x1^4*x2^2 + 15*x1^2*x2^4 + 33*x2^6]
sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,2],[-1,1]]),MS([[1,1],[-1,1]])]
sage: G = MatrixGroup(gens)
sage: G.invariant_generators()  ## takes a long time (several mins)
[x1^20 + x1^16*x2^4 + x1^12*x2^8 + x1^8*x2^12 + x1^4*x2^16 + x2^20, x1^20*x2^4 + x1^16*x2^8
sage: F=CyclotomicField(8)
sage: z=F.gen()
sage: a=z+1/z
sage: b=z^2
sage: MS=MatrixSpace(F,2,2)
sage: g1=MS([[1/a,1/a],[1/a,-1/a]])
sage: g2=MS([[1,0],[0,b]])
sage: g3=MS([[b,0],[0,1]])
sage: G=MatrixGroup([g1,g2,g3])
sage: G.invariant_generators()
[x1^8 + 14*x1^4*x2^4 + x2^8,
 x1^24 + 10626/1025*x1^20*x2^4 + 735471/1025*x1^16*x2^8 + 2704156/1025*x1^12*x2^12 + 735471/
```

AUTHORS:

•David Joyner, Simon King and Martin Albrecht.

REFERENCES:

•Singular reference manual

• 1.Sturmfels, "Algorithms in invariant theory", Springer-Verlag, 1993.

•S. King, "Minimal Generating Sets of non-modular invariant rings of finite groups", arXiv:math.AC/0703035

module_composition_factors(*method=None*)

Returns a list of triples consisting of [base field, dimension, irreducibility], for each of the Meataxe composition factors modules. The method="verbose" option returns more information, but in Meataxe notation.

EXAMPLES:

```
sage: F=GF(3);MS=MatrixSpace(F,4,4)
sage: M=MS(0)
sage: M[0,1]=1;M[1,2]=1;M[2,3]=1;M[3,0]=1
sage: G = MatrixGroup([M])
sage: G.module_composition_factors()
[[Finite Field of size 3, 1, True],
 [Finite Field of size 3, 1, True],
 [Finite Field of size 3, 2, True]]
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[0,1],[-1,0]]),MS([[1,1],[2,3]])]
sage: G = MatrixGroup(gens)
sage: G.module_composition_factors()
[[Finite Field of size 7, 2, True]]
```

Type "G.module_composition_factors(method='verbose')" to get a more verbose version.

For more on MeatAxe notation, see http://www.gap-system.org/Manuals/doc/htm/ref/CHAP067.htm

class MatrixGroup_gens_finite_field(*gensG*)

is_MatrixGroup(*x*)

EXAMPLES:

```
sage: from sage.groups.matrix_gps.matrix_group import is_MatrixGroup
sage: is_MatrixGroup(MatrixSpace(QQ,3))
False
sage: is_MatrixGroup(Mat(QQ,3))
False
sage: is_MatrixGroup(GL(2,ZZ))
True
sage: is_MatrixGroup(MatrixGroup([matrix(2,[1,1,0,1])]))
True
```

22.11 Matrix Group Elements

AUTHORS:

- David Joyner (2006-05): initial version David Joyner
- David Joyner (2006-05): various modifications to address William Stein's TODO's.
- William Stein (2006-12-09): many revisions.

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens); G
Matrix group over Finite Field of size 3 with 2 generators:
 [[[1, 0], [0, 1]], [[1, 1], [0, 1]]]
sage: g = G([[1,1],[0,1]])
sage: h = G([[1,2],[0,1]])
sage: g*h
[1 0]
[0 1]
```

You cannot add two matrices, since this is not a group operation. You can coerce matrices back to the matrix space and add them there:

```
sage: g + h
...
TypeError: unsupported operand type(s) for +: 'MatrixGroupElement' and 'MatrixGroupElement'


sage: g.matrix() + h.matrix()
[2 0]
[0 2]


sage: 2*g
...
TypeError: unsupported operand parent(s) for '*': 'Integer Ring' and 'Matrix group over Finite Field
 [[[1, 0], [0, 1]], [[1, 1], [0, 1]]]'
sage: 2*g.matrix()
[2 2]
[0 2]
```

class **MatrixGroupElement** (*g, parent, check=True*)

An element of a matrix group.

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
       sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: g = G.random_element()
sage: type(g)
<class 'sage.groups.matrix_gps.matrix_group_element.MatrixGroupElement'>
```

list ()

Return list representation of this matrix.

EXAMPLES:

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[1,0],[0,1]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: g = G.0
sage: g.list()
[[1, 0], [0, 1]]
```

matrix ()

Obtain the usual matrix (as an element of a matrix space) associated to this matrix group element.

One reason to compute the associated matrix is that matrices support a huge range of functionality.

EXAMPLES:

```
sage: k = GF(7); G = MatrixGroup([matrix(k,2,[1,1,0,1]), matrix(k,2,[1,0,0,2])])
sage: g = G.0
sage: g.matrix()
[1 1]
[0 1]
sage: parent(g.matrix())
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 7
```

Matrices have extra functionality that matrix group elements do not have.

```
sage: g.matrix().charpoly('t')
t^2 + 5*t + 1
```

order()

Return the order of this group element, which is the smallest positive integer n such that $g^n = 1$, or +Infinity if no such integer exists.

EXAMPLES:

```
sage: k = GF(7); G = MatrixGroup([matrix(k,2,[1,1,0,1]), matrix(k,2,[1,0,0,2])])
sage: G
Matrix group over Finite Field of size 7 with 2 generators:
 [[[1, 1], [0, 1]], [[1, 0], [0, 2]]]
sage: G.order()
21
```

See trac #1170

```
sage: gl=GL(2,ZZ); gl
General Linear Group of degree 2 over Integer Ring
sage: g=gl.gens()[2]; g
[1 1]
[0 1]
sage: g.order()
+Infinity
```

word_problem(*words=None*)

Right this group element in terms of the elements of the list `words`.

If G and H are permutation groups (with G the parent of self), H is a subgroup of G generated by a list `words` of elements of G. If g is in H, return the expression for g as a word in the elements of `words`.

ALGORITHM: Use GAP, which has optimized algorithms for solving the word problem (the GAP functions EpimorphismFromFreeGroup and PreImagesRepresentative).

INPUT:

- `words` - list (default: None) a list of elements of the parent group; if words is empty, uses the gens of the parent group.

EXAMPLE:

```
sage: G = GL(2,5); G
General Linear Group of degree 2 over Finite Field of size 5
sage: G.gens()
[
[2 0]
[0 1],
[4 1]
[4 0]
]
sage: G(1).word_problem([G.1, G.0])
1
```

Next we construct a more complicated element of the group from the generators:

```
sage: (G.0).order(), (G.1).order()
(4, 3)
sage: g = G.0^3 * G.1^2
```

We then ask to solve the word problem, with the generators reversed (just to make it trickier):

```
sage: s = g.word_problem([G.1, G.0]); s
([2 0]
[0 1])^-1 *
([4 1]
[4 0])^-1
```

It worked!

```
sage: s.prod() == g
True
```

AUTHORS:

- David Joyner and William Stein

is_MatrixGroupElement(*x*)

22.12 Homomorphisms Between Matrix Groups

AUTHORS:

- David Joyner and William Stein (2006-03): initial version
- David Joyner (2006-05): examples

class MatrixGroupMap(*parent*)
A set-theoretic map between matrix groups.

class MatrixGroupMorphism(*parent*)

class MatrixGroupMorphism_im_gens(*homset, imgsH, check=True*)
Some python code for wrapping GAP's GroupHomomorphismByImages function but only for matrix groups. Can be expensive if G is large. Returns "fail" if gens does not generate self or if the map does not extend to a group homomorphism, self - other.

TODO: what does it mean to return fail? It's a constructor for a class.

EXAMPLES:

```
sage: F = GF(5); MS = MatrixSpace(F,2,2)
sage: G = MatrixGroup([MS([1,1,0,1])])
sage: H = MatrixGroup([MS([1,0,1,1])])
sage: phi = G.hom(H.gens())
sage: phi
Homomorphism : Matrix group over Finite Field of size 5 with 1 generators:
 [[[1, 1], [0, 1]]] --> Matrix group over Finite Field of size 5 with 1 generators:
 [[[1, 0], [1, 1]]]
sage: phi(MS([1,1,0,1]))
[1 0]
[1 1]
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: F.multiplicative_generator()
3
```

```
sage: G = MatrixGroup([MS([3,0,0,1])])
sage: a = G.gens()[0]^2
sage: phi = G.hom([a])
```

image(*J*)

J must be a subgroup of G. Computes the subgroup of H which is the image of J.

EXAMPLES:

```
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: F.multiplicative_generator()
3
sage: G = MatrixGroup([MS([3,0,0,1])])
sage: a = G.gens()[0]^2
sage: phi = G.hom([a])
sage: phi.image(G.gens()[0])
'[ [ Z(7)^2, 0*Z(7) ], [ 0*Z(7), Z(7)^0 ] ]'
sage: H = MatrixGroup([MS(a.list())])
sage: H
Matrix group over Finite Field of size 7 with 1 generators:
[[[2, 0], [0, 1]]]
sage: phi.image(H)
'Group([ [ [ Z(7)^4, 0*Z(7) ], [ 0*Z(7), Z(7)^0 ] ] ])'
```

kernel()

EXAMPLES:

```
sage: F = GF(7); MS = MatrixSpace(F,2,2)
sage: F.multiplicative_generator()
3
sage: G = MatrixGroup([MS([3,0,0,1])])
sage: a = G.gens()[0]^2
sage: phi = G.hom([a])
sage: phi.kernel()
'Group([ [ [ Z(7)^3, 0*Z(7) ], [ 0*Z(7), Z(7)^0 ] ] ])'
sage: phi.image(G.gens()[0])
'[ [ Z(7)^2, 0*Z(7) ], [ 0*Z(7), Z(7)^0 ] ]'
```

22.13 Matrix Group Homsets

AUTHORS:

- William Stein (2006-05-07): initial version

class MatrixGroupHomset(*G, H*)

is_MatrixGroupHomset(*x*)

22.14 Linear Groups

AUTHORS:

- William Stein: initial version
- David Joyner: degree, base_ring, random, order methods; examples

- David Joyner (2006-05): added center, more examples, renamed random attributes, bug fixes.
- William Stein (2006-12): total rewrite

REFERENCES:

- [KL] Peter Kleidman and Martin Liebeck. The subgroup structure of the finite classical groups. Cambridge University Press, 1990.
- [C] R. W. Carter. Simple groups of Lie type, volume 28 of Pure and Applied Mathematics. John Wiley and Sons, 1972.

EXAMPLES:

22.15 General Linear Groups

EXAMPLES:

```
sage: GL(4,QQ)
General Linear Group of degree 4 over Rational Field
sage: GL(1,ZZ)
General Linear Group of degree 1 over Integer Ring
sage: GL(100,RR)
General Linear Group of degree 100 over Real Field with 53 bits of precision
sage: GL(3,GF(49,'a'))
General Linear Group of degree 3 over Finite Field in a of size 7^2
```

AUTHORS:

- David Joyner (2006-01)
- William Stein (2006-01)
- David Joyner (2006-05): added _latex_, __str__, examples
- William Stein (2006-12-09): rewrite

GL(*n, R, var='a'*)

Return the general linear group of degree n over the ring R.

EXAMPLES:

```
sage: G = GL(6,GF(5))
sage: G.order()
11064475422000000000000000
sage: G.base_ring()
Finite Field of size 5
```

```
sage: F = GF(3); MS = MatrixSpace(F,2,2)
sage: gens = [MS([[0,1],[1,0]]),MS([[1,1],[0,1]])]
sage: G = MatrixGroup(gens)
sage: G.order()
48
sage: H = GL(2,F)
sage: H.order()
48
```

```
sage: H == G
True
sage: H.as_matrix_group() == G
True
sage: H.gens()
[
[2 0]
[0 1],
[2 1]
[2 0]
]
```

class GeneralLinearGroup_finite_field(*n, R, var='a'*)

class GeneralLinearGroup_generic(*n, R, var='a'*)

22.16 Special Linear Groups

AUTHORS:

- William Stein: initial version
- David Joyner (2006-05): added examples, _latex_, __str__, gens, as_matrix_group
- William Stein (2006-12-09): rewrite

EXAMPLES:

```
sage: SL(2, ZZ)
Special Linear Group of degree 2 over Integer Ring
sage: G = SL(2,GF(3)); G
Special Linear Group of degree 2 over Finite Field of size 3
sage: G.is_finite()
True
sage: G.conjugacy_class_representatives()
[
[1 0]
[0 1],
[0 2]
[1 1],
[0 1]
[2 1],
[2 0]
[0 2],
[0 2]
[1 2],
[0 1]
[2 2],
[0 2]
[1 0]
]
sage: G = SL(6,GF(5))
sage: G.gens()
[
[2 0 0 0 0 0]
[0 3 0 0 0 0]
```

```
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1],
[4 0 0 0 0 1]
[4 0 0 0 0 0]
[0 4 0 0 0 0]
[0 0 4 0 0 0]
[0 0 0 4 0 0]
[0 0 0 0 4 0]
]
```

SL(*n, R, var='a'*)

Return the special linear group of degree n over the ring R.

EXAMPLES:

```
sage: SL(3,GF(2))
Special Linear Group of degree 3 over Finite Field of size 2
sage: G = SL(15,GF(7)); G
Special Linear Group of degree 15 over Finite Field of size 7
sage: G.order()
19567125956981469620152190624295863411240180071820494789160673696387130667378823633935199663436
sage: len(G.gens())
2
sage: G = SL(2,ZZ); G
Special Linear Group of degree 2 over Integer Ring
sage: G.gens()
[
[ 0  1]
[-1  0],
[1 1]
[0 1]
]
```

Next we compute generators for $\mathrm{SL}_3(\mathbf{Z})$.

```
sage: G = SL(3,ZZ); G
Special Linear Group of degree 3 over Integer Ring
sage: G.gens()
[
[0 1 0]
[0 0 1]
[1 0 0],
[ 0  1  0]
[-1  0  0]
[ 0  0  1],
[1 1 0]
[0 1 0]
[0 0 1]
]
```

class SpecialLinearGroup_finite_field(*n, R, var='a'*)

class SpecialLinearGroup_generic(*n, R, var='a'*)

22.17 Orthogonal Linear Groups

Paraphrased from the GAP manual: The general orthogonal group $GO(e, d, q)$ consists of those $d \times d$ matrices over the field $GF(q)$ that respect a non-singular quadratic form specified by e. (Use the GAP command InvariantQuadraticForm to determine this form explicitly.) The value of e must be 0 for odd d (and can optionally be omitted in this case), respectively one of 1 or -1 for even d.

SpecialOrthogonalGroup returns a group isomorphic to the special orthogonal group $SO(e, d, q)$, which is the subgroup of all those matrices in the general orthogonal group that have determinant one. (The index of $SO(e, d, q)$ in $GO(e, d, q)$ is 2 if q is odd, but $SO(e, d, q) = GO(e, d, q)$ if q is even.)

Warning: GAP notation: GO([e,] d, q), SO([e,] d, q) ([...] denotes and optional value)

Sage notation: GO(d, GF(q), e=0), SO(d, GF(q), e=0)

There is no Python trick I know of to allow the first argument to have the default value e=0 and leave the other two arguments as non-default. This forces us into non-standard notation.

AUTHORS:

- David Joyner (2006-03): initial version
- David Joyner (2006-05): added examples, _latex_, __str__, gens, as_matrix_group
- William Stein (2006-12-09): rewrite

GO(*n, R, e=0*)

Return the general orthogonal group.

EXAMPLES:

class GeneralOrthogonalGroup_finite_field(*n, R, e=0, var='a'*)

class GeneralOrthogonalGroup_generic(*n, R, e=0, var='a'*)

EXAMPLES:

```
sage: GO( 3, GF(7), 0)
General Orthogonal Group of degree 3, form parameter 0, over the Finite Field of size 7
sage: GO( 3, GF(7), 0).order()
672
sage: GO( 3, GF(7), 0).random_element()
[5 1 4]
[1 0 0]
[6 0 1]
```

invariant_quadratic_form()

This wraps GAP's command "InvariantQuadraticForm". From the GAP documentation:

INPUT:

- `self` - a matrix group G

OUTPUT:

- `Q` - the matrix satisfying the property: The quadratic form q on the natural vector space V on which G acts is given by $q(v) = vQv^t$, and the invariance under G is given by the equation $q(v) = q(vM)$ for all $v \in V$ and $M \in G$.

EXAMPLES:

```
sage: G = GO( 4, GF(7), 1)
sage: G.invariant_quadratic_form()
[0 1 0 0]
[0 0 0 0]
[0 0 3 0]
[0 0 0 1]
```

class **OrthogonalGroup**(*n, R, e=0, var='a'*)

invariant_form()

Return the invariant form of this orthogonal group.

TODO: What is the point of this? What does it do? How does it work?

EXAMPLES:

```
sage: G = SO( 4, GF(7), 1)
sage: G.invariant_form()
1
```

SO(*n, R, e=0, var='a'*)

Return the special orthogonal group of degree n over the ring R.

INPUT:

- n - the degree
- R - ring
- e - a parameter for orthogonal groups only depending on the invariant form

EXAMPLES:

```
sage: G = SO(3,GF(5))
sage: G.gens()
[
[2 0 0]
[0 3 0]
[0 0 1],
[3 2 3]
[0 2 0]
[0 3 1],
[1 4 4]
[4 0 0]
[2 0 4]
]
sage: G = SO(3,GF(5))
sage: G.as_matrix_group()
Matrix group over Finite Field of size 5 with 3 generators:
[[[2, 0, 0], [0, 3, 0], [0, 0, 1]], [[3, 2, 3], [0, 2, 0], [0, 3, 1]], [[1, 4, 4], [4, 0, 0], [2
```

class **SpecialOrthogonalGroup_finite_field**(*n, R, e=0, var='a'*)

class **SpecialOrthogonalGroup_generic**(*n, R, e=0, var='a'*)

EXAMPLES:

```
sage: G = SO( 4, GF(7), 1); G
Special Orthogonal Group of degree 4, form parameter 1, over the Finite Field of size 7
sage: G._gap_init_()
'SO(1, 4, 7)'
sage: G.random_element()
[1 2 5 0]
```

```
[2 2 1 0]
[1 3 1 5]
[1 3 1 3]
```

invariant_quadratic_form()

Return the quadratic form $q(v) = vQv^t$ on the space on which this group G that satisfies the equation $q(v) = q(vM)$ for all $v \in V$ and $M \in G$.

Note: Uses GAP's command InvariantQuadraticForm.

OUTPUT:

- Q - matrix that defines the invariant quadratic form.

EXAMPLES:

```
sage: G = SO( 4, GF(7), 1)
sage: G.invariant_quadratic_form()
[0 1 0 0]
[0 0 0 0]
[0 0 3 0]
[0 0 0 1]
```

22.18 Symplectic Linear Groups

AUTHORS:

- David Joyner (2006-03): initial version, modified from special_linear (by W. Stein)

EXAMPLES:

```
sage: G = Sp(4,GF(7))
sage: G._gap_init_()
'Sp(4, 7)'
sage: G
Symplectic Group of rank 2 over Finite Field of size 7
sage: G.random_element()
[1 6 5 5]
[2 1 4 5]
[1 2 4 5]
[4 0 2 2]
sage: G.order()
276595200
```

Sp(*n, R, var='a'*)

Return the symplectic group of degree n over R.

EXAMPLES:

```
sage: Sp(4,5)
Symplectic Group of rank 2 over Finite Field of size 5
sage: Sp(3,GF(7))
...
ValueError: the degree n (=3) must be even
```

class SymplecticGroup_finite_field(*n, R, var='a'*)

class SymplecticGroup_generic(*n, R, var='a'*)

22.19 Unitary Groups $GU(n, q)$ and $SU(n, q)$

These are $n \times n$ unitary matrices with entries in $GF(q^2)$.

AUTHORS:

- David Joyner (2006-03): initial version, modified from special_linear (by W. Stein)
- David Joyner (2006-05): minor additions (examples, _latex_, __str__, gens)
- William Stein (2006-12): rewrite

EXAMPLES:

```
sage: G = SU(3,GF(5))
sage: G.order()
378000
sage: G
Special Unitary Group of degree 3 over Finite Field of size 5
sage: G._gap_init_()
'SU(3, 5)'
sage: G.random_element()
[      1 4*a + 4 4*a + 1]
[2*a + 4 2*a + 1       0]
[      4     3*a 4*a + 2]
sage: G.base_ring()
Finite Field of size 5
sage: G.field_of_definition()
Finite Field in a of size 5^2
```

GU(*n, F, var='a'*)

Return the general unitary group of degree n over the finite field F.

INPUT:

- `n` - a positive integer
- `F` - finite field
- `var` - variable used to represent generator of quadratic extension of F, if needed.

EXAMPLES:

```
sage: G = GU(3,GF(7)); G
General Unitary Group of degree 3 over Finite Field of size 7
sage: G.gens()
[
[  a   0   0]
[  0   1   0]
[  0   0 5*a],
[6*a   6   1]
[  6   6   0]
[  1   0   0]
]
sage: G = GU(2,QQ)
...
NotImplementedError: general unitary group only implemented over finite fields
```

```
sage: G = GU(3,GF(5), var='beta')
sage: G.gens()
[
[  beta       0       0]
[     0       1       0]
[     0       0 3*beta],
[4*beta       4       1]
[     4       4       0]
[     1       0       0]
]
```

class GeneralUnitaryGroup_finite_field(*n, R, var='a'*)

SU(*n, F, var='a'*)

Return the special unitary group of degree n over F.

EXAMPLES:

```
sage: SU(3,5)
Special Unitary Group of degree 3 over Finite Field of size 5
sage: SU(3,QQ)
...
NotImplementedError: special unitary group only implemented over finite fields
```

class SpecialUnitaryGroup_finite_field(*n, R, var='a'*)

class UnitaryGroup_finite_field(*n, R, var='a'*)

field_of_definition()

Return the field of definition of this general unity group.

EXAMPLES:

```
sage: G = GU(3,GF(5))
sage: G.field_of_definition()
Finite Field in a of size 5^2
sage: G.base_field()
Finite Field of size 5
```

CHAPTER

TWENTYTHREE

GENERAL RINGS, IDEALS, AND MORPHISMS

23.1 Ideals

Sage provides functionality for computing with ideals. One can create an ideal in any commutative ring R by giving a list of generators, using the notation `R.ideal([a,b,...])`.

Cyclic(*R, n=None, homog=False, singular=Singular*)

Ideal of cyclic n-roots from 1-st n variables of R if R is coercable to Singular. If n==None n is set to R.ngens()

INPUT:

- `R` - base ring to construct ideal for
- `n` - number of cyclic roots (default: None)
- `homog` - if True a homogenous ideal is returned using the last variable in the ideal (default: False)
- `singular` - singular instance to use

Note: R will be set as the active ring in Singular

EXAMPLES:

An example from a multivariate polynomial ring over the rationals:

```
sage: P.<x,y,z> = PolynomialRing(QQ,3,order='lex')
sage: I = sage.rings.ideal.Cyclic(P)
sage: I
Ideal (x + y + z, x*y + x*z + y*z, x*y*z - 1) of Multivariate Polynomial
Ring in x, y, z over Rational Field
sage: I.groebner_basis()
[x + y + z, y^2 + y*z + z^2, z^3 - 1]
```

We compute a Groebner basis for cyclic 6, which is a standard benchmark and test ideal:

```
sage: R.<x,y,z,t,u,v> = QQ['x,y,z,t,u,v']
sage: I = sage.rings.ideal.Cyclic(R,6)
sage: B = I.groebner_basis()
sage: len(B)
45
```

FieldIdeal(*R*)

Let q = R.base_ring().order() and (x0,...,x_n) = R.gens() then if q is finite this constructor returns $< x_0^q - x_0, ..., x_n^q - x_n >$.

We call this ideal the field ideal and the generators the field equations.

EXAMPLES:

The Field Ideal generated from the polynomial ring over two variables in the finite field of size 2:

```
sage: P.<x,y> = PolynomialRing(GF(2),2)
sage: I = sage.rings.ideal.FieldIdeal(P); I
Ideal (x^2 + x, y^2 + y) of Multivariate Polynomial Ring in x, y over
Finite Field of size 2
```

Another, similar example:

```
sage: Q.<x1,x2,x3,x4> = PolynomialRing(GF(2^4,name='alpha'), 4)
sage: J = sage.rings.ideal.FieldIdeal(Q); J
Ideal (x1^16 + x1, x2^16 + x2, x3^16 + x3, x4^16 + x4) of
Multivariate Polynomial Ring in x1, x2, x3, x4 over Finite
Field in alpha of size 2^4
```

Ideal (**args, **kwds*)

Create the ideal in ring with given generators.

There are some shorthand notations for creating an ideal, in addition to using the Ideal function:

```
--  R.ideal(gens, coerce=True)
--  gens*R
--  R*gens
```

INPUT:

- `R (optional)` - a ring (if not given, will try to infer it from gens)
- `gens` - list of elements generating the ideal
- `coerce (optional)` - bool (default: True); whether gens need to be coerced into ring.

OUTPUT: The ideal of ring generated by gens.

EXAMPLES:

```
sage: R, x = PolynomialRing(ZZ, 'x').objgen()
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: I
Ideal (x^2 + 3*x + 4, x^2 + 1) of Univariate Polynomial Ring in x over Integer Ring
sage: Ideal(R, [4 + 3*x + x^2, 1 + x^2])
Ideal (x^2 + 3*x + 4, x^2 + 1) of Univariate Polynomial Ring in x over Integer Ring
sage: Ideal((4 + 3*x + x^2, 1 + x^2))
Ideal (x^2 + 3*x + 4, x^2 + 1) of Univariate Polynomial Ring in x over Integer Ring


sage: ideal(x^2-2*x+1, x^2-1)
Ideal (x^2 - 2*x + 1, x^2 - 1) of Univariate Polynomial Ring in x over Integer Ring
sage: ideal([x^2-2*x+1, x^2-1])
Ideal (x^2 - 2*x + 1, x^2 - 1) of Univariate Polynomial Ring in x over Integer Ring
sage: l = [x^2-2*x+1, x^2-1]
sage: ideal(f^2 for f in l)
Ideal (x^4 - 4*x^3 + 6*x^2 - 4*x + 1, x^4 - 2*x^2 + 1) of
Univariate Polynomial Ring in x over Integer Ring
```

This example illustrates how Sage finds a common ambient ring for the ideal, even though 1 is in the integers (in this case).

```
sage: R.<t> = ZZ['t']
sage: i = ideal(1,t,t^2)
sage: i
Ideal (1, t, t^2) of Univariate Polynomial Ring in t over Integer Ring
sage: ideal(1/2,t,t^2)
Principal ideal (1) of Univariate Polynomial Ring in t over Rational Field
```

This shows that the issues at trac #1104 are resolved:

```
sage: Ideal(3, 5)
Principal ideal (1) of Integer Ring
sage: Ideal(ZZ, 3, 5)
Principal ideal (1) of Integer Ring
sage: Ideal(2, 4, 6)
Principal ideal (2) of Integer Ring
```

TESTS:

```
sage: R, x = PolynomialRing(ZZ, 'x').objgen()
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: I == loads(dumps(I))
True


sage: I = Ideal(R, [4 + 3*x + x^2, 1 + x^2])
sage: I == loads(dumps(I))
True


sage: I = Ideal((4 + 3*x + x^2, 1 + x^2))
sage: I == loads(dumps(I))
True
```

class **Ideal_fractional**(*ring, gens, coerce=True*)

class **Ideal_generic**(*ring, gens, coerce=True*)

An ideal.

apply_morphism(*phi*)

Apply the morphism phi to every element of this ideal. Returns an ideal in the domain of phi.

EXAMPLES: sage: psi = CC['x'].hom([-CC['x'].0]) sage: J = ideal([CC['x'].0 + 1]); J Principal ideal (1.00000000000000*x + 1.00000000000000) of Univariate Polynomial Ring in x over Complex Field with 53 bits of precision sage: psi(J) Principal ideal (-1.00000000000000*x + 1.00000000000000) of Univariate Polynomial Ring in x over Complex Field with 53 bits of precision sage: J.apply_morphism(psi) Principal ideal (-1.00000000000000*x + 1.00000000000000) of Univariate Polynomial Ring in x over Complex Field with 53 bits of precision
sage: psi = ZZ['x'].hom([-ZZ['x'].0]) sage: J = ideal([ZZ['x'].0, 2]); J Ideal (x, 2) of Univariate Polynomial Ring in x over Integer Ring sage: psi(J) Ideal (-x, 2) of Univariate Polynomial Ring in x over Integer Ring sage: J.apply_morphism(psi) Ideal (-x, 2) of Univariate Polynomial Ring in x over Integer Ring

TESTS: sage: K.<a> = NumberField(x^2 + 1) sage: A = K.ideal(a) sage: taus = K.embeddings(K) sage: A.apply_morphism(taus[0]) # identity Fractional ideal (a) sage: A.apply_morphism(taus[1]) # complex conjugation Fractional ideal (-a) sage: A.apply_morphism(taus[0]) == A.apply_morphism(taus[1]) True
sage: K.<a> = NumberField(x^2 + 5) sage: B = K.ideal([2, a + 1]); B Fractional ideal (2, a + 1) sage: taus = K.embeddings(K) sage: B.apply_morphism(taus[0]) # identity Fractional ideal (2, a + 1)
Since 2 is totally ramified, complex conjugation fixes it:
sage: B.apply_morphism(taus[1]) # complex conjugation Fractional ideal (2, a + 1)

sage: taus[1](B) Fractional ideal (2, a + 1)

base_ring()

Returns the base ring of this ideal.

EXAMPLES:

```
sage: R = ZZ
sage: I = 3*R; I
Principal ideal (3) of Integer Ring
sage: J = 2*I; J
Principal ideal (6) of Integer Ring
sage: I.base_ring(); J.base_ring()
Integer Ring
Integer Ring
```

We construct an example of an ideal of a quotient ring:

```
sage: R = PolynomialRing(QQ, 'x'); x = R.gen()
sage: I = R.ideal(x^2 - 2)
sage: I.base_ring()
Rational Field
```

And p-adic numbers:

```
sage: R = Zp(7, prec=10); R
7-adic Ring with capped relative precision 10
sage: I = 7*R; I
Principal ideal (7 + O(7^11)) of 7-adic Ring with capped relative precision 10
sage: I.base_ring()
7-adic Ring with capped relative precision 10
```

category()

Return the category of this ideal.

EXAMPLES:

Note that category is dependent on the ring of the ideal.

```
sage: I = ZZ.ideal(7)
sage: J = ZZ[x].ideal(7,x)
sage: K = ZZ[x].ideal(7)
sage: I.category()
Category of ring ideals in Integer Ring
sage: J.category()
Category of ring ideals in Univariate Polynomial Ring in x
over Integer Ring
sage: K.category()
Category of ring ideals in Univariate Polynomial Ring in x
over Integer Ring
```

gens()

Return a set of generators / a basis of self. This is usually the set of generators provided during object creation.

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQ,2)
sage: I = Ideal([x,y+1]); I
Ideal (x, y + 1) of Multivariate Polynomial Ring in x, y over Rational Field
sage: I.gens()
(x, y + 1)
```

```
sage: ZZ.ideal(5,10).gens()
(5,)
```

gens_reduced()

Same as gens() for this ideal, since there is currently no special gens_reduced algorithm implemented for this ring.

This method is provided so that ideals in ZZ have the method gens_reduced(), just like ideals of number fields.

EXAMPLES:

```
sage: ZZ.ideal(5).gens_reduced()
(5,)
```

is_maximal()

Returns True if the ideal is maximal in the ring containing the ideal.

TODO: Make self.is_maximal() work! Write this code!

EXAMPLES:

```
sage: R = ZZ
sage: I = R.ideal(7)
sage: I.is_maximal()
...
NotImplementedError
```

is_prime()

Returns True if the ideal is prime in the ring containing the ideal.

TODO: Make self.is_prime() work! Write this code!

EXAMPLES:

```
sage: R = ZZ[x]
sage: I = R.ideal(7)
sage: I.is_prime()
...
NotImplementedError
```

is_principal()

Returns True if the ideal is principal in the ring containing the ideal.

TODO: Code is naive. Only keeps track of ideal generators as set during intiialization of the ideal. (Can the base ring change? See example below.)

EXAMPLES:

```
sage: R = ZZ[x]
sage: I = R.ideal(2,x)
sage: I.is_principal()
...
NotImplementedError
sage: J = R.base_extend(QQ).ideal(2,x)
sage: J.is_principal()
True
```

is_trivial()

Return True if this ideal is (0) or (1).

TESTS:

```
sage: I = ZZ.ideal(5)
sage: I.is_trivial()
False
```

```
sage: I = ZZ['x'].ideal(-1)
sage: I.is_trivial()
True

sage: I = ZZ['x'].ideal(ZZ['x'].gen()^2)
sage: I.is_trivial()
False

sage: I = QQ['x', 'y'].ideal(-5)
sage: I.is_trivial()
True

sage: I = CC['x'].ideal(0)
sage: I.is_trivial()
True
```

reduce(*f*)

Return the reduction of the element of f modulo the ideal I (=self). This is an element of R that is equivalent modulo I to f.

EXAMPLES:

```
sage: ZZ.ideal(5).reduce(17)
2
sage: parent(ZZ.ideal(5).reduce(17))
Integer Ring
```

ring()

Returns the ring containing this ideal.

EXAMPLES:

```
sage: R = ZZ
sage: I = 3*R; I
Principal ideal (3) of Integer Ring
sage: J = 2*I; J
Principal ideal (6) of Integer Ring
sage: I.ring(); J.ring()
Integer Ring
Integer Ring
```

Note that `self.ring()` is different from `self.base_ring()`

```
sage: R = PolynomialRing(QQ, 'x'); x = R.gen()
sage: I = R.ideal(x^2 - 2)
sage: I.base_ring()
Rational Field
sage: I.ring()
Univariate Polynomial Ring in x over Rational Field
```

Another example using polynomial rings:

```
sage: R = PolynomialRing(QQ, 'x'); x = R.gen()
sage: I = R.ideal(x^2 - 3)
sage: I.ring()
Univariate Polynomial Ring in x over Rational Field
sage: Rbar = R.quotient(I, names='a')
sage: S = PolynomialRing(Rbar, 'y'); y = Rbar.gen(); S
Univariate Polynomial Ring in y over Univariate Quotient Polynomial Ring in a over Rational
sage: J = S.ideal(y^2 + 1)
sage: J.ring()
Univariate Polynomial Ring in y over Univariate Quotient Polynomial Ring in a over Rational
```

class **Ideal_pid**(*ring, gen*)

An ideal of a principal ideal domain.

gcd(*other*)

Returns the greatest common divisor of the principal ideal with the ideal `other`; that is, the largest principal ideal contained in both the ideal and `other`

TODO: This is not implemented in the case when `other` is neither principal nor when the generator of `self` is contained in `other`. Also, it seems that this class is used only in PIDs–is this redundant? Note: second example is broken.

EXAMPLES:

An example in the principal ideal domain ZZ:

```
sage: R = ZZ
sage: I = R.ideal(42)
sage: J = R.ideal(70)
sage: I.gcd(J)
Principal ideal (14) of Integer Ring
sage: J.gcd(I)
Principal ideal (14) of Integer Ring
```

TESTS:

We cannot take the gcd of a principal ideal with a non-principal ideal as well: (gcd(I,J) should be (7))

```
sage: I = ZZ.ideal(7)
sage: J = ZZ[x].ideal(7,x)
sage: I.gcd(J)
...
NotImplementedError
sage: J.gcd(I)
...
AttributeError: 'Ideal_generic' object has no attribute 'gcd'
```

Note:

```
sage: type(I)
<class 'sage.rings.ideal.Ideal_pid'>
sage: type(J)
<class 'sage.rings.ideal.Ideal_generic'>
```

is_prime()

Returns True if the ideal is prime. This relies on the ring elements having a method is_irreducible() implemented, since an ideal (a) is prime iff a is irreducible (or 0)

EXAMPLES:

```
sage: ZZ.ideal(2).is_prime()
True
sage: ZZ.ideal(-2).is_prime()
True
sage: ZZ.ideal(4).is_prime()
False
sage: ZZ.ideal(0).is_prime()
True
sage: R.<x>=QQ[]
sage: P=R.ideal(x^2+1); P
Principal ideal (x^2 + 1) of Univariate Polynomial Ring in x over Rational Field
sage: P.is_prime()
True
```

reduce(*f*)

Return the reduction of f modulo self.

EXAMPLES:

```
sage: I = 8*ZZ
sage: I.reduce(10)
2
sage: n = 10; n.mod(I)
2
```

residue_field()

Return the residue class field of this ideal, which must be prime.

TODO: Implement this for more general rings. Currently only defined for ZZ and for number field orders.

EXAMPLES:

```
sage: P = ZZ.ideal(61); P
Principal ideal (61) of Integer Ring
sage: F = P.residue_field(); F
Residue field of Integers modulo 61
sage: pi = F.reduction_map(); pi
Partially defined reduction map from Rational Field to Residue field of Integers modulo 61
sage: pi(123/234)
6
sage: pi(1/61)
...
ZeroDivisionError: Cannot reduce rational 1/61 modulo 61: it has negative valuation
sage: lift = F.lift_map(); lift
Lifting map from Residue field of Integers modulo 61 to Rational Field
sage: lift(F(12345/67890))
33
sage: (12345/67890) % 61
33
```

TESTS:

```
sage: ZZ.ideal(96).residue_field()
...
ValueError: The ideal (Principal ideal (96) of Integer Ring) is not prime

sage: R.<x>=QQ[]
sage: I=R.ideal(x^2+1)
sage: I.is_prime()
True
sage: I.residue_field()
Traceback (most recent call last):
NotImplementedError: residue_field() is only implemented for ZZ and rings of integers of num
```

class **Ideal_principal**(*ring, gen*)

A principal ideal.

divides(*other*)

Returns True if self divides other.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(QQ)
sage: I = P.ideal(x)
sage: J = P.ideal(x^2)
sage: I.divides(J)
True
sage: J.divides(I)
False
```

gen()

Returns the generator of the principal ideal. The generators are elements of the ring containing the ideal.

EXAMPLES:

A simple example in the integers:

```
sage: R = ZZ
sage: I = R.ideal(7)
sage: J = R.ideal(7, 14)
sage: I.gen(); J.gen()
7
7
```

Note that the generator belongs to the ring from which the ideal was initialized:

```
sage: R = ZZ[x]
sage: I = R.ideal(x)
sage: J = R.base_extend(QQ).ideal(2,x)
sage: a = I.gen(); a
x
sage: b = J.gen(); b
1
sage: a.base_ring()
Integer Ring
sage: b.base_ring()
Rational Field
```

is_principal()

Returns True if the ideal is principal in the ring containing the ideal. When the ideal construction is explicitly principal (i.e. when we define an ideal with one element) this is always the case.

EXAMPLES:

Note that Sage automatically coerces ideals into principal ideals during initialization:

```
sage: R = ZZ[x]
sage: I = R.ideal(x)
sage: J = R.ideal(2,x)
sage: K = R.base_extend(QQ).ideal(2,x)
sage: I
Principal ideal (x) of Univariate Polynomial Ring in x
over Integer Ring
sage: J
Ideal (2, x) of Univariate Polynomial Ring in x over Integer Ring
sage: K
Principal ideal (1) of Univariate Polynomial Ring in x
over Rational Field
sage: I.is_principal()
True
sage: K.is_principal()
True
```

Katsura(*R, n=None, homog=False, singular=Singular*)

n-th katsura ideal of R if R is coercable to Singular. If n==None n is set to R.ngens()

INPUT:

- `R` - base ring to construct ideal for
- `n` - which katsura ideal of R
- `homog` - if True a homogenous ideal is returned using the last variable in the ideal (default: False)
- `singular` - singular instance to use

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ,3)
sage: I = sage.rings.ideal.Katsura(P,3); I
Ideal (x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y)
of Multivariate Polynomial Ring in x, y, z over Rational Field

sage: Q.<x> = PolynomialRing(QQ,1)
sage: J = sage.rings.ideal.Katsura(Q,1); J
Ideal (x - 1) of Multivariate Polynomial Ring in x over Rational Field
```

is_Ideal(*x*)

Returns True if object is an ideal of a ring.

EXAMPLES:

A simple example involving the ring of integers. Note that Sage does not interpret rings objects themselves as ideals. However, one can still explicitly construct these ideals:

```
sage: from sage.rings.ideal import is_Ideal
sage: R = ZZ
sage: is_Ideal(R)
False
sage: 1*R; is_Ideal(1*R)
Principal ideal (1) of Integer Ring
True
sage: 0*R; is_Ideal(0*R)
Principal ideal (0) of Integer Ring
True
```

Sage recognizes ideals of polynomial rings as well:

```
sage: R = PolynomialRing(QQ, 'x'); x = R.gen()
sage: I = R.ideal(x^2 + 1); I
Principal ideal (x^2 + 1) of Univariate Polynomial Ring in x over Rational Field
sage: is_Ideal(I)
True
sage: is_Ideal((x^2 + 1)*R)
True
```

23.2 Monoid of Ring Ideals

IdealMonoid(*R*)

class IdealMonoid_c(*R*)

ring()

23.3 Homomorphisms of rings

We give a large number of examples of ring homomorphisms.

EXAMPLE: Natural inclusion $\mathbf{Z} \hookrightarrow \mathbf{Q}$.

```
sage: H = Hom(ZZ, QQ)
sage: phi = H([1])
sage: phi(10)
10
sage: phi(3/1)
3
sage: phi(2/3)
...
TypeError: 2/3 must be coercible into Integer Ring
```

There is no homomorphism in the other direction:

```
sage: H = Hom(QQ, ZZ)
sage: H([1])
...
TypeError: images do not define a valid homomorphism
```

EXAMPLE: Reduction to finite field.

```
sage: H = Hom(ZZ, GF(9, 'a'))
sage: phi = H([1])
sage: phi(5)
2
sage: psi = H([4])
sage: psi(5)
2
```

EXAMPLE: Map from single variable polynomial ring.

```
sage: R, x = PolynomialRing(ZZ, 'x').objgen()
sage: phi = R.hom([2], GF(5))
sage: phi
Ring morphism:
  From: Univariate Polynomial Ring in x over Integer Ring
  To:   Finite Field of size 5
  Defn: x |--> 2
sage: phi(x + 12)
4
```

EXAMPLE: Identity map on the real numbers.

```
sage: f = RR.hom([RR(1)]); f
Ring endomorphism of Real Field with 53 bits of precision
  Defn: 1.00000000000000 |--> 1.00000000000000
sage: f(2.5)
2.50000000000000
sage: f = RR.hom( [2.0] )
...
TypeError: images do not define a valid homomorphism
```

EXAMPLE: Homomorphism from one precision of field to another.

From smaller to bigger doesn't make sense:

```
sage: R200 = RealField(200)
sage: f = RR.hom( R200 )
```

```
...
TypeError: Natural coercion morphism from Real Field with 53 bits of precision to Real Field with 200
```

From bigger to small does:

```
sage: f = RR.hom( RealField(15) )
sage: f(2.5)
2.500
sage: f(RR.pi())
3.142
```

EXAMPLE: Inclusion map from the reals to the complexes:

```
sage: i = RR.hom([CC(1)]); i
Ring morphism:
  From: Real Field with 53 bits of precision
  To:   Complex Field with 53 bits of precision
  Defn: 1.00000000000000 |--> 1.00000000000000
sage: i(RR('3.1'))
3.10000000000000
```

EXAMPLE: A map from a multivariate polynomial ring to itself:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: phi = R.hom([y,z,x^2]); phi
Ring endomorphism of Multivariate Polynomial Ring in x, y, z over Rational Field
  Defn: x |--> y
        y |--> z
        z |--> x^2
sage: phi(x+y+z)
x^2 + y + z
```

EXAMPLE: An endomorphism of a quotient of a multi-variate polynomial ring:

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: S.<a,b> = quo(R, ideal(1 + y^2))
sage: phi = S.hom([a^2, -b])
sage: phi
Ring endomorphism of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the idea
  Defn: a |--> a^2
        b |--> -b
sage: phi(b)
-b
sage: phi(a^2 + b^2)
a^4 - 1
```

EXAMPLE: The reduction map from the integers to the integers modulo 8, viewed as a quotient ring:

```
sage: R = ZZ.quo(8*ZZ)
sage: pi = R.cover()
sage: pi
Ring morphism:
  From: Integer Ring
  To:   Ring of integers modulo 8
  Defn: Natural quotient map
sage: pi.domain()
Integer Ring
```

```
sage: pi.codomain()
Ring of integers modulo 8
sage: pi(10)
2
sage: pi.lift()
Set-theoretic ring morphism:
  From: Ring of integers modulo 8
  To:   Integer Ring
  Defn: Choice of lifting map
sage: pi.lift(13)
5
```

EXAMPLE: Inclusion of GF(2) into GF(4,'a').

```
sage: k = GF(2)
sage: i = k.hom(GF(4, 'a'))
sage: i
Ring Coercion morphism:
  From: Finite Field of size 2
  To:   Finite Field in a of size 2^2
sage: i(0)
0
sage: a = i(1); a.parent()
Finite Field in a of size 2^2
```

We next compose the inclusion with reduction from the integers to GF(2).

```
sage: pi = ZZ.hom(k)
sage: pi
Ring Coercion morphism:
  From: Integer Ring
  To:   Finite Field of size 2
sage: f = i * pi
sage: f
Composite map:
  From: Integer Ring
  To:   Finite Field in a of size 2^2
  Defn:   Ring Coercion morphism:
          From: Integer Ring
          To:   Finite Field of size 2
        then
          Ring Coercion morphism:
          From: Finite Field of size 2
          To:   Finite Field in a of size 2^2
sage: a = f(5); a
1
sage: a.parent()
Finite Field in a of size 2^2
```

EXAMPLE: Inclusion from $\mathbf{Q}$ to the 3-adic field.

```
sage: phi = QQ.hom(Qp(3, print_mode = 'series'))
sage: phi
Ring Coercion morphism:
  From: Rational Field
  To:   3-adic Field with capped relative precision 20
sage: phi.codomain()
```

```
3-adic Field with capped relative precision 20
sage: phi(394)
1 + 2*3 + 3^2 + 2*3^3 + 3^4 + 3^5 + O(3^20)
```

EXAMPLE: An automorphism of a quotient of a univariate polynomial ring.

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<sqrt2> = R.quo(x^2-2)
sage: sqrt2^2
2
sage: (3+sqrt2)^10
993054*sqrt2 + 1404491
sage: c = S.hom([-sqrt2])
sage: c(1+sqrt2)
-sqrt2 + 1
```

Note that Sage verifies that the morphism is valid:

```
sage: (1 - sqrt2)^2
-2*sqrt2 + 3
sage: c = S.hom([1-sqrt2])    # this is not valid
...
TypeError: images do not define a valid homomorphism
```

EXAMPLE: Endomorphism of power series ring.

```
sage: R.<t> = PowerSeriesRing(QQ); R
Power Series Ring in t over Rational Field
sage: f = R.hom([t^2]); f
Ring endomorphism of Power Series Ring in t over Rational Field
  Defn: t |--> t^2
sage: R.set_default_prec(10)
sage: s = 1/(1 + t); s
1 - t + t^2 - t^3 + t^4 - t^5 + t^6 - t^7 + t^8 - t^9 + O(t^10)
sage: f(s)
1 - t^2 + t^4 - t^6 + t^8 - t^10 + t^12 - t^14 + t^16 - t^18 + O(t^20)
```

EXAMPLE: Frobenious on a power series ring over a finite field.

```
sage: R.<t> = PowerSeriesRing(GF(5))
sage: f = R.hom([t^5]); f
Ring endomorphism of Power Series Ring in t over Finite Field of size 5
  Defn: t |--> t^5
sage: a = 2 + t + 3*t^2 + 4*t^3 + O(t^4)
sage: b = 1 + t + 2*t^2 + t^3 + O(t^5)
sage: f(a)
2 + t^5 + 3*t^10 + 4*t^15 + O(t^20)
sage: f(b)
1 + t^5 + 2*t^10 + t^15 + O(t^25)
sage: f(a*b)
2 + 3*t^5 + 3*t^10 + t^15 + O(t^20)
sage: f(a)*f(b)
2 + 3*t^5 + 3*t^10 + t^15 + O(t^20)
```

EXAMPLE: Homomorphism of Laurent series ring.

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = R.hom([t^3 + t]); f
Ring endomorphism of Laurent Series Ring in t over Rational Field
  Defn: t |--> t + t^3
sage: R.set_default_prec(10)
sage: s = 2/t^2 + 1/(1 + t); s
2*t^-2 + 1 - t + t^2 - t^3 + t^4 - t^5 + t^6 - t^7 + t^8 - t^9 + O(t^10)
sage: f(s)
2*t^-2 - 3 - t + 7*t^2 - 2*t^3 - 5*t^4 - 4*t^5 + 16*t^6 - 9*t^7 + O(t^8)
sage: f = R.hom([t^3]); f
Ring endomorphism of Laurent Series Ring in t over Rational Field
  Defn: t |--> t^3
sage: f(s)
2*t^-6 + 1 - t^3 + t^6 - t^9 + t^12 - t^15 + t^18 - t^21 + t^24 - t^27
sage: s = 2/t^2 + 1/(1 + t); s
2*t^-2 + 1 - t + t^2 - t^3 + t^4 - t^5 + t^6 - t^7 + t^8 - t^9 + O(t^10)
sage: f(s)
2*t^-6 + 1 - t^3 + t^6 - t^9 + t^12 - t^15 + t^18 - t^21 + t^24 - t^27
```

Note that the homomorphism must result in a converging Laurent series, so the valuation of the image of the generator must be positive:

```
sage: R.hom([1/t])
...
TypeError: images do not define a valid homomorphism
sage: R.hom([1])
...
TypeError: images do not define a valid homomorphism
```

EXAMPLE: Complex conjugation on cyclotomic fields.

```
sage: K.<zeta7> = CyclotomicField(7)
sage: c = K.hom([1/zeta7]); c
Ring endomorphism of Cyclotomic Field of order 7 and degree 6
  Defn: zeta7 |--> -zeta7^5 - zeta7^4 - zeta7^3 - zeta7^2 - zeta7 - 1
sage: a = (1+zeta7)^5; a
zeta7^5 + 5*zeta7^4 + 10*zeta7^3 + 10*zeta7^2 + 5*zeta7 + 1
sage: c(a)
5*zeta7^5 + 5*zeta7^4 - 4*zeta7^2 - 5*zeta7 - 4
sage: c(zeta7 + 1/zeta7)       # this element is obviously fixed by inversion
-zeta7^5 - zeta7^4 - zeta7^3 - zeta7^2 - 1
sage: zeta7 + 1/zeta7
-zeta7^5 - zeta7^4 - zeta7^3 - zeta7^2 - 1
```

EXAMPLE: Embedding a number field into the reals.

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<beta> = NumberField(x^3 - 2)
sage: alpha = RR(2)^(1/3); alpha
1.25992104989487
sage: i = K.hom([alpha],check=False); i
Ring morphism:
  From: Number Field in beta with defining polynomial x^3 - 2
  To:   Real Field with 53 bits of precision
  Defn: beta |--> 1.25992104989487
sage: i(beta)
```

```
1.25992104989487
sage: i(beta^3)
2.00000000000000
sage: i(beta^2 + 1)
2.58740105196820
```

An example from Jim Carlson:

```
sage: K = QQ # by the way :-)
sage: R.<a,b,c,d> = K[]; R
Multivariate Polynomial Ring in a, b, c, d over Rational Field
sage: S.<u> = K[]; S
Univariate Polynomial Ring in u over Rational Field
sage: f = R.hom([0,0,0,u], S); f
Ring morphism:
  From: Multivariate Polynomial Ring in a, b, c, d over Rational Field
  To:   Univariate Polynomial Ring in u over Rational Field
  Defn: a |--> 0
        b |--> 0
        c |--> 0
        d |--> u
sage: f(a+b+c+d)
u
sage: f( (a+b+c+d)^2 )
u^2
```

TESTS:

```
sage: H = Hom(ZZ, QQ)
sage: H == loads(dumps(H))
True


sage: K.<zeta7> = CyclotomicField(7)
sage: c = K.hom([1/zeta7])
sage: c == loads(dumps(c))
True


sage: R.<t> = PowerSeriesRing(GF(5))
sage: f = R.hom([t^5])
sage: f == loads(dumps(f))
True
```

class RingHomomorphism()

Homomorphism of rings.

inverse_image()

Return the inverse image of the ideal I under this ring homomorphism.

EXAMPLES:

This is not implemented in any generality yet:

```
sage: f = ZZ.hom(ZZ)
sage: f.inverse_image(ZZ.ideal(2))
...
NotImplementedError
```

is_injective()

Return whether or not this morphism is injective, or raise a NotImplementedError.

EXAMPLES:

Note that currently this is not implemented in most interesting cases:

```
sage: f = ZZ.hom(QQ)
sage: f.is_injective()
...
NotImplementedError
```

is_zero()

Return True if this is the zero map and False otherwise.

A *ring* homomorphism is considered to be 0 if and only if it sends the 1 element of the domain to the 0 element of the codomain. Since rings in Sage all have a 1 element, the zero homomorphism is only to a ring of order 1, where 1==0, e.g., the ring `Integers(1)`.

EXAMPLES:

First an example of a map that is obviously nonzero.

```
sage: h = Hom(ZZ, QQ)
sage: f = h.natural_map()
sage: f.is_zero()
False
```

Next we make the zero ring as $\mathbf{Z}/1\mathbf{Z}$.

```
sage: R = Integers(1)
sage: R
Ring of integers modulo 1
sage: h = Hom(ZZ, R)
sage: f = h.natural_map()
sage: f.is_zero()
True
```

Finally we check an example in characteristic 2.

```
sage: h = Hom(ZZ, GF(2))
sage: f = h.natural_map()
sage: f.is_zero()
False
```

lift()

Return a lifting homomorphism associated to this homomorphism, if it has been defined.

If x is not None, return the value of the lift morphism on x.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: f = R.hom([x,x])
sage: f(x+y)
2*x
sage: f.lift()
...
ValueError: no lift map defined
sage: g = R.hom(R)
sage: f._set_lift(g)
sage: f.lift() == g
True
sage: f.lift(x)
x
```

pushforward()

Returns the pushforward of the ideal I under this ring homomorphism.

EXAMPLES:

```
sage: R.<x,y> = QQ[]; S.<xx,yy> = R.quo([x^2,y^2]);  f = S.cover()
sage: f.pushforward(R.ideal([x,3*x+x*y+y^2]))
Ideal (xx, xx*yy + 3*xx) of Quotient of Multivariate Polynomial Ring in x, y over Rational F
```

class RingHomomorphism_coercion()

class RingHomomorphism_cover()

A homomorphism induced by quotienting a ring out by an ideal.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S.<a,b> = R.quo(x^2 + y^2)
sage: phi = S.cover(); phi
Ring morphism:
  From: Multivariate Polynomial Ring in x, y over Rational Field
  To:   Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 +
  Defn: Natural quotient map
sage: phi(x+y)
a + b
```

kernel()

Return the kernel of this covering morphism, which is the ideal that was quotiented out by.

EXAMPLES:

```
sage: f = Zmod(6).cover()
sage: f.kernel()
Principal ideal (6) of Integer Ring
```

class RingHomomorphism_from_quotient()

A ring homomorphism with domain a generic quotient ring.

INPUT:

- `parent` - a ring homset Hom(R,S)
- `phi` - a ring homomorphism C –> S, where C is the domain of R.cover()

OUTPUT: a ring homomorphism

The domain R is a quotient object $C \to R$, and `R.cover()` is the ring homomorphism $\varphi : C \to R$. The condition on the elements `im_gens` of S is that they define a homomorphism $C \to S$ such that each generator of the kernel of φ maps to 0.

EXAMPLES:

```
sage: R.<x, y, z> = PolynomialRing(QQ, 3)
sage: S.<a, b, c> = R.quo(x^3 + y^3 + z^3)
sage: phi = S.hom([b, c, a]); phi
Ring endomorphism of Quotient of Multivariate Polynomial Ring in x, y, z over Rational Field by
  Defn: a |--> b
        b |--> c
        c |--> a
sage: phi(a+b+c)
a + b + c
sage: loads(dumps(phi)) == phi
True
```

Validity of the homomorphism is determined, when possible, and a TypeError is raised if there is no homomorphism sending the generators to the given images.

```
sage: S.hom([b^2, c^2, a^2])
...
TypeError: images do not define a valid homomorphism
```

morphism_from_cover()

Underlying morphism used to define this quotient map, i.e., the morphism from the cover of the domain.

EXAMPLES:

```
sage: R.<x,y> = QQ[]; S.<xx,yy> = R.quo([x^2,y^2])
sage: S.hom([yy,xx]).morphism_from_cover()
Ring morphism:
  From: Multivariate Polynomial Ring in x, y over Rational Field
  To:   Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x
  Defn: x |--> yy
        y |--> xx
```

class RingHomomorphism_im_gens()

A ring homomorphism determined by the images of generators.

im_gens()

Return the images of the generators of the domain.

OUTPUT: • `list` – a copy of the list of gens (it is safe to change this)

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: f = R.hom([x,x+y])
sage: f.im_gens()
[x, x + y]
```

We verify that the returned list of images of gens is a copy, so changing it doesn't change f:

```
sage: f.im_gens()[0] = 5
sage: f.im_gens()
[x, x + y]
```

class RingMap()

Set-theoretic map between rings.

class RingMap_lift()

Given rings R and S such that for any $x \in R$ the function `x.lift()` is an element that naturally coerces to S, this returns the set-theoretic ring map $R \to S$ sending x to `x.lift()`.

EXAMPLES:

```
sage: R, (x,y) = PolynomialRing(QQ, 2, 'xy').objgens()
sage: S.<xbar,ybar> = R.quo( (x^2 + y^2, y) )
sage: S.lift()
Set-theoretic ring morphism:
  From: Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 +
  To:   Multivariate Polynomial Ring in x, y over Rational Field
  Defn: Choice of lifting map
sage: S.lift() == 0
False
```

is_RingHomomorphism()

Return True if phi is of type RingHomomorphism.

EXAMPLES:

```
sage: f = Zmod(8).cover()
sage: sage.rings.morphism.is_RingHomomorphism(f)
True
sage: sage.rings.morphism.is_RingHomomorphism(2/3)
False
```

23.4 Space of homomorphisms between two rings.

RingHomset(*R*, *S*)

class RingHomset_generic(*R*, *S*)

has_coerce_map_from(*x*)
The default for coercion maps between ring homomorphism spaces is very restrictive (until more implementation work is done).

natural_map()

class RingHomset_quo_ring(*R*, *S*)
Space of ring homomorphism where the domain is a (formal) quotient ring.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S.<a,b> = R.quotient(x^2 + y^2)
sage: phi = S.hom([b,a]); phi
Ring endomorphism of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the
  Defn: a |--> b
        b |--> a
sage: phi(a)
b
sage: phi(b)
a
```

TESTS:

We test pickling of a homset from a quotient.

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S.<a,b> = R.quotient(x^2 + y^2)
sage: H = S.Hom(R)
sage: H == loads(dumps(H))
True
```

We test pickling of actual homomorphisms in a quotient:

```
sage: phi = S.hom([b,a])
sage: phi == loads(dumps(phi))
True
```

is_RingHomset(*H*)

23.5 Infinity Rings

The unsigned infinity "ring" is the set of two elements

1. infinity
2. A number less than infinity

The rules for arithmetic are that the unsigned infinity ring does not canonically coerce to any other ring, and all other rings canonically coerce to the unsigned infinity ring, sending all elements to the single element "a number less than infinity" of the unsigned infinity ring. Arithmetic and comparisons then take place in the unsigned infinity ring, where all arithmetic operations that are well-defined are defined.

The infinity "ring" is the set of five elements

1. plus infinity
2. a positive finite element
3. zero
4. a negative finite element
5. negative infinity

The infinity ring coerces to the unsigned infinity ring, sending the infinite elements to infinity and the non-infinite elements to "a number less than infinity." Any ordered ring coerces to the infinity ring in the obvious way.

Note: the shorthand oo is predefined in Sage to be the same as +Infinity in the infinity ring. It is considered equal to, but not the same as Infinity in the UnsignedInfinityRing:

```
sage: oo
+Infinity
sage: oo is InfinityRing.0
True
sage: oo is UnsignedInfinityRing.0
False
sage: oo == UnsignedInfinityRing.0
True
```

EXAMPLES:

We fetch the unsigned infinity ring and create some elements:

```
sage: P = UnsignedInfinityRing; P
The Unsigned Infinity Ring
sage: P(5)
A number less than infinity
sage: P.ngens()
1
sage: unsigned_oo = P.0; unsigned_oo
Infinity
```

We compare finite numbers with infinity:

```
sage: 5 < unsigned_oo
True
sage: 5 > unsigned_oo
False
sage: unsigned_oo < 5
False
sage: unsigned_oo > 5
True
```

We do arithmetic:

```
sage: unsigned_oo + 5
Infinity
```

We make 1 / unsigned_oo return the integer 0 so that arithmetic of the following type works:

```
sage: (1/unsigned_oo) + 2
2
sage: 32/5 - (2.439/unsigned_oo)
32/5
```

Note that many operations are not defined, since the result is not well-defined:

```
sage: unsigned_oo/0
...
ValueError: unsigned oo times smaller number not defined
```

What happened above is that 0 is canonically coerced to "a number less than infinity" in the unsigned infinity ring, and the quotient is then not well-defined.

```
sage: 0/unsigned_oo
0
sage: unsigned_oo * 0
...
ValueError: unsigned oo times smaller number not defined
sage: unsigned_oo/unsigned_oo
...
ValueError: unsigned oo times smaller number not defined
```

In the infinity ring, we can negate infinity, multiply positive numbers by infinity, etc.

```
sage: P = InfinityRing; P
The Infinity Ring
sage: P(5)
A positive finite number
```

The symbol oo is predefined as a shorthand for +Infinity:

```
sage: oo
+Infinity
```

We compare finite and infinite elements:

```
sage: 5 < oo
True
sage: P(-5) < P(5)
True
sage: P(2) < P(3)
False
sage: -oo < oo
True
```

We can do more arithmetic than in the unsigned infinity ring:

```
sage: 2 * oo
+Infinity
sage: -2 * oo
-Infinity
sage: 1 - oo
-Infinity
sage: 1 / oo
0
sage: -1 / oo
0
```

We make 1 / oo and 1 / -oo return the integer 0 instead of the infinity ring Zero so that arithmetic of the following type works:

```
sage: (1/oo) + 2
2
sage: 32/5 - (2.439/-oo)
32/5
```

If we try to subtract infinities or multiply infinity by zero we still get an error:

```
sage: oo - oo
...
SignError: cannot add infinity to minus infinity
sage: 0 * oo
...
SignError: cannot multiply infinity by zero
sage: P(2) + P(-3)
...
SignError: cannot add positive finite value to negative finite value
```

TESTS:

```
sage: P = InfinityRing
sage: P == loads(dumps(P))
True
```

```
sage: P(2) == loads(dumps(P(2)))
True
```

The following is assumed in a lot of code (i.e., "is" is used for testing whether something is infinity), so make sure it is satisfied:

```
sage: loads(dumps(infinity)) is infinity
True
```

class AnInfinity()

lcm(*x*)

Return the least common multiple of oo and x, which is by definition oo unless x is 0.

EXAMPLES:

```
sage: oo.lcm(0)
0
sage: oo.lcm(oo)
```

```
+Infinity
sage: oo.lcm(-oo)
+Infinity
sage: oo.lcm(10)
+Infinity
sage: (-oo).lcm(10)
+Infinity
```

class **FiniteNumber**(*parent, x*)

sqrt()

EXAMPLES:

```
sage: InfinityRing(7).sqrt()
A positive finite number
sage: InfinityRing(0).sqrt()
Zero
sage: InfinityRing(-.001).sqrt()
...
SignError: cannot take square root of a negative number
```

class **InfinityRing_class**()

fraction_field()

This isn't really a ring, let alone an integral domain.

TEST:

```
sage: InfinityRing.fraction_field()
...
TypeError: infinity 'ring' has no fraction field
```

gen(*n=0*)

The two generators are plus and minus infinity.

EXAMPLES:

```
sage: InfinityRing.gen(0)
+Infinity
sage: InfinityRing.gen(1)
-Infinity
sage: InfinityRing.gen(2)
...
IndexError: n must be 0 or 1
```

gens()

The two generators are plus and minus infinity.

EXAMPLES:

```
sage: InfinityRing.gens()
[+Infinity, -Infinity]
```

ngens()

The two generators are plus and minus infinity.

EXAMPLES:

```
sage: InfinityRing.ngens()
2
sage: len(InfinityRing.gens())
2
```

class LessThanInfinity(*parent=The Unsigned Infinity Ring*)

class MinusInfinity()

sqrt()
EXAMPLES:

```
sage: (-oo).sqrt()
...
SignError: cannot take square root of negative infinity
```

class PlusInfinity()

sqrt()
EXAMPLES:

```
sage: oo.sqrt()
+Infinity
```

exception SignError

class UnsignedInfinity()

class UnsignedInfinityRing_class()

fraction_field()
The unsigned infinity ring isn't an integral domain.
EXAMPLES:

```
sage: UnsignedInfinityRing.fraction_field()
...
TypeError: infinity 'ring' has no fraction field
```

gen(*n=0*)
The "generator" of self is the infinity object.
EXAMPLES:

```
sage: UnsignedInfinityRing.gen()
Infinity
sage: UnsignedInfinityRing.gen(1)
...
IndexError: UnsignedInfinityRing only has one generator
```

gens()
The "generator" of self is the infinity object.
EXAMPLES:

```
sage: UnsignedInfinityRing.gens()
[Infinity]
```

less_than_infinity()
This is the element that represents a finite value.
EXAMPLES:

```
sage: UnsignedInfinityRing.less_than_infinity()
A number less than infinity
sage: UnsignedInfinityRing(5) is UnsignedInfinityRing.less_than_infinity()
True
```

ngens()
The unsigned infinity ring has one "generator."
EXAMPLES:

```
sage: UnsignedInfinityRing.ngens()
1
sage: len(UnsignedInfinityRing.gens())
1
```

is_Infinite(*x*)
This is a type check for infinity elements.
EXAMPLES:

```
sage: sage.rings.infinity.is_Infinite(oo)
True
sage: sage.rings.infinity.is_Infinite(-oo)
True
sage: sage.rings.infinity.is_Infinite(unsigned_infinity)
True
sage: sage.rings.infinity.is_Infinite(3)
False
sage: sage.rings.infinity.is_Infinite(RR(infinity))
False
sage: sage.rings.infinity.is_Infinite(ZZ)
False
```

23.6 Fraction Field of Integral Domains

AUTHORS:

- William Stein (with input from David Joyner, David Kohel, and Joe Wetherell)
- Burcin Erocal

EXAMPLES:

Quotienting is a constructor for an element of the fraction field:

```
sage: R.<x> = QQ[]
sage: (x^2-1)/(x+1)
x - 1
sage: parent((x^2-1)/(x+1))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

The GCD is not taken (since it doesn't converge sometimes) in the inexact case.

```
sage: Z.<z> = CC[]
sage: I = CC.gen()
sage: (1+I+z)/(z+0.1*I)
(1.00000000000000*z + 1.00000000000000 + 1.00000000000000*I)/(1.00000000000000*z + 0.100000000000000
sage: (1+I*z)/(z+1.1)
(1.00000000000000*I*z + 1.00000000000000)/(1.00000000000000*z + 1.10000000000000)
```

TESTS:

```
sage: F = FractionField(IntegerRing())
sage: F == loads(dumps(F))
True

sage: F = FractionField(PolynomialRing(RationalField(),'x'))
sage: F == loads(dumps(F))
True

sage: F = FractionField(PolynomialRing(IntegerRing(),'x'))
sage: F == loads(dumps(F))
True

sage: F = FractionField(PolynomialRing(RationalField(),2,'x'))
sage: F == loads(dumps(F))
True
```

FractionField(*R, names=None*)

Create the fraction field of the integral domain R.

INPUT:

- R - an integral domain
- names - ignored

EXAMPLES: We create some example fraction fields.

```
sage: FractionField(IntegerRing())
Rational Field
sage: FractionField(PolynomialRing(RationalField(),'x'))
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: FractionField(PolynomialRing(IntegerRing(),'x'))
Fraction Field of Univariate Polynomial Ring in x over Integer Ring
sage: FractionField(PolynomialRing(RationalField(),2,'x'))
Fraction Field of Multivariate Polynomial Ring in x0, x1 over Rational Field
```

Dividing elements often implicitly creates elements of the fraction field.

```
sage: x = PolynomialRing(RationalField(), 'x').gen()
sage: f = x/(x+1)
sage: g = x**3/(x+1)
sage: f/g
1/x^2
sage: g/f
x^2
```

The input must be an integral domain.

```
sage: Frac(Integers(4))
...
TypeError: R must be an integral domain.
```

class FractionField_generic(*R, element_class=<type 'sage.rings.fraction_field_element.FractionFieldElement'>*)

The fraction field of an integral domain.

base_ring()
Return the base ring of self; this is the base ring of the ring which this fraction field is the fraction field of.
EXAMPLES:

```
sage: R = Frac(ZZ['t'])
sage: R.base_ring()
Integer Ring
```

characteristic()
Return the characteristic of this fraction field.
EXAMPLES:

```
sage: R = Frac(ZZ['t'])
sage: R.base_ring()
Integer Ring
sage: R = Frac(ZZ['t']); R.characteristic()
0
sage: R = Frac(GF(5)['w']); R.characteristic()
5
```

construction()
EXAMPLES:

```
sage: Frac(ZZ['x']).construction()
(FractionField, Univariate Polynomial Ring in x over Integer Ring)
sage: K = Frac(GF(3)['t'])
sage: f, R = K.construction()
sage: f(R)
Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 3
sage: f(R) == K
True
```

gen(*i=0*)
Return the ith generator of self.
EXAMPLES:

```
sage: R = Frac(PolynomialRing(QQ,'z',10)); R
Fraction Field of Multivariate Polynomial Ring in z0, z1, z2, z3, z4, z5, z6, z7, z8, z9 ove
sage: R.0
z0
sage: R.gen(3)
z3
sage: R.3
z3
```

is_exact()
EXAMPLES:

```
sage: Frac(ZZ['x']).is_exact()
True
sage: Frac(CDF['x']).is_exact()
False
```

is_field()
Returns True, since the fraction field is a field.
EXAMPLES:

```
sage: Frac(ZZ).is_field()
True
```

ngens()
This is the same as for the parent object.
EXAMPLES:

```
sage: R = Frac(PolynomialRing(QQ,'z',10)); R
Fraction Field of Multivariate Polynomial Ring in z0, z1, z2, z3, z4, z5, z6, z7, z8, z9 ove
sage: R.ngens()
10
```

random_element(**args, **kwds*)
Returns a random element in this fraction field.
EXAMPLES:

```
sage: F = ZZ['x'].fraction_field()
sage: F.random_element()
(2*x - 8)/(-x^2 + x)

sage: F.random_element(degree=5)
(-12*x^5 - 2*x^4 - x^3 - 95*x^2 + x + 2)/(-x^5 + x^4 - x^3 + x^2)
```

ring()
Return the ring that this is the fraction field of.
EXAMPLES:

```
sage: R = Frac(QQ['x,y'])
sage: R
Fraction Field of Multivariate Polynomial Ring in x, y over Rational Field
sage: R.ring()
Multivariate Polynomial Ring in x, y over Rational Field
```

is_FractionField(*x*)
Tests whether or not x inherits from FractionField_generic.
EXAMPLES:

```
sage: from sage.rings.fraction_field import is_FractionField
sage: is_FractionField(Frac(ZZ['x']))
True
sage: is_FractionField(QQ)
False
```

23.7 Fraction Field Elements

AUTHORS:

- William Stein (input from David Joyner, David Kohel, and Joe Wetherell)

class **FractionFieldElement**()
EXAMPLES:

```
sage: K, x = FractionField(PolynomialRing(QQ, 'x')).objgen()
sage: K
Fraction Field of Univariate Polynomial Ring in x over Rational Field
sage: loads(K.dumps()) == K
True
sage: f = (x^3 + x)/(17 - x^19); f
```

```
(x^3 + x)/(-x^19 + 17)
sage: loads(f.dumps()) == f
True
```

TESTS:

Test if #5451 is fixed:

```
sage: A = FiniteField(9,'theta')['t']
sage: K.<t> = FractionField(A)
sage: f= 2/(t^2+2*t); g =t^9/(t^18 + t^10 + t^2);f+g
(2*t^15 + 2*t^14 + 2*t^13 + 2*t^12 + 2*t^11 + 2*t^10 + 2*t^9 + t^7 + t^6 + t^5 + t^4 + t^3 + t^2
```

copy()

EXAMPLES:

```
sage: R.<x,y> = ZZ[]
sage: f = x/y+1; f
(x + y)/y
sage: f.copy()
(x + y)/y
```

denominator()

EXAMPLES:

```
sage: R.<x,y> = ZZ[]
sage: f = x/y+1; f
(x + y)/y
sage: f.denominator()
y
```

derivative()

The derivative of this rational function, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See Also:

_derivative()

EXAMPLES:

```
sage: F = FractionField(PolynomialRing(RationalField(),'x'))
sage: x = F.gen()
sage: (1/x).derivative()
-1/x^2

sage: (x+1/x).derivative(x, 2)
2/x^3

sage: F = FractionField(PolynomialRing(RationalField(),'x,y'))
sage: x,y = F.gens()
sage: (1/(x+y)).derivative(x,y)
2/(x^3 + 3*x^2*y + 3*x*y^2 + y^3)
```

factor()

Return the factorization of self over the base ring

EXAMPLES:

```
sage: K.<x> = QQ[]
sage: f = (x^3+x)/(x-3)
```

```
sage: f.factor()
(x - 3)^-1 * x * (x^2 + 1)
```

is_one()

Returns True if this element is equal to one.

EXAMPLES:

```
sage: F = ZZ['x,y'].fraction_field()
sage: x,y = F.gens()
sage: (x/x).is_one()
True
sage: (x/y).is_one()
False
```

is_zero()

Returns True if this element is equal to zero.

EXAMPLES:

```
sage: F = ZZ['x,y'].fraction_field()
sage: x,y = F.gens()
sage: t = F(0)/x
sage: t.is_zero()
True
sage: u = 1/x - 1/x
sage: u.is_zero()
True
sage: u.parent() is F
True
```

numerator()

EXAMPLES:

```
sage: R.<x,y> = ZZ[]
sage: f = x/y+1; f
(x + y)/y
sage: f.numerator()
x + y
```

partial_fraction_decomposition()

Decomposes fraction field element into a whole part and a list of fraction field elements over prime power denominators.

The sum will be equal to the original fraction.

AUTHORS:

- Robert Bradshaw (2007-05-31)

EXAMPLES:

```
sage: S.<t> = QQ[]
sage: q = 1/(t+1) + 2/(t+2) + 3/(t-3); q
(6*t^2 + 4*t - 6)/(t^3 - 7*t - 6)
sage: whole, parts = q.partial_fraction_decomposition(); parts
[3/(t - 3), 1/(t + 1), 2/(t + 2)]
sage: sum(parts) == q
True
sage: q = 1/(t^3+1) + 2/(t^2+2) + 3/(t-3)^5
sage: whole, parts = q.partial_fraction_decomposition(); parts
[1/3/(t + 1), 3/(t^5 - 15*t^4 + 90*t^3 - 270*t^2 + 405*t - 243), (-1/3*t + 2/3)/(t^2 - t + 1
sage: sum(parts) == q
True
```

We do the best we can over in-exact fields:

```
sage: R.<x> = RealField(20)[]
sage: q = 1/(x^2 + 2)^2 + 1/(x-1); q
(1.0000*x^4 + 4.0000*x^2 + 1.0000*x + 3.0000)/(1.0000*x^5 - 1.0000*x^4 + 4.0000*x^3 - 4.0000
sage: whole, parts = q.partial_fraction_decomposition(); parts
[1.0000/(1.0000*x - 1.0000), 1.0000/(1.0000*x^4 + 4.0000*x^2 + 4.0000)]
sage: sum(parts)
(1.0000*x^4 + 4.0000*x^2 + 1.0000*x + 3.0000)/(1.0000*x^5 - 1.0000*x^4 + 4.0000*x^3 - 4.0000
```

TESTS:

We test partial fraction for irreducible denominators:

```
sage: R.<x> = ZZ[]
sage: q = x^2/(x-1)
sage: q.partial_fraction_decomposition()
(x + 1, [1/(x - 1)])
sage: q = x^10/(x-1)^5
sage: whole, parts = q.partial_fraction_decomposition()
sage: whole + sum(parts) == q
True
```

And also over finite fields (see trac #6052):

```
sage: R.<x> = GF(2)[]
sage: q = (x+1)/(x^3+x+1)
sage: q.partial_fraction_decomposition()
(0, [(x + 1)/(x^3 + x + 1)])
```

reduce()

Divides out the gcd of the numerator and denominator.

Automatically called for exact rings, but because it may be numerically unstable for inexact rings it must be called manually in that case.

EXAMPLES:

```
sage: R.<x> = RealField(10)[]
sage: f = (x^2+2*x+1)/(x+1); f
(1.0*x^2 + 2.0*x + 1.0)/(1.0*x + 1.0)
sage: f.reduce(); f
1.0*x + 1.0
```

valuation()

Return the valuation of self, assuming that the numerator and denominator have valuation functions defined on them.

EXAMPLES:

```
sage: x = PolynomialRing(RationalField(),'x').gen()
sage: f = (x**3 + x)/(x**2 - 2*x**3)
sage: f
(x^2 + 1)/(-2*x^2 + x)
sage: f.valuation()
-1
```

is_FractionFieldElement()

Returns whether or not x is of type FractionFieldElement

EXAMPLES:

```
sage: from sage.rings.fraction_field_element import is_FractionFieldElement
sage: R.<x> = ZZ[]
sage: is_FractionFieldElement(x/2)
```

```
False
sage: is_FractionFieldElement(2/x)
True
sage: is_FractionFieldElement(1/3)
False
```

make_element()
Used for unpickling FractionFieldElement objects (and subclasses).

EXAMPLES:

```
sage: from sage.rings.fraction_field_element import make_element
sage: R = ZZ['x,y']
sage: x,y = R.gens()
sage: F = R.fraction_field()
sage: make_element(F, 1+x, 1+y)
(x + 1)/(y + 1)
```

make_element_old()
Used for unpickling old FractionFieldElement pickles.

EXAMPLES:

```
sage: from sage.rings.fraction_field_element import make_element_old
sage: R.<x,y> = ZZ[]
sage: F = R.fraction_field()
sage: make_element_old(F, {'_FractionFieldElement__numerator':x+y,'_FractionFieldElement__denomi
(x + y)/(x - y)
```

23.8 Quotient Rings

AUTHORS:

- William Stein

TESTS:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: S = R.quotient_ring(I);
sage: S == loads(dumps(S))
True
```

QuotientRing(*R, I, names=None*)
Creates a quotient ring of the ring R by the ideal I. Variables are labeled by names. (If the quotient ring is a quotient of a polynomial ring.). If names isn't given, 'bar' will be appended to the variable names in R.

INPUTS:

- R - a commutative ring
- I - an ideal of R
- names - a list of strings to be used as names for the variables in the quotient ring R/I

OUTPUTS: R/I - the quotient ring R mod the ideal I

EXAMPLES:

Some simple quotient rings with the integers:

```
sage: R = QuotientRing(ZZ,7*ZZ); R
Quotient of Integer Ring by the ideal (7)
sage: R.gens()
(1,)
sage: 1*R(3); 6*R(3); 7*R(3)
3
4
0


sage: S = QuotientRing(ZZ,ZZ.ideal(8)); S
Quotient of Integer Ring by the ideal (8)
sage: 2*S(4)
0
```

With polynomial rings: (note that the variable name of the quotient ring can be specified as shown below)

```
sage: R.<xx> = QuotientRing(QQ[x], QQ[x].ideal(x^2 + 1)); R
Univariate Quotient Polynomial Ring in xx over Rational Field with modulus x^2 + 1
sage: R.gens(); R.gen()
(xx,)
xx
sage: for n in range(4): xx^n
1
xx
-1
-xx


sage: S = QuotientRing(QQ[x], QQ[x].ideal(x^2 - 2)); S
Univariate Quotient Polynomial Ring in xbar over Rational Field with
modulus x^2 - 2
sage: xbar = S.gen(); S.gen()
xbar
sage: for n in range(3): xbar^n
1
xbar
2
```

Sage coerces objects into ideals when possible:

```
sage: R = QuotientRing(QQ[x], x^2 + 1); R
Univariate Quotient Polynomial Ring in xbar over Rational Field with
modulus x^2 + 1
```

By Noether's homomorphism theorems, the quotient of a quotient ring in R is just the quotient of R by the sum of the ideals. In this example, we end up modding out the ideal (x) from the ring QQ[x,y]:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: S.<a,b> = QuotientRing(R,R.ideal(1 + y^2))
sage: T.<c,d> = QuotientRing(S,S.ideal(a))
sage: T
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x, y^2 + 1)
sage: R.gens(); S.gens(); T.gens()
(x, y)
```

```
(a, b)
(0, d)
sage: for n in range(4): d^n
1
d
-1
-d
```

class **QuotientRing_generic**(*R, I, names*)

The quotient ring of R by the ideal I.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ,'x')
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: S = R.quotient_ring(I); S
Quotient of Univariate Polynomial Ring in x over Integer Ring by the ideal (x^2 + 3*x + 4, x^2 +
```

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: S.<a,b> = R.quo(x^2 + y^2)
sage: a^2 + b^2 == 0
True
sage: S(0) == a^2 + b^2
True
```

EXAMPLE: Quotient of quotient

A quotient of a quotient is just the quotient of the original top ring by the sum of two ideals.

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: S.<a,b> = R.quo(1 + y^2)
sage: T.<c,d> = S.quo(a)
sage: T
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x, y^2 + 1)
sage: T.gens()
(0, d)
```

characteristic()

Return the characteristic of the quotient ring.

TODO: Not yet implemented!

EXAMPLES:

```
sage: Q = QuotientRing(ZZ,7*ZZ)
sage: Q.characteristic()
...
NotImplementedError
```

construction()

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ,'x')
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: R.quotient_ring(I).construction()
(QuotientFunctor, Univariate Polynomial Ring in x over Integer Ring)
```

TESTS:

```
sage: F, R = Integers(5).construction()
sage: F(R)
Ring of integers modulo 5
sage: F, R = GF(5).construction()
sage: F(R)
Finite Field of size 5
```

cover()

The covering ring homomorphism $R \to R/I$, equipped with a section.

EXAMPLES:

```
sage: R = ZZ.quo(3*ZZ)
sage: pi = R.cover()
sage: pi
Ring morphism:
  From: Integer Ring
  To:   Ring of integers modulo 3
  Defn: Natural quotient map
sage: pi(5)
2
sage: l = pi.lift()
```

EXAMPLES:

```
sage: R.<x,y>  = PolynomialRing(QQ)
sage: Q = R.quo( (x^2,y^2) )
sage: pi = Q.cover()
sage: pi(x^3+y)
ybar
sage: l = pi.lift(x+y^3)
sage: l
x
sage: l = pi.lift(); l
Set-theoretic ring morphism:
  From: Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x
  To:   Multivariate Polynomial Ring in x, y over Rational Field
  Defn: Choice of lifting map
sage: l(x+y^3)
x
```

cover_ring()

Returns the cover ring of the quotient ring: that is, the original ring R from which we modded out an ideal, I.

TODO: PolynomialQuotientRings_field objects don't have a `cover_ring` function.

EXAMPLES:

```
sage: Q = QuotientRing(ZZ,7*ZZ)
sage: Q.cover_ring()
Integer Ring

sage: Q = QuotientRing(QQ[x], x^2 + 1)
sage: Q.cover_ring()
...
AttributeError: 'PolynomialQuotientRing_field' object has no attribute 'cover_ring'
```

defining_ideal()

Returns the ideal generating this quotient ring.

EXAMPLES:

In the integers:

```
sage: Q = QuotientRing(ZZ,7*ZZ)
sage: Q.defining_ideal()
Principal ideal (7) of Integer Ring
```

An example involving a quotient of a quotient. By Noether's homomorphism theorems, this is actually a quotient by a sum of two ideals:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: S.<a,b> = QuotientRing(R,R.ideal(1 + y^2))
sage: T.<c,d> = QuotientRing(S,S.ideal(a))
sage: S.defining_ideal()
Ideal (y^2 + 1) of Multivariate Polynomial Ring in x, y over Rational Field
sage: T.defining_ideal()
Ideal (x, y^2 + 1) of Multivariate Polynomial Ring in x, y over Rational Field
```

gen(*i=0*)

Returns the ith generator for this quotient ring.

EXAMPLES:

```
sage: R = QuotientRing(ZZ,7*ZZ)
sage: R.gen(0)
1
```

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: S.<a,b> = QuotientRing(R,R.ideal(1 + y^2))
sage: T.<c,d> = QuotientRing(S,S.ideal(a))
sage: T
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x, y^2 +
sage: R.gen(0); R.gen(1)
x
y
sage: S.gen(0); S.gen(1)
a
b
sage: T.gen(0); T.gen(1)
0
d
```

ideal(**gens, **kwds*)

Return the ideal of self with the given generators.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: S = R.quotient_ring(x^2+y^2)
sage: S.ideal()
Ideal (0) of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ide
sage: S.ideal(x+y+1)
Ideal (xbar + ybar + 1) of Quotient of Multivariate Polynomial Ring in x, y over Rational Fi
```

is_field()

Returns True if the quotient ring is a field. Checks to see if the defining ideal is maximal.

TESTS:

Requires the `is_maximal` function to be implemented:

```
sage: Q = QuotientRing(ZZ,7*ZZ)
sage: Q.is_field()
...
NotImplementedError
```

is_integral_domain()

If this function returns True then self is definitely an integral domain. If it returns False, then either self is definitely not an integral domain or this function was unable to determine whether or not self is an integral domain.

Use `self.defining_ideal().is_prime()` to find out for sure whether this quotient ring is really not an integral domain, of if Sage is unable to determine the answer.

EXAMPLES:

```
sage: R = Integers(8)
sage: R.is_integral_domain()
False
sage: R.<a,b,c> = ZZ['a','b','c']
sage: I = R.ideal(a,b)
sage: Q = R.quotient_ring(I)
sage: Q.is_integral_domain()
...
NotImplementedError
```

lift()

Return the lifting map to the cover.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S = R.quotient(x^2 + y^2)
sage: pi = S.cover(); pi
Ring morphism:
  From: Multivariate Polynomial Ring in x, y over Rational Field
  To:   Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x
  Defn: Natural quotient map
sage: L = S.lift(); L
Set-theoretic ring morphism:
  From: Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x
  To:   Multivariate Polynomial Ring in x, y over Rational Field
  Defn: Choice of lifting map
sage: L(S.0)
x
sage: L(S.1)
y
```

Note that some reduction may be applied so that the lift of a reduction need not equal the original element.

```
sage: z = pi(x^3 + 2*y^2); z
-xbar*ybar^2 + 2*ybar^2
sage: L(z)
-x*y^2 + 2*y^2
sage: L(z) == x^3 + 2*y^2
False
```

ngens()

Returns the number of generators for this quotient ring.

TODO: Note that `ngens` counts 0 as a generator. Does this make sense? That is, since 0 only generates itself and the fact that this is true for all rings, is there a way to "knock it off" of the generators list if a generator of some original ring is modded out?

EXAMPLES:

```
sage: R = QuotientRing(ZZ,7*ZZ)
sage: R.gens(); R.ngens()
(1,)
1
```

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: S.<a,b> = QuotientRing(R,R.ideal(1 + y^2))
sage: T.<c,d> = QuotientRing(S,S.ideal(a))
sage: T
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x, y^2 +
sage: R.gens(); S.gens(); T.gens()
(x, y)
(a, b)
(0, d)
sage: R.ngens(); S.ngens(); T.ngens()
2
2
2
```

is_QuotientRing(*x*)

Tests whether or not x inherits from QuotientRing_generic.

EXAMPLES:

```
sage: from sage.rings.quotient_ring import is_QuotientRing
sage: R.<x> = PolynomialRing(ZZ,'x')
sage: I = R.ideal([4 + 3*x + x^2, 1 + x^2])
sage: S = R.quotient_ring(I)
sage: is_QuotientRing(S)
True
sage: is_QuotientRing(R)
False
```

23.9 Quotient Ring Elements

AUTHORS:

- William Stein

class QuotientRingElement(*parent, rep, reduce=True*)

An element of a quotient ring R/I.

INPUT:

- `parent` - the ring R/I
- `rep` - a representative of the element in R; this is used as the internal representation of the element
- `reduce` - bool (optional, default: True) - if True, then the internal representation of the element is `rep` reduced modulo the ideal I

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: S.<xbar> = R.quo((4 + 3*x + x^2, 1 + x^2)); S
Quotient of Univariate Polynomial Ring in x over Integer Ring by the ideal (x^2 + 3*x + 4, x^2 +
sage: v = S.gens(); v
(xbar,)
```

```
sage: loads(v[0].dumps()) == v[0]
True
```

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S = R.quo(x^2 + y^2); S
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 + y^2)
sage: S.gens()
(xbar, ybar)
```

We name each of the generators.

```
sage: S.<a,b> = R.quotient(x^2 + y^2)
sage: a
a
sage: b
b
sage: a^2 + b^2 == 0
True
sage: b.lift()
y
sage: (a^3 + b^2).lift()
-x*y^2 + y^2
```

is_unit()

Return True if self is a unit in the quotient ring.

TODO: This is not fully implemented, as illustrated in the example below. So far, self is determined to be unit only if its representation in the cover ring R is also a unit.

EXAMPLES:

```
sage: R.<x,y> = QQ[]; S.<a,b> = R.quo(1 - x*y); type(a)
<class 'sage.rings.quotient_ring_element.QuotientRingElement'>
sage: a*b
1
sage: a.is_unit()
...
NotImplementedError
sage: S(1).is_unit()
True
```

lc()

Return the leading coefficent of this quotient ring element.

EXAMPLE:

```
sage: R.<x,y,z>=PolynomialRing(GF(7),3,order='lex')
sage: I = sage.rings.ideal.FieldIdeal(R)
sage: Q = R.quo( I )
sage: f = Q( z*y + 2*x )
sage: f.lc()
2
```

TESTS:

```
sage: R.<x,y> = QQ[]; S.<a,b> = R.quo(x^2 + y^2); type(a)
<class 'sage.rings.quotient_ring_element.QuotientRingElement'>
sage: (a+3*a*b+b).lc()
3
```

lift()

If self is an element of R/I, then return self as an element of R.

EXAMPLES:

```
sage: R.<x,y> = QQ[]; S.<a,b> = R.quo(x^2 + y^2); type(a)
<class 'sage.rings.quotient_ring_element.QuotientRingElement'>
sage: a.lift()
x
sage: (3/5*(a + a^2 + b^2)).lift()
3/5*x
```

lm()

Return the leading monomial of this quotient ring element.

EXAMPLE:

```
sage: R.<x,y,z>=PolynomialRing(GF(7),3,order='lex')
sage: I = sage.rings.ideal.FieldIdeal(R)
sage: Q = R.quo( I )
sage: f = Q( z*y + 2*x )
sage: f.lm()
xbar
```

TESTS:

```
sage: R.<x,y> = QQ[]; S.<a,b> = R.quo(x^2 + y^2); type(a)
<class 'sage.rings.quotient_ring_element.QuotientRingElement'>
sage: (a+3*a*b+b).lm()
a*b
```

lt()

Return the leading term of this quotient ring element.

EXAMPLE:

```
sage: R.<x,y,z>=PolynomialRing(GF(7),3,order='lex')
sage: I = sage.rings.ideal.FieldIdeal(R)
sage: Q = R.quo( I )
sage: f = Q( z*y + 2*x )
sage: f.lt()
2*xbar
```

TESTS:

```
sage: R.<x,y> = QQ[]; S.<a,b> = R.quo(x^2 + y^2); type(a)
<class 'sage.rings.quotient_ring_element.QuotientRingElement'>
sage: (a+3*a*b+b).lt()
3*a*b
```

monomials()

EXAMPLES:

```
sage: R.<x,y> = QQ[]; S.<a,b> = R.quo(x^2 + y^2); type(a)
<class 'sage.rings.quotient_ring_element.QuotientRingElement'>
sage: a.monomials()
[a]
sage: (a+a*b).monomials()
[a*b, a]
```

reduce(G)

Reduce this quotient ring element by a set of quotient ring elements G.

INPUT:

- G - a list of quotient ring elements

EXAMPLE:

```
sage: P.<a,b,c,d,e> = PolynomialRing(GF(2), 5, order='lex')
sage: I1 = ideal([a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1])
sage: Q = P.quotient( sage.rings.ideal.FieldIdeal(P) )
sage: I2 = ideal([Q(f) for f in I1.gens()])
sage: f = Q((a*b + c*d + 1)^2  + e)
sage: f.reduce(I2.gens())
ebar
```

variables()

EXAMPLES:

```
sage: R.<x,y> = QQ[]; S.<a,b> = R.quo(x^2 + y^2); type(a)
<class 'sage.rings.quotient_ring_element.QuotientRingElement'>
sage: a.variables()
[a]
sage: b.variables()
[b]
sage: s = a^2 + b^2 + 1; s
1
sage: s.variables()
[]
sage: (a+b).variables()
[a, b]
```

CHAPTER

TWENTYFOUR

STANDARD COMMUTATIVE RINGS

24.1 Ring $\mathbb{Z}$ of Integers

The class `IntegerRing` represents the ring $\mathbf{Z}$ of (arbitrary precision) integers. Each integer is an instance of the class `Integer`, which is defined in a Pyrex extension module that wraps GMP integers (the `mpz_t` type in GMP).

```
sage: Z = IntegerRing(); Z
Integer Ring
sage: Z.characteristic()
0
sage: Z.is_field()
False
```

There is a unique instances of class `IntegerRing`. To create an `Integer`, coerce either a Python int, long, or a string. Various other types will also coerce to the integers, when it makes sense.

```
sage: a = Z(1234); b = Z(5678); print a, b
1234 5678
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: a + b
6912
sage: Z('9480384908398593485983458394 5394')
948038490839859348598345839453 94
```

IntegerRing()
Return the integer ring

EXAMPLE:

```
sage: IntegerRing()
Integer Ring
sage: ZZ==IntegerRing()
True
```

class **IntegerRing_class**()
The ring of integers.

In order to introduce the ring $\mathbf{Z}$ of integers, we illustrate creation, calling a few functions, and working with its elements.

```
sage: Z = IntegerRing(); Z
Integer Ring
```

```
sage: Z.characteristic()
0
sage: Z.is_field()
False
```

We next illustrate basic arithmetic in $\mathbf{Z}$:

```
sage: a = Z(1234); b = Z(5678); print a, b
1234 5678
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: a + b
6912
sage: b + a
6912
sage: a * b
7006652
sage: b * a
7006652
sage: a - b
-4444
sage: b - a
4444
```

When we divide to integers using /, the result is automatically coerced to the field of rational numbers, even if the result is an integer.

```
sage: a / b
617/2839
sage: type(a/b)
<type 'sage.rings.rational.Rational'>
sage: a/a
1
sage: type(a/a)
<type 'sage.rings.rational.Rational'>
```

For floor division, instead using the // operator:

```
sage: a // b
0
sage: type(a//b)
<type 'sage.rings.integer.Integer'>
```

Next we illustrate arithmetic with automatic coercion. The types that coerce are: str, int, long, Integer.

```
sage: a + 17
1251
sage: a * 374
461516
sage: 374 * a
461516
sage: a/19
1234/19
sage: 0 + Z(-64)
-64
```

Integers can be coerced:

```
sage: a = Z(-64)
sage: int(a)
-64
```

We can create integers from several types of objects.

```
sage: ZZ(17/1)
17
sage: ZZ(Mod(19,23))
19
sage: ZZ(2 + 3*5 + O(5^3))
17
```

absolute_degree()
Return the absolute degree of the integers, which is 1
EXAMPLE:

```
sage: ZZ.absolute_degree()
1
```

characteristic()
Return the characteristic of the integers, which is 0
EXAMPLE:

```
sage: ZZ.characteristic()
0
```

completion()
Returns the completion of Z at p.
EXAMPLES:

```
sage: ZZ.completion(infinity, 53)
Real Field with 53 bits of precision
sage: ZZ.completion(5, 15, {'print_mode': 'bars'})
5-adic Ring with capped relative precision 15
```

degree()
Return the degree of the integers, which is 1
EXAMPLE:

```
sage: ZZ.degree()
1
```

extension()
Returns the order in the number field defined by poly generated (as a ring) by a root of poly.
EXAMPLES:

```
sage: ZZ.extension(x^2-5, 'a')
Order in Number Field in a with defining polynomial x^2 - 5
sage: ZZ.extension([x^2 + 1, x^2 + 2], 'a,b')
Relative Order in Number Field in a with defining polynomial x^2 + 1 over its base field
```

fraction_field()
Returns the field of rational numbers - the fraction field of the integers.
EXAMPLES:

```
sage: ZZ.fraction_field()
Rational Field
sage: ZZ.fraction_field() == QQ
True
```

gen()
Returns the additive generator of the integers, which is 1.
EXAMPLES:

```
sage: ZZ.gen()
1
sage: type(ZZ.gen())
<type 'sage.rings.integer.Integer'>
```

gens()
Returns the tuple (1,) containing a single element, the additive generator of the integers, which is 1.
EXAMPLES:

```
sage: ZZ.gens(); ZZ.gens()[0]
(1,)
1
sage: type(ZZ.gens()[0])
<type 'sage.rings.integer.Integer'>
```

is_atomic_repr()
Return True, since elements of the integers do not have to be printed with parentheses around them, when they are coefficients, e.g., in a polynomial.
EXAMPLE:

```
sage: ZZ.is_atomic_repr()
True
```

is_field()
Return False - the integers are not a field.
EXAMPLES:

```
sage: ZZ.is_field()
False
```

is_finite()
Return False - the integers are an infinite ring.
EXAMPLES:

```
sage: ZZ.is_finite()
False
```

is_noetherian()
Return True - the integers are a Noetherian ring.
EXAMPLES:

```
sage: ZZ.is_noetherian()
True
```

is_subring()
Return True if ZZ is a subring of other in a natural way.
Every ring of characteristic 0 contains ZZ as a subring.
EXAMPLES:

```
sage: ZZ.is_subring(QQ)
True
```

krull_dimension()
Return the Krull dimension of the integers, which is 1.
EXAMPLE:

```
sage: ZZ.krull_dimension()
1
```

ngens()

Returns the number of additive generators of the ring, which is 1.

EXAMPLES:

```
sage: ZZ.ngens()
1
sage: len(ZZ.gens())
1
```

order()

Return the order (cardinality) of the integers, which is +Infinity.

EXAMPLE:

```
sage: ZZ.order()
+Infinity
```

parameter()

Returns an integer of degree 1 for the Euclidean property of ZZ, namely 1.

EXAMPLES:

```
sage: ZZ.parameter()
1
```

quotient()

Return the quotient of $\mathbf{Z}$ by the ideal I or integer I.

EXAMPLES:

```
sage: ZZ.quo(6*ZZ)
Ring of integers modulo 6
sage: ZZ.quo(0*ZZ)
Integer Ring
sage: ZZ.quo(3)
Ring of integers modulo 3
sage: ZZ.quo(3*QQ)
...
TypeError: I must be an ideal of ZZ
```

random_element()

Return a random integer.

ZZ.random_element() return an integer using the default distribution described below

ZZ.random_element(n) return an integer uniformly distributed between 0 and n-1, inclusive.

ZZ.random_element(min, max) return an integer uniformly destributed between min and max-1, inclusive.

The default distribution for ZZ.random_element() is based on $X = \text{trunc}(4/(5R))$, where R is a random variable uniformly distributed between -1 and 1. This gives $\Pr(X = 0) = 1/5$, and $\Pr(X = n) = 2/(5|n|(|n| + 1))$ for $n \neq 0$. Most of the samples will be small; -1, 0, and 1 occur with probability 1/5 each. But we also have a small but non-negligible proportion of "outliers"; $\Pr(|X| \geq n) = 4/(5n)$, so for instance, we expect that $|X| \geq 1000$ on one in 1250 samples.

We actually use an easy-to-compute truncation of the above distribution; the probabilities given above hold fairly well up to about $|n| = 10000$, but around $|n| = 30000$ some values will never be returned at all, and we will never return anything greater than 2^{30}.

EXAMPLES:

The default distribution is on average 50% ± 1

```
sage: [ZZ.random_element() for _ in range(10)]
[-8, 2, 0, 0, 1, -1, 2, 1, -95, -1]
```

The default uniform distribution is integers between -2 and 2 inclusive:

```
sage: [ZZ.random_element(distribution="uniform") \
        for _ in range(10)]
[2, -2, 2, -2, -1, 1, -1, 2, 1, 0]
```

If a range is given, the distribution is uniform in that range:

```
sage: ZZ.random_element(-10,10)
-5
sage: ZZ.random_element(10)
7
sage: ZZ.random_element(10^50)
624989715467826655980230365229312342668011858916 99
sage: [ZZ.random_element(5) for _ in range(10)]
[1, 3, 4, 0, 3, 4, 0, 3, 0, 1]
```

Notice that the right endpoint is not included:

```
sage: [ZZ.random_element(-2,2) for _ in range(10)]
[-1, -2, 0, -2, 1, -1, -1, -2, -2, 1]
```

We compute a histogram over 1000 samples of the default distribution:

```
sage: from collections import defaultdict
sage: d = defaultdict(lambda: 0)
sage: for _ in range(1000):
...       samp = ZZ.random_element()
...       d[samp] = d[samp] + 1
sage: sorted(d.items())
[(-1026, 1), (-248, 1), (-145, 1), (-81, 1), (-80, 1), (-79, 1), (-75, 1), (-69, 1), (-68, 1
```

range()

Optimized range function for Sage integer.

AUTHORS:

- Robert Bradshaw (2007-09-20)

EXAMPLES:

```
sage: ZZ.range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
sage: ZZ.range(-5,5)
[-5, -4, -3, -2, -1, 0, 1, 2, 3, 4]
sage: ZZ.range(0,50,5)
[0, 5, 10, 15, 20, 25, 30, 35, 40, 45]
sage: ZZ.range(0,50,-5)
[]
sage: ZZ.range(50,0,-5)
[50, 45, 40, 35, 30, 25, 20, 15, 10, 5]
sage: ZZ.range(50,0,5)
[]
sage: ZZ.range(50,-1,-5)
[50, 45, 40, 35, 30, 25, 20, 15, 10, 5, 0]
```

It uses different code if the step doesn't fit in a long:

```
sage: ZZ.range(0,2^83,2^80)
[0, 1208925819614629174706176, 2417851639229258349412352, 3626777458843887524118528, 4835703
```

residue_field()

Return the residue field of the integers modulo the given prime, ie $\mathbf{Z}/p\mathbf{Z}$.

INPUT:

- `prime` - a prime number
- `check` - (boolean, default True) whether or not to check the primality of prime.

OUTPUT: The residue field at this prime.

EXAMPLES:

```
sage: F = ZZ.residue_field(61); F
Residue field of Integers modulo 61
sage: pi = F.reduction_map(); pi
Partially defined reduction map from Rational Field to Residue field of Integers modulo 61
sage: pi(123/234)
6
sage: pi(1/61)
...
ZeroDivisionError: Cannot reduce rational 1/61 modulo 61: it has negative valuation
sage: lift = F.lift_map(); lift
Lifting map from Residue field of Integers modulo 61 to Rational Field
sage: lift(F(12345/67890))
33
sage: (12345/67890) % 61
33
```

Construction can be from a prime ideal instead of a prime:

```
sage: ZZ.residue_field(ZZ.ideal(97))
Residue field of Integers modulo 97
```

TESTS:

```
sage: ZZ.residue_field(ZZ.ideal(96))
...
TypeError: Principal ideal (96) of Integer Ring is not prime
sage: ZZ.residue_field(96)
...
TypeError: 96 is not prime
```

zeta()

Return a primitive n'th root of unity in the integers, or raise an error if none exists

INPUT:

- `n` - a positive integer (default 2)

OUTPUT: an n'th root of unity in ZZ

EXAMPLE:

```
sage: ZZ.zeta()
-1
sage: ZZ.zeta(1)
1
sage: ZZ.zeta(3)
...
ValueError: no nth root of unity in integer ring
sage: ZZ.zeta(0)
...
ValueError: n must be positive in zeta()
```

clear_mpz_globals()

crt_basis()

Compute and return a Chinese Remainder Theorem basis for the list X of coprime integers.

INPUT:

- X - a list of Integers that are coprime in pairs

OUTPUT:

- E - a list of Integers such that E[i] = 1 (mod X[i]) and E[i] = 0 (mod X[j]) for all j!=i.

The E[i] have the property that if A is a list of objects, e.g., integers, vectors, matrices, etc., where A[i] is moduli X[i], then a CRT lift of A is simply

sum E[i] * A[i].

ALGORITHM: To compute E[i], compute integers s and t such that

s * X[i] + t * (prod over i!=j of X[j]) = 1. (*)

Then E[i] = t * (prod over i!=j of X[j]). Notice that equation (*) implies that E[i] is congruent to 1 modulo X[i] and to 0 modulo the other X[j] for j!=i.

COMPLEXITY: We compute len(X) extended GCD's.

EXAMPLES:

```
sage: X = [11,20,31,51]
sage: E = crt_basis([11,20,31,51])
sage: E[0]%X[0]; E[1]%X[0]; E[2]%X[0]; E[3]%X[0]
1
0
0
0
sage: E[0]%X[1]; E[1]%X[1]; E[2]%X[1]; E[3]%X[1]
0
1
0
0
sage: E[0]%X[2]; E[1]%X[2]; E[2]%X[2]; E[3]%X[2]
0
0
1
0
sage: E[0]%X[3]; E[1]%X[3]; E[2]%X[3]; E[3]%X[3]
0
0
0
1
```

factor()

Return the factorization of the positive integer n as a sorted list of tuples (p_i, e_i) such that $n = \prod p_i^{e_i}$.

For further documentation see sage.rings.arith.factor()

EXAMPLE:

```
sage: sage.rings.integer_ring.factor(420)
2^2 * 3 * 5 * 7
```

gmp_randrange()

init_mpz_globals()

is_IntegerRing()
Internal funtion: returns true iff x is the ring ZZ of integers

EXAMPLES:

```
sage: from sage.rings.integer_ring import  is_IntegerRing
sage: is_IntegerRing(ZZ)
True
sage: is_IntegerRing(QQ)
False
sage: is_IntegerRing(parent(3))
True
sage: is_IntegerRing(parent(1/3))
False
```

24.2 Elements of the ring $\mathbb{Z}$ of integers

AUTHORS:

- William Stein (2005): initial version
- Gonzalo Tornaria (2006-03-02): vastly improved python/GMP conversion; hashing
- Didier Deshommes (2006-03-06): numerous examples and docstrings
- William Stein (2006-03-31): changes to reflect GMP bug fixes
- William Stein (2006-04-14): added GMP factorial method (since it's now very fast).
- David Harvey (2006-09-15): added nth_root, exact_log
- David Harvey (2006-09-16): attempt to optimise Integer constructor
- Rishikesh (2007-02-25): changed quo_rem so that the rem is positive
- David Harvey, Martin Albrecht, Robert Bradshaw (2007-03-01): optimized Integer constructor and pool
- Pablo De Napoli (2007-04-01): multiplicative_order should return +infinity for non zero numbers
- Robert Bradshaw (2007-04-12): is_perfect_power, Jacobi symbol (with Kronecker extension). Convert some methods to use GMP directly rather than pari, Integer(), PY_NEW(Integer)
- David Roe (2007-03-21): sped up valuation and is_square, added val_unit, is_power, is_power_of and divide_knowing_divisible_by
- Robert Bradshaw (2008-03-26): gamma function, multifactorials
- Robert Bradshaw (2008-10-02): bounded squarefree part

EXAMPLES:

Add 2 integers:

```
sage: a = Integer(3) ; b = Integer(4)
sage: a + b == 7
True
```

Add an integer and a real number:

```
sage: a + 4.0
7.00000000000000
```

Add an integer and a rational number:

```
sage: a + Rational(2)/5
17/5
```

Add an integer and a complex number:

```
sage: b = ComplexField().0 + 1.5
sage: loads((a+b).dumps()) == a+b
True
```

```
sage: z = 32
sage: -z
-32
sage: z = 0; -z
0
sage: z = -0; -z
0
sage: z = -1; -z
1
```

Multiplication:

```
sage: a = Integer(3) ; b = Integer(4)
sage: a * b == 12
True
sage: loads((a * 4.0).dumps()) == a*b
True
sage: a * Rational(2)/5
6/5
```

```
sage: list([2,3]) * 4
[2, 3, 2, 3, 2, 3, 2, 3]
```

```
sage: 'sage'*Integer(3)
'sagesagesage'
```

COERCIONS: Returns version of this integer in the multi-precision floating real field R.

```
sage: n = 9390823
sage: RR = RealField(200)
sage: RR(n)
9.3908230000000000000000000000000000000000000000000000000000e6
```

GCD_list()

Return the GCD of a list v of integers. Elements of v are converted to Sage integers if they aren't already.

This function is used, e.g., by rings/arith.py

INPUT:

- v - list or tuple

OUTPUT: integer

EXAMPLES:

```
sage: from sage.rings.integer import GCD_list
sage: w = GCD_list([3,9,30]); w
3
sage: type(w)
<type 'sage.rings.integer.Integer'>
```

Check that the bug reported in trac #3118 has been fixed:

```
sage: sage.rings.integer.GCD_list([2,2,3])
1
```

The inputs are converted to Sage integers.

```
sage: w = GCD_list([int(3), int(9), '30']); w
3
sage: type(w)
<type 'sage.rings.integer.Integer'>
```

class Integer()

The `Integer` class represents arbitrary precision integers. It derives from the `Element` class, so integers can be used as ring elements anywhere in Sage.

Integer() interprets numbers and strings that begin with 0 as octal numbers, and numbers and strings that begin with 0x as hexadecimal numbers.

The class `Integer` is implemented in Pyrex, as a wrapper of the GMP `mpz_t` integer type.

EXAMPLES:

```
sage: Integer(010)
8
sage: Integer(0x10)
16
sage: Integer(10)
10
sage: Integer('0x12')
18
sage: Integer('012')
10
```

additive_order()

Return the additive order of self.

EXAMPLES:

```
sage: ZZ(0).additive_order()
1
sage: ZZ(1).additive_order()
+Infinity
```

binary()

Return the binary digits of self as a string.

EXAMPLES:

```
sage: print Integer(15).binary()
1111
sage: print Integer(16).binary()
10000
sage: print Integer(169384023840928430928430982 43).binary()
1101101011101100011110001110010010100111010001101010001111111000101000000000101111000010000
```

bits()

Return the bits in self as a list, least significant first.

EXAMPLES:

```
sage: 500.bits()
[0, 0, 1, 0, 1, 1, 1, 1, 1]
sage: 11.bits()
[1, 1, 0, 1]
```

ceil()

Return the ceiling of self, which is self since self is an integer.

EXAMPLES:

```
sage: n = 6
sage: n.ceil()
6
```

conjugate()

Return the complex conjugate of this integer, which is the integer itself.

EXAMPLES: sage: n = 205 sage: n.conjugate() 205

coprime_integers()

Return the positive integers $< m$ that are coprime to self.

EXAMPLES:

```
sage: n = 8
sage: n.coprime_integers(8)
[1, 3, 5, 7]
sage: n.coprime_integers(11)
[1, 3, 5, 7, 9]
sage: n = 5; n.coprime_integers(10)
[1, 2, 3, 4, 6, 7, 8, 9]
sage: n.coprime_integers(5)
[1, 2, 3, 4]
sage: n = 99; n.coprime_integers(99)
[1, 2, 4, 5, 7, 8, 10, 13, 14, 16, 17, 19, 20, 23, 25, 26, 28, 29, 31, 32, 34, 35, 37, 38, 4
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples

ALGORITHM: Naive - compute lots of GCD's. If this isn't good enough for you, please code something better and submit a patch.

crt()

Return the unique integer between 0 and mn that is congruent to the integer modulo m and to y modulo n. We assume that m and n are coprime.

EXAMPLES:

```
sage: n = 17
sage: m = n.crt(5, 23, 11); m
247
sage: m%23
17
```

```
sage: m%11
5
```

denominator()

Return the denominator of this integer.

EXAMPLES:

```
sage: x = 5
sage: x.denominator()
1
sage: x = 0
sage: x.denominator()
1
```

digits()

Return a list of digits for self in the given base in little endian order.

The return is unspecified if self is a negative number and the digits are given.

INPUT:

- `base` - integer (default: 2)
- `digits` - optional indexable object as source for the digits
- `padto` - the minimal length of the returned list, sufficient number of zeros are added to make the list minimum that length (default: 0)

EXAMPLE:

```
sage: 17.digits()
[7, 1]
sage: 5.digits(base=2, digits=["zero","one"])
['one', 'zero', 'one']
sage: 5.digits(3)
[2, 1]
sage: 0.digits(base=10)  # 0 has 0 digits
[]
sage: 0.digits(base=2)  # 0 has 0 digits
[]
sage: 10.digits(16,'0123456789abcdef')
['a']
sage: 0.digits(16,'0123456789abcdef')
[]
sage: 0.digits(16,'0123456789abcdef',padto=1)
['0']
sage: 123.digits(base=10,padto=5)
[3, 2, 1, 0, 0]
sage: 123.digits(base=2,padto=3)       # padto is the minimal length
[1, 1, 0, 1, 1, 1, 1]
sage: 123.digits(base=2,padto=10,digits=(1,-1))
[-1, -1, 1, -1, -1, -1, -1, 1, 1, 1]
sage: a=9939082340; a.digits(10)
[0, 4, 3, 2, 8, 0, 9, 3, 9, 9]
sage: a.digits(512)
[100, 302, 26, 74]
sage: (-12).digits(10)
[-2, -1]
sage: (-12).digits(2)
[0, 0, -1, -1]
```

We support large bases

```
sage: n=2^6000
sage: n.digits(2^3000)
[0, 0, 1]

sage: base=3; n=25
sage: l=n.digits(base)
sage: # the next relationship should hold for all n,base
sage: sum(base^i*l[i] for i in range(len(l)))==n
True
sage: base=3; n=-30; l=n.digits(base); sum(base^i*l[i] for i in range(len(l)))==n
True
```

Note: In some cases it is faster to give a digits collection. This would be particularly true for computing the digits of a series of small numbers. In these cases, the code is careful to allocate as few python objects as reasonably possible.

```
sage: digits = range(15)
sage: l=[ZZ(i).digits(15,digits) for i in range(100)]
sage: l[16]
[1, 1]
```

This function is comparable to `str` for speed.

```
sage: n=3^100000
sage: n.digits(base=10)[-1]  # slightly slower than str
1
sage: n=10^10000
sage: n.digits(base=10)[-1]  # slightly faster than str
1
```

AUTHORS:

- Joel B. Mohler (2008-03-02): significantly rewrote this entire function

divide_knowing_divisible_by()

Returns the integer self / right when self is divisible by right.

If self is not divisible by right, the return value is undefined, and may not even be close to self/right for multi-word integers.

EXAMPLES:

```
sage: a = 8; b = 4
sage: a.divide_knowing_divisible_by(b)
2
sage: (100000).divide_knowing_divisible_by(25)
4000
sage: (100000).divide_knowing_divisible_by(26) # close (random)
3846
```

However, often it's way off.

```
sage: a = 2^70; a
1180591620717411303424
sage: a // 11  # floor divide
107326510974310118493
sage: a.divide_knowing_divisible_by(11) # way off and possibly random
432153614787434223889704550 40
```

divides()

Return True if self divides n.

EXAMPLES:

```
sage: Z = IntegerRing()
sage: Z(5).divides(Z(10))
True
sage: Z(0).divides(Z(5))
False
sage: Z(10).divides(Z(5))
False
```

divisors()

Returns a list of all positive integer divisors of the integer self.

EXAMPLES:

```
sage: a = -3; a.divisors()
[1, 3]
sage: a = 6; a.divisors()
[1, 2, 3, 6]
sage: a = 28; a.divisors()
[1, 2, 4, 7, 14, 28]
sage: a = 2^5; a.divisors()
[1, 2, 4, 8, 16, 32]
sage: a = 100; a.divisors()
[1, 2, 4, 5, 10, 20, 25, 50, 100]
sage: a = 1; a.divisors()
[1]
sage: a = 0; a.divisors()
...
ValueError: n must be nonzero
sage: a = 2^3 * 3^2 * 17; a.divisors()
[1, 2, 3, 4, 6, 8, 9, 12, 17, 18, 24, 34, 36, 51, 68, 72, 102, 136, 153, 204, 306, 408, 612,
sage: a = odd_part(factorial(31))
sage: v = a.divisors(); len(v)
172800
sage: prod(e+1 for p,e in factor(a))
172800
sage: all([t.divides(a) for t in v])
True
```

Note: If one first computes all the divisors and then sorts it, the sorting step can easily dominate the runtime. Note, however, that (non-negative) multiplication on the left preserves relative order. One can leverage this fact to keep the list in order as one computes it using a process similar to that of the merge sort when adding new elements.

exact_log()

Returns the largest integer k such that $m^k \leq$ self, i.e., the floor of $\log_m(\text{self})$.

This is guaranteed to return the correct answer even when the usual log function doesn't have sufficient precision.

INPUT:

- m - integer >= 2

AUTHORS:

- David Harvey (2006-09-15)
- **Joel B. Mohler (2009-04-08) – rewrote this to handle small cases** and/or easy cases up to 100x faster..

EXAMPLES:

```
sage: Integer(125).exact_log(5)
3
sage: Integer(124).exact_log(5)
```

```
2
sage: Integer(126).exact_log(5)
3
sage: Integer(3).exact_log(5)
0
sage: Integer(1).exact_log(5)
0
sage: Integer(178^1700).exact_log(178)
1700
sage: Integer(178^1700-1).exact_log(178)
1699
sage: Integer(178^1700+1).exact_log(178)
1700
sage: # we need to exercise the large base code path too
sage: Integer(1780^1700-1).exact_log(1780)
1699

sage: # The following are very very fast.
sage: # Note that for base m a perfect power of 2, we get the exact log by counting bits.
sage: n=29835798237501857013751098 35; m=32
sage: n.exact_log(m)
18
sage: # The next is a favorite of mine.  The log2 approximate is exact and immediately prova
sage: n=901537105709127095179025790107932517092579012709417092479012097421 24;m=2135097213095
sage: n.exact_log(m)
4

sage: x = 3^100000
sage: RR(log(RR(x), 3))
100000.000000000
sage: RR(log(RR(x + 100000), 3))
100000.000000000

sage: x.exact_log(3)
100000
sage: (x+1).exact_log(3)
100000
sage: (x-1).exact_log(3)
99999

sage: x.exact_log(2.5)
...
TypeError: Attempt to coerce non-integral RealNumber to Integer
```

exp()

Returns the exponential function of self as a real number.

This function is provided only so that Sage integers may be treated in the same manner as real numbers when convenient.

INPUT:

- `prec` - integer (default: None): if None, returns symbolic, else to given bits of precision as in RealField

EXAMPLES:

```
sage: Integer(8).exp()
e^8
sage: Integer(8).exp(prec=100)
2980.9579870417282747435920995
```

```
sage: exp(Integer(8))
e^8
```

For even fairly large numbers, this may not be useful.

```
sage: y=Integer(145^145)
sage: y.exp()
e^25024207011349079210459585279553675697932183658421565260323592409432707306554163224876110
sage: y.exp(prec=53) # default RealField precision
+infinity
```

factor()

Return the prime factorization of the integer as a list of pairs (p, e), where p is prime and e is a positive integer.

INPUT:

- `algorithm` - string
  - `'pari'` - (default) use the PARI c library
  - `'kash'` - use KASH computer algebra system (requires the optional kash package be installed)
- `proof` - bool (default: True) whether or not to prove primality of each factor (only applicable for PARI).
- `limit` - int or None (default: None) if limit is given it must fit in a signed int, and the factorization is done using trial division and primes up to limit.

EXAMPLES:

```
sage: n = 2^100 - 1; n.factor()
3 * 5^3 * 11 * 31 * 41 * 101 * 251 * 601 * 1801 * 4051 * 8101 * 268501
```

We use proof=False, which doesn't prove correctness of the primes that appear in the factorization:

```
sage: n = 92038409284239042384829034820394809238408234908 2
sage: n.factor(proof=False)
2 * 11 * 1531 * 4402903 * 10023679 * 619162955472170540533894518173
sage: n.factor(proof=True)
2 * 11 * 1531 * 4402903 * 10023679 * 619162955472170540533894518173
```

We factor using trial division only:

```
sage: n.factor(limit=1000)
2 * 11 * 41835640583745019265831379463815822381094652231
```

factorial()

Return the factorial $n! = 1 \cdot 2 \cdot 3 \cdots n$. Self must fit in an `unsigned long int`.

EXAMPLES:

```
sage: for n in srange(7):
...       print n, n.factorial()
0 1
1 1
2 2
3 6
4 24
5 120
6 720
```

floor()

Return the floor of self, which is just self since self is an integer.

EXAMPLES:

```
sage: n = 6
sage: n.floor()
6
```

gamma()

The gamma function on integers is the factorial function (shifted by one) on positive integers, and $\pm\infty$ on non-positive integers.

EXAMPLES:

```
sage: gamma(5)
24
sage: gamma(0)
Infinity
sage: gamma(-1)
Infinity
sage: gamma(-2^150)
Infinity
```

gcd()

Return the greatest common divisor of self and n.

EXAMPLE:

```
sage: gcd(-1,1)
1
sage: gcd(0,1)
1
sage: gcd(0,0)
0
sage: gcd(2,2^6)
2
sage: gcd(21,2^6)
1
```

inverse_mod()

Returns the inverse of self modulo n, if this inverse exists. Otherwise, raises a `ZeroDivisionError` exception.

INPUT:

- `self` - Integer
- `n` - Integer

OUTPUT:

- `x` - Integer such that x*self = 1 (mod m), or raises ZeroDivisionError.

IMPLEMENTATION:

Call the mpz_invert GMP library function.

EXAMPLES:

```
sage: a = Integer(189)
sage: a.inverse_mod(10000)
4709
sage: a.inverse_mod(-10000)
4709
sage: a.inverse_mod(1890)
...
ZeroDivisionError: Inverse does not exist.
sage: a = Integer(19)**100000
sage: b = a*a
sage: c = a.inverse_mod(b)
```

```
...
ZeroDivisionError: Inverse does not exist.
```

inverse_of_unit()

Return inverse of self if self is a unit in the integers, i.e., self is -1 or 1. Otherwise, raise a ZeroDivisionError.

EXAMPLES:

```
sage: (1).inverse_of_unit()
1
sage: (-1).inverse_of_unit()
-1
sage: 5.inverse_of_unit()
...
ZeroDivisionError: Inverse does not exist.
sage: 0.inverse_of_unit()
...
ZeroDivisionError: Inverse does not exist.
```

is_integral()

Return `True` since integers are integral, i.e., satisfy a monic polynomial with integer coefficients.

EXAMPLES:

```
sage: Integer(3).is_integral()
True
```

is_irreducible()

Returns `True` if self is irreducible, i.e. +/- prime

EXAMPLES:

```
sage: z = 2^31 - 1
sage: z.is_irreducible()
True
sage: z = 2^31
sage: z.is_irreducible()
False
sage: z = 7
sage: z.is_irreducible()
True
sage: z = -7
sage: z.is_irreducible()
True
```

is_one()

Returns `True` if the integer is 1, otherwise `False`.

EXAMPLES:

```
sage: Integer(1).is_one()
True
sage: Integer(0).is_one()
False
```

is_perfect_power()

Returns `True` if self is a perfect power.

EXAMPLES:

```
sage: z = 8
sage: z.is_perfect_power()
True
sage: 144.is_perfect_power()
```

```
True
sage: 10.is_perfect_power()
False
sage: (-8).is_perfect_power()
True
sage: (-4).is_perfect_power()
False
```

This is a test to make sure we workaround a bug in GMP. (See trac #4612.)

```
sage: [ -a for a in srange(100) if not (-a^3).is_perfect_power() ]
[]
```

is_power()

Returns `True` if self is a perfect power, ie if there exist integers a and b, $b > 1$ with $self = a^b$.

EXAMPLES:

```
sage: Integer(-27).is_power()
True
sage: Integer(12).is_power()
False
```

is_power_of()

Returns `True` if there is an integer b with `self` $= n^b$.

EXAMPLES:

```
sage: Integer(64).is_power_of(4)
True
sage: Integer(64).is_power_of(16)
False
```

TESTS:

```
sage: Integer(-64).is_power_of(-4)
True
sage: Integer(-32).is_power_of(-2)
True
sage: Integer(1).is_power_of(1)
True
sage: Integer(-1).is_power_of(-1)
True
sage: Integer(0).is_power_of(1)
False
sage: Integer(0).is_power_of(0)
True
sage: Integer(1).is_power_of(0)
True
sage: Integer(1).is_power_of(8)
True
sage: Integer(-8).is_power_of(2)
False
```

Note: For large integers self, is_power_of() is faster than is_power(). The following examples gives some indication of how much faster.

```
sage: b = lcm(range(1,10000))
sage: b.exact_log(2)
14446
sage: t=cputime()
sage: for a in range(2, 1000): k = b.is_power()
sage: cputime(t)      # random
```

```
0.53203299999999976
sage: t=cputime()
sage: for a in range(2, 1000): k = b.is_power_of(2)
sage: cputime(t)      # random
0.0
sage: t=cputime()
sage: for a in range(2, 1000): k = b.is_power_of(3)
sage: cputime(t)      # random
0.032002000000000308

sage: b = lcm(range(1, 1000))
sage: b.exact_log(2)
1437
sage: t=cputime()
sage: for a in range(2, 10000): k = b.is_power() # note that we change the range from the ex
sage: cputime(t)      # random
0.17201100000000036
sage: t=cputime(); TWO=int(2)
sage: for a in range(2, 10000): k = b.is_power_of(TWO)
sage: cputime(t)      # random
0.0040000000000000036
sage: t=cputime()
sage: for a in range(2, 10000): k = b.is_power_of(3)
sage: cputime(t)      # random
0.040003000000000011
sage: t=cputime()
sage: for a in range(2, 10000): k = b.is_power_of(a)
sage: cputime(t)      # random
0.02800199999999986
```

is_prime()

Returns `True` if self is prime.

Note: Integer primes are by definition *positive*! This is different than Magma, but the same as in Pari. See also the `is_irreducible`() method.

EXAMPLES:

```
sage: z = 2^31 - 1
sage: z.is_prime()
True
sage: z = 2^31
sage: z.is_prime()
False
sage: z = 7
sage: z.is_prime()
True
sage: z = -7
sage: z.is_prime()
False
sage: z.is_irreducible()
True
```

is_prime_power()

Returns True if x is a prime power, and False otherwise.

INPUT:

- `flag` (for primality testing) - int
  - `0` (default): use a combination of algorithms.
  - `1`: certify primality using the Pocklington-Lehmer Test.

-2: certify primality using the APRCL test.

EXAMPLES:

```
sage: (-10).is_prime_power()
False
sage: (10).is_prime_power()
False
sage: (64).is_prime_power()
True
sage: (3^10000).is_prime_power()
True
sage: (10000).is_prime_power(flag=1)
False
```

Note: Currently in the case when self is a perfect power, we call self.factor, due to a bug in Pari's ispower function. See Trac #4777. We illustrate that this is fixed below.

```
sage: n = 150607571^14
sage: n.is_prime_power()
True
```

is_pseudoprime()

Returns True if self is a pseudoprime

EXAMPLES:

```
sage: z = 2^31 - 1
sage: z.is_pseudoprime()
True
sage: z = 2^31
sage: z.is_pseudoprime()
False
```

is_square()

Returns True if self is a perfect square

EXAMPLES:

```
sage: Integer(4).is_square()
True
sage: Integer(41).is_square()
False
```

is_squarefree()

Returns True if this integer is not divisible by the square of any prime and False otherwise.

EXAMPLES:

```
sage: Integer(100).is_squarefree()
False
sage: Integer(102).is_squarefree()
True
```

is_unit()

Returns true if this integer is a unit, i.e., 1 or -1.

EXAMPLES:

```
sage: for n in srange(-2,3):
...       print n, n.is_unit()
-2 False
-1 True
0 False
1 True
2 False
```

isqrt()

Returns the integer floor of the square root of self, or raises an `ValueError` if self is negative.

EXAMPLE:

```
sage: a = Integer(5)
sage: a.isqrt()
2

sage: Integer(-102).isqrt()
...
ValueError: square root of negative integer not defined.
```

jacobi()

Calculate the Jacobi symbol $\left(\frac{self}{b}\right)$.

EXAMPLES:

```
sage: z = -1
sage: z.jacobi(17)
1
sage: z.jacobi(19)
-1
sage: z.jacobi(17*19)
-1
sage: (2).jacobi(17)
1
sage: (3).jacobi(19)
-1
sage: (6).jacobi(17*19)
-1
sage: (6).jacobi(33)
0
sage: a = 3; b = 7
sage: a.jacobi(b) == -b.jacobi(a)
True
```

kronecker()

Calculate the Kronecker symbol $\left(\frac{self}{b}\right)$ with the Kronecker extension $(self/2) = (2/self)$ when self odd, or $(self/2) = 0$ when $self$ even.

EXAMPLES:

```
sage: z = 5
sage: z.kronecker(41)
1
sage: z.kronecker(43)
-1
sage: z.kronecker(8)
-1
sage: z.kronecker(15)
0
sage: a = 2; b = 5
sage: a.kronecker(b) == b.kronecker(a)
True
```

list()

Return a list with this integer in it, to be compatible with the method for number fields.

EXAMPLES:

```
sage: m = 5
sage: m.list()
[5]
```

log()

Returns symbolic log by default, unless the logarithm is exact (for an integer base). When precision is given, the RealField approximation to that bit precision is used.

This function is provided primarily so that Sage integers may be treated in the same manner as real numbers when convenient. Direct use of exact_log is probably best for arithmetic log computation.

INPUT:

- `m` - default: natural log base e
- `prec` - integer (default: None): if None, returns symbolic, else to given bits of precision as in RealField

EXAMPLES:

```
sage: Integer(124).log(5)
log(124)/log(5)
sage: Integer(124).log(5,100)
2.9950093311241087454822446806
sage: Integer(125).log(5)
3
sage: Integer(125).log(5,prec=53)
3.00000000000000
sage: log(Integer(125))
log(125)
```

For extremely large numbers, this works:

```
sage: x = 3^100000
sage: log(x,3)
100000
```

Do NOT try log(x) here, at least not with current Maxima symbolic backend; it may take a very, very long time. But approximations work well - up to the given precision, of course.

```
sage: x.log(3,53) # default precision for RealField
100000.000000000
sage: (x+1).log(3,53)
100000.000000000
sage: (x+1).log(3,1000)
100000.00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

We can use non-integer bases, with default e:

```
sage: x.log(2.5,prec=53)
119897.784671579
```

multifactorial()

Computes the k-th factorial $n!^{(k)}$ of self. For k=1 this is the standard factorial, and for k greater than one it is the product of every k-th terms down from self to k. The recursive definition is used to extend this function to the negative integers.

EXAMPLES:

```
sage: 5.multifactorial(1)
120
sage: 5.multifactorial(2)
15
sage: 23.multifactorial(2)
316234143225
```

```
sage: prod([1..23, step=2])
316234143225
sage: (-29).multifactorial(7)
1/2640
```

multiplicative_order()

Return the multiplicative order of self.

EXAMPLES:

```
sage: ZZ(1).multiplicative_order()
1
sage: ZZ(-1).multiplicative_order()
2
sage: ZZ(0).multiplicative_order()
+Infinity
sage: ZZ(2).multiplicative_order()
+Infinity
```

nbits()

Return the number of bits in self.

EXAMPLES:

```
sage: 500.nbits()
9
sage: 5.nbits()
3
sage: 12345.nbits() == len(12345.binary())
True
```

ndigits()

Return the number of digits of self expressed in the given base.

INPUT:

- `base` - integer (default: 10)

EXAMPLES:

```
sage: n = 52
sage: n.ndigits()
2
sage: n = -10003
sage: n.ndigits()
5
sage: n = 15
sage: n.ndigits(2)
4
sage: n=1000**1000000+1
sage: n.ndigits()
3000001
sage: n=1000**1000000-1
sage: n.ndigits()
3000000
sage: n=10**10000000-10**9999990
sage: n.ndigits()
10000000
```

next_prime()

Returns the next prime after self.

INPUT:

- `proof` - bool or None (default: None, see proof.arithmetic or sage.structure.proof) Note that the global Sage default is proof=True

EXAMPLES:

```
sage: Integer(100).next_prime()
101
```

Use Proof = False, which is way faster:

```
sage: b = (2^1024).next_prime(proof=False)

sage: Integer(0).next_prime()
2
sage: Integer(1001).next_prime()
1009
```

next_probable_prime()

Returns the next probable prime after self, as determined by PARI.

EXAMPLES:

```
sage: (-37).next_probable_prime()
2
sage: (100).next_probable_prime()
101
sage: (2^512).next_probable_prime()
13407807929942597099574024998205846127479365820592393377723561443721764030073546976801874298
sage: 0.next_probable_prime()
2
sage: 126.next_probable_prime()
127
sage: 144168.next_probable_prime()
144169
```

nth_root()

Returns the (possibly truncated) n'th root of self.

INPUT:

- `n` - integer >= 1 (must fit in C int type).
- `truncate_mode` - boolean, whether to allow truncation if self is not an n'th power.

OUTPUT: If truncate_mode is 0 (default), then returns the exact n'th root if self is an n'th power, or raises a ValueError if it is not.

If truncate_mode is 1, then if either n is odd or self is positive, returns a pair (root, exact_flag) where root is the truncated nth root (rounded towards zero) and exact_flag is a boolean indicating whether the root extraction was exact; otherwise raises a ValueError.

AUTHORS:

- David Harvey (2006-09-15)
- Interface changed by John Cremona (2009-04-04)

EXAMPLES:

```
sage: Integer(125).nth_root(3)
5
sage: Integer(124).nth_root(3)
...
ValueError: 124 is not a 3rd power
sage: Integer(124).nth_root(3, truncate_mode=1)
(4, False)
sage: Integer(125).nth_root(3, truncate_mode=1)
(5, True)
```

```
sage: Integer(126).nth_root(3, truncate_mode=1)
(5, False)

sage: Integer(-125).nth_root(3)
-5
sage: Integer(-125).nth_root(3,truncate_mode=1)
(-5, True)
sage: Integer(-124).nth_root(3,truncate_mode=1)
(-4, False)
sage: Integer(-126).nth_root(3,truncate_mode=1)
(-5, False)

sage: Integer(125).nth_root(2, True)
(11, False)
sage: Integer(125).nth_root(3, True)
(5, True)

sage: Integer(125).nth_root(-5)
...
ValueError: n (=-5) must be positive

sage: Integer(-25).nth_root(2)
...
ValueError: cannot take even root of negative number

sage: a=9
sage: a.nth_root(3)
...
ValueError: 9 is not a 3rd power

sage: a.nth_root(22)
...
ValueError: 9 is not a 22nd power

sage: ZZ(2^20).nth_root(21)
...
ValueError: 1048576 is not a 21st power

sage: ZZ(2^20).nth_root(21, truncate_mode=1)
(1, False)
```

numerator()

Return the numerator of this integer.

EXAMPLE:

```
sage: x = 5
sage: x.numerator()
5

sage: x = 0
sage: x.numerator()
0
```

ord()

Synonym for valuation

EXAMPLES:

```
sage: n=12
sage: n.ord(3)
1
```

ordinal_str()

Returns a string representation of the ordinal associated to self.

EXAMPLES:

```
sage: [ZZ(n).ordinal_str() for n in range(25)]
['0th',
'1st',
'2nd',
'3rd',
'4th',
...
'10th',
'11th',
'12th',
'13th',
'14th',
...
'20th',
'21st',
'22nd',
'23rd',
'24th']

sage: ZZ(1001).ordinal_str()
'1001st'
```

powermod()

Compute self**exp modulo mod.

EXAMPLES:

```
sage: z = 2
sage: z.powermod(31,31)
2
sage: z.powermod(0,31)
1
sage: z.powermod(-31,31) == 2^-31 % 31
True
```

As expected, the following is invalid:

```
sage: z.powermod(31,0)
...
ZeroDivisionError: cannot raise to a power modulo 0
```

powermodm_ui()

Computes self**exp modulo mod, where exp is an unsigned long integer.

EXAMPLES:

```
sage: z = 32
sage: z.powermodm_ui(2, 4)
0
sage: z.powermodm_ui(2, 14)
2
sage: z.powermodm_ui(2^32-2, 14)
2
```

```
sage: z.powermodm_ui(2^32-1, 14)
...                                                                 # 32-bit
OverflowError: exp (=4294967295) must be <= 4294967294              # 32-bit
8               # 64-bit
sage: z.powermodm_ui(2^65, 14)
...
OverflowError: exp (=36893488147419103232) must be <= 4294967294  # 32-bit
OverflowError: exp (=36893488147419103232) must be <= 18446744073709551614     # 64-bit
```

prime_divisors()

The prime divisors of self, sorted in increasing order. If n is negative, we do *not* include -1 among the prime divisors, since -1 is not a prime number.

EXAMPLES:

```
sage: a = 1; a.prime_divisors()
[]
sage: a = 100; a.prime_divisors()
[2, 5]
sage: a = -100; a.prime_divisors()
[2, 5]
sage: a = 2004; a.prime_divisors()
[2, 3, 167]
```

prime_factors()

The prime divisors of self, sorted in increasing order. If n is negative, we do *not* include -1 among the prime divisors, since -1 is not a prime number.

EXAMPLES:

```
sage: a = 1; a.prime_divisors()
[]
sage: a = 100; a.prime_divisors()
[2, 5]
sage: a = -100; a.prime_divisors()
[2, 5]
sage: a = 2004; a.prime_divisors()
[2, 3, 167]
```

prime_to_m_part()

Returns the prime-to-m part of self, i.e., the largest divisor of self that is coprime to m.

INPUT:

- `m` - Integer

OUTPUT: Integer

EXAMPLES:

```
sage: z = 43434
sage: z.prime_to_m_part(20)
21717
```

quo_rem()

Returns the quotient and the remainder of self divided by other. Note that the remainder returned is always either zero or of the same sign as other.

INPUT:

- `other` - the integer the divisor

OUTPUT:

- `q` - the quotient of self/other
- `r` - the remainder of self/other

EXAMPLES:

```
sage: z = Integer(231)
sage: z.quo_rem(2)
(115, 1)
sage: z.quo_rem(-2)
(-116, -1)
sage: z.quo_rem(0)
...
ZeroDivisionError: other (=0) must be nonzero
```

radical()

Return the product of the prime divisors of self.

If self is 0, returns 1.

EXAMPLES:

```
sage: Integer(10).radical()
10
sage: Integer(20).radical()
10
sage: Integer(-20).radical()
-10
sage: Integer(0).radical()
1
sage: Integer(36).radical()
6
```

rational_reconstruction()

Return the rational reconstruction of this integer modulo m, i.e., the unique (if it exists) rational number that reduces to self modulo m and whose numerator and denominator is bounded by sqrt(m/2).

EXAMPLES:

```
sage: (3/7)%100
29
sage: (29).rational_reconstruction(100)
3/7
```

sqrt()

The square root function.

INPUT:

- `prec` - integer (default: None): if None, returns an exact square root; otherwise returns a numerical square root if necessary, to the given bits of precision.
- `extend` - bool (default: True); if True, return a square root in an extension ring, if necessary. Otherwise, raise a ValueError if the square is not in the base ring. Ignored if prec is not None.
- `all` - bool (default: False); if True, return all square roots of self, instead of just one.

EXAMPLES:

```
sage: Integer(144).sqrt()
12
sage: sqrt(Integer(144))
12
sage: Integer(102).sqrt()
sqrt(102)

sage: n = 2
sage: n.sqrt(all=True)
[sqrt(2), -sqrt(2)]
sage: n.sqrt(prec=10)
```

```
1.4
sage: n.sqrt(prec=100)
1.4142135623730950488016887242
sage: n.sqrt(prec=100,all=True)
[1.4142135623730950488016887242, -1.4142135623730950488016887242]
sage: n.sqrt(extend=False)
...
ValueError: square root of 2 not an integer
sage: Integer(144).sqrt(all=True)
[12, -12]
sage: Integer(0).sqrt(all=True)
[0]
sage: type(Integer(5).sqrt())
<type 'sage.symbolic.expression.Expression'>
sage: type(Integer(5).sqrt(prec=53))
<type 'sage.rings.real_mpfr.RealNumber'>
sage: type(Integer(-5).sqrt(prec=53))
<type 'sage.rings.complex_number.ComplexNumber'>
```

sqrt_approx()

EXAMPLES:

```
sage: 5.sqrt_approx(prec=200)
doctest:1172: DeprecationWarning: This function is deprecated.  Use sqrt with a given number
2.2360679774997896964091736687312762354406183596115257242709
sage: 5.sqrt_approx()
2.23606797749979
sage: 4.sqrt_approx()
2
```

sqrtrem()

Return (s, r) where s is the integer square root of self and r is the remainder such that self $= s^2 + r$. Raises `ValueError` if self is negative.

EXAMPLES:

```
sage: 25.sqrtrem()
(5, 0)
sage: 27.sqrtrem()
(5, 2)
sage: 0.sqrtrem()
(0, 0)

sage: Integer(-102).sqrtrem()
...
ValueError: square root of negative integer not defined.
```

squarefree_part()

Return the square free part of x (=self), i.e., the unique integer z that $x = zy^2$, with y^2 a perfect square and z square-free.

Use `self.radical()` for the product of the primes that divide self.

If self is 0, just returns 0.

EXAMPLES:

```
sage: squarefree_part(100)
1
sage: squarefree_part(12)
3
sage: squarefree_part(17*37*37)
```

```
17
sage: squarefree_part(-17*32)
-34
sage: squarefree_part(1)
1
sage: squarefree_part(-1)
-1
sage: squarefree_part(-2)
-2
sage: squarefree_part(-4)
-1

sage: a = 8 * 5^6 * 101^2
sage: a.squarefree_part(bound=2).factor()
2 * 5^6 * 101^2
sage: a.squarefree_part(bound=5).factor()
2 * 101^2
sage: a.squarefree_part(bound=1000)
2
sage: a = 7^3 * next_prime(2^100)^2 * next_prime(2^200)
sage: a / a.squarefree_part(bound=1000)
49
```

str()

Return the string representation of `self` in the given base.

EXAMPLES:

```
sage: Integer(2^10).str(2)
'10000000000'
sage: Integer(2^10).str(17)
'394'

sage: two=Integer(2)
sage: two.str(1)
...
ValueError: base (=1) must be between 2 and 36

sage: two.str(37)
...
ValueError: base (=37) must be between 2 and 36

sage: big = 10^5000000
sage: s = big.str()                    # long time (> 20 seconds)
sage: len(s)                           # long time (depends on above defn of s)
5000001
sage: s[:10]                           # long time (depends on above defn of s)
'1000000000'
```

support()

Return a sorted list of the primes dividing this integer.

OUTPUT: The sorted list of primes appearing in the factorization of this rational with positive exponent.

EXAMPLES:

```
sage: factorial(10).support()
[2, 3, 5, 7]
sage: (-999).support()
[3, 37]
```

Trying to find the support of 0 gives an arithmetic error:

```
sage: 0.support()
...
ArithmeticError: Support of 0 not defined.
```

test_bit()

Return the bit at index.

EXAMPLES:

```
sage: w = 6
sage: w.str(2)
'110'
sage: w.test_bit(2)
1
sage: w.test_bit(-1)
0
```

trailing_zero_bits()

Return the number of trailing zero bits in self, i.e. the exponent of the largest power of 2 dividing self.

EXAMPLES:

```
sage: 11.trailing_zero_bits()
0
sage: (-11).trailing_zero_bits()
0
sage: (11<<5).trailing_zero_bits()
5
sage: (-11<<5).trailing_zero_bits()
5
sage: 0.trailing_zero_bits()
0
```

val_unit()

Returns a pair: the p-adic valuation of self, and the p-adic unit of self.

INPUT:

- p - an integer at least 2.

OUTPUT:

- v_p(self) - the p-adic valuation of self
- u_p(self) - self / $p^{v_p(\text{self})}$

EXAMPLE:

```
sage: n = 60
sage: n.val_unit(2)
(2, 15)
sage: n.val_unit(3)
(1, 20)
sage: n.val_unit(7)
(0, 60)
sage: (2^11).val_unit(4)
(5, 2)
sage: 0.val_unit(2)
(+Infinity, 1)
```

valuation()

Return the p-adic valuation of self.

INPUT:

- p - an integer at least 2.

EXAMPLE:

```
sage: n = 60
sage: n.valuation(2)
2
sage: n.valuation(3)
1
sage: n.valuation(7)
0
sage: n.valuation(1)
...
ValueError: You can only compute the valuation with respect to a integer larger than 1.
```

We do not require that p is a prime:

```
sage: (2^11).valuation(4)
5
```

class IntegerWrapper()

Python classes have problems inheriting from Integer directly, but they don't have issues with inheriting from IntegerWrapper.

LCM_list()

Return the LCM of a list v of integers. Elements of v are converted to Sage integers if they aren't already.

This function is used, e.g., by rings/arith.py

INPUT:

- v - list or tuple

OUTPUT: integer

EXAMPLES:

```
sage: from sage.rings.integer import LCM_list
sage: w = LCM_list([3,9,30]); w
90
sage: type(w)
<type 'sage.rings.integer.Integer'>
```

The inputs are converted to Sage integers.

```
sage: w = LCM_list([int(3), int(9), '30']); w
90
sage: type(w)
<type 'sage.rings.integer.Integer'>
```

clear_mpz_globals()

free_integer_pool()

gmp_randrange()

init_mpz_globals()

class int_to_Z()

is_Integer()

Return true if x is of the Sage integer type.

EXAMPLES:

```
sage: from sage.rings.integer import is_Integer
sage: is_Integer(2)
True
sage: is_Integer(2/1)
False
sage: is_Integer(int(2))
False
sage: is_Integer(long(2))
False
sage: is_Integer('5')
False
```

class long_to_Z()

EXAMPLES:

```
sage: f = ZZ.coerce_map_from(long); f
Native morphism:
  From: Set of Python objects of type 'long'
  To:   Integer Ring
sage: f(1rL)
1
sage: f(-100000000000000000000001r)
-100000000000000000000001
```

make_integer()

Create a Sage integer from the base-32 Python *string* s. This is used in unpickling integers.

EXAMPLES:

```
sage: from sage.rings.integer import make_integer
sage: make_integer('-29')
-73
sage: make_integer(29)
...
TypeError: expected string or Unicode object, sage.rings.integer.Integer found
```

parent()

24.3 Ring $\mathbf{Z}/n\mathbf{Z}$ of integers modulo n

EXAMPLES:

```
sage: R = Integers(97)
sage: a = R(5)
sage: a**1000000000000000000000000000000000000000000000000000000000000000000000000000000
61
```

This example illustrates the relation between $\mathbf{Z}/p\mathbf{Z}$ and $\mathbf{F}_p$. In particular, there is a canonical map to $\mathbf{F}_p$, but not in the other direction.

```
sage: r = Integers(7)
sage: s = GF(7)
sage: r.has_coerce_map_from(s)
False
sage: s.has_coerce_map_from(r)
```

```
True
sage: s(1) + r(1)
2
sage: parent(s(1) + r(1))
Finite Field of size 7
sage: parent(r(1) + s(1))
Finite Field of size 7
```

We list the elements of $\mathbf{Z}/3\mathbf{Z}$

```
sage: R = Integers(3)
sage: list(R)
[0, 1, 2]
```

AUTHORS:

- William Stein (initial code)
- David Joyner (2005-12-22): most examples
- Robert Bradshaw (2006-08-24): convert to SageX (Cython)
- William Stein (2007-04-29): square_roots_of_one

class **IntegerModFactory**()

Return the quotient ring $\mathbf{Z}/n\mathbf{Z}$.

INPUT:

•`order` - integer (default: 0), positive or negative

EXAMPLES:

```
sage: IntegerModRing(15)
Ring of integers modulo 15
sage: IntegerModRing(7)
Ring of integers modulo 7
sage: IntegerModRing(-100)
Ring of integers modulo 100
```

Note that you can also use `Integers`, which is a synonym for `IntegerModRing`.

```
sage: Integers(18)
Ring of integers modulo 18
sage: Integers() is Integers(0) is ZZ
True
```

create_key(*order=0*)

create_object(*version, order*)

EXAMPLES:

```
sage: R = Integers(10)
sage: loads(dumps(R)) is R
True
```

class **IntegerModRing_generic**(*order, cache=None*)

The ring of integers modulo N, with N composite.

EXAMPLES:

```
sage: R = IntegerModRing(97)
sage: a = R(5)
sage: a**(10^62)
61
```

cardinality()

characteristic()

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: FF = IntegerModRing(17)
sage: FF.characteristic()
17
sage: R.characteristic()
18
```

coerce_map_from_impl(S)

EXAMPLES:

```
sage: R = Integers(15)
sage: f = R.coerce_map_from(Integers(450)); f
Natural morphism:
  From: Ring of integers modulo 450
  To:   Ring of integers modulo 15
sage: f(-1)
14
sage: f = R.coerce_map_from(int); f
Native morphism:
  From: Set of Python objects of type 'int'
  To:   Ring of integers modulo 15
sage: f(-1r)
14
sage: f = R.coerce_map_from(ZZ); f
Natural morphism:
  From: Integer Ring
  To:   Ring of integers modulo 15
sage: f(-1)
14
sage: f = R.coerce_map_from(Integers(10)); print f
None
sage: f = R.coerce_map_from(QQ); print f
None
```

factored_order()

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: FF = IntegerModRing(17)
sage: R.factored_order()
2 * 3^2
sage: FF.factored_order()
17
```

field()

If this ring is a field, return the corresponding field as a finite field, which may have extra functionality and structure. Otherwise, raise a ValueError.

EXAMPLES:

```
sage: R = Integers(7); R
Ring of integers modulo 7
sage: R.field()
Finite Field of size 7
sage: R = Integers(9)
sage: R.field()
...
ValueError: self must be a field
```

is_field()

Return True precisely if the order is prime.

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: R.is_field()
False
sage: FF = IntegerModRing(17)
sage: FF.is_field()
True
```

is_finite()

Return True since Z/NZ is finite for all positive N.

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: R.is_finite()
True
```

is_integral_domain()

Return True if and only if the order of self is prime.

EXAMPLES:

```
sage: Integers(389).is_integral_domain()
True
sage: Integers(389^2).is_integral_domain()
False
```

is_noetherian()

EXAMPLES:

```
sage: Integers(8).is_noetherian()
True
```

krull_dimension()

EXAMPLES:

```
sage: Integers(18).krull_dimension()
0
```

list_of_elements_of_multiplicative_group()

modulus()

Return the polynomial $x - 1$ over this ring.

Note: This function exists for consistency with the finite-field modulus function.

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: R.modulus()
x + 17
sage: R = IntegerModRing(17)
```

```
sage: R.modulus()
x + 16
```

multiplicative_generator()

Return a generator for the multiplicative group of this ring, assuming the multiplicative group is cyclic. Use the unit_gens function to obtain generators even in the non-cyclic case.

EXAMPLES:

```
sage: R = Integers(7); R
Ring of integers modulo 7
sage: R.multiplicative_generator()
3
sage: R = Integers(9)
sage: R.multiplicative_generator()
2
sage: Integers(8).multiplicative_generator()
...
ValueError: multiplicative group of this ring is not cyclic
sage: Integers(4).multiplicative_generator()
3
sage: Integers(25*3).multiplicative_generator()
...
ValueError: multiplicative group of this ring is not cyclic
sage: Integers(25*3).unit_gens()
[26, 52]
sage: Integers(162).unit_gens()
[83]
```

multiplicative_group_is_cyclic()

Return True if the multiplicative group of this field is cyclic. This is the case exactly when the order is less than 8, a power of an odd prime, or twice a power of an odd prime.

EXAMPLES:

```
sage: R = Integers(7); R
Ring of integers modulo 7
sage: R.multiplicative_group_is_cyclic()
True
sage: R = Integers(9)
sage: R.multiplicative_group_is_cyclic()
True
sage: Integers(8).multiplicative_group_is_cyclic()
False
sage: Integers(4).multiplicative_group_is_cyclic()
True
sage: Integers(25*3).multiplicative_group_is_cyclic()
False
```

We test that #5250 is fixed:

```
sage: Integers(162).multiplicative_group_is_cyclic()
True
```

multiplicative_subgroups()

Return generators for each subgroup of $(\mathbf{Z}/N\mathbf{Z})^*$.

EXAMPLES:

```
sage: Integers(5).multiplicative_subgroups()
[[2], [4], []]
sage: Integers(15).multiplicative_subgroups()
```

```
[[11, 7], [4, 11], [8], [11], [14], [7], [4], []]
sage: Integers(2).multiplicative_subgroups()
[[]]
sage: len(Integers(341).multiplicative_subgroups())
80
```

order()

quadratic_nonresidue()

Return a quadratic non-residue in self.

EXAMPLES:

```
sage: R = Integers(17)
sage: R.quadratic_nonresidue()
3
sage: R(3).is_square()
False
```

random_element(*bound=None*)

Return a random element of this ring.

If bound is not None, return the coercion of an integer in the interval [-bound, bound] into this ring.

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: R.random_element()
2
```

square_roots_of_one()

Return all square roots of 1 in self, i.e., all solutions to $x^2 - 1 = 0$.

OUTPUT:

- `tuple` - the square roots of 1 in self.

EXAMPLES:

```
sage: R = Integers(2^10)
sage: [x for x in R if x^2 == 1]
[1, 511, 513, 1023]
sage: R.square_roots_of_one()
(1, 511, 513, 1023)

sage: v = Integers(9*5).square_roots_of_one(); v
(1, 19, 26, 44)
sage: [x^2 for x in v]
[1, 1, 1, 1]
sage: v = Integers(9*5*8).square_roots_of_one(); v
(1, 19, 71, 89, 91, 109, 161, 179, 181, 199, 251, 269, 271, 289, 341, 359)
sage: [x^2 for x in v]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

unit_gens()

Returns generators for the unit group $(\mathbf{Z}/N\mathbf{Z})^*$.

We compute the list of generators using a deterministic algorithm, so the generators list will always be the same. For each odd prime divisor of N there will be exactly one corresponding generator; if N is even there will be 0, 1 or 2 generators according to whether 2 divides N to order 1, 2 or ≥ 3.

INPUT: (none)

OUTPUT:

- `list` - a list of elements of self

EXAMPLES:

```
sage: R = IntegerModRing(18)
sage: R.unit_gens()
[11]
sage: R = IntegerModRing(17)
sage: R.unit_gens()
[3]
sage: IntegerModRing(next_prime(10^30)).unit_gens()
[5]
```

unit_group_exponent()
EXAMPLES:

```
sage: R = IntegerModRing(17)
sage: R.unit_group_exponent()
16
sage: R = IntegerModRing(18)
sage: R.unit_group_exponent()
6
```

unit_group_order()
Return the order of the unit group of this residue class ring.
EXAMPLES;

```
sage: R = Integers(500)
sage: R.unit_group_order()
200
```

crt(*v*)
INPUT: v - (list) a lift of elements of rings.IntegerMod(n), for various coprime moduli n.

is_IntegerModRing(*x*)
Return True if x is an integer modulo ring.
EXAMPLES:

```
sage: from sage.rings.integer_mod_ring import is_IntegerModRing
sage: R = IntegerModRing(17)
sage: is_IntegerModRing(R)
True
sage: is_IntegerModRing(GF(13))
True
sage: is_IntegerModRing(GF(4, 'a'))
False
sage: is_IntegerModRing(10)
False
sage: is_IntegerModRing(ZZ)
False
```

24.4 Elements of $\mathbb{Z}/n\mathbb{Z}$

An element of the integers modulo n.

There are three types of integer_mod classes, depending on the size of the modulus.

- `IntegerMod_int` stores its value in a `int_fast32_t` (typically an `int`); this is used if the modulus is less than $\sqrt{2^{31}-1}$.

- `IntegerMod_int64` stores its value in a `int_fast64_t` (typically a `long long`); this is used if the modulus is less than $2^{31} - 1$.
- `IntegerMod_gmp` stores its value in a `mpz_t`; this can be used for an arbitrarily large modulus.

All extend `IntegerMod_abstract`.

For efficency reasons, it stores the modulus (in all three forms, if possible) in a common (cdef) class `NativeIntStruct` rather than in the parent.

AUTHORS:

- Robert Bradshaw: most of the work
- Didier Deshommes: bit shifting
- William Stein: editing and polishing; new arith architecture
- Robert Bradshaw: implement native is_square and square_root
- William Stein: sqrt

TESTS:

```
sage: R = Integers(101^3)
sage: a = R(824362); b = R(205942)
sage: a * b
851127
```

class Int_to_IntegerMod()

EXAMPLES:

We make sure it works for every type.

```
sage: from sage.rings.integer_mod import Int_to_IntegerMod
sage: Rs = [Integers(2**k) for k in range(1,50,10)]
sage: [type(R(0)) for R in Rs]
[<type 'sage.rings.integer_mod.IntegerMod_int'>, <type 'sage.rings.integer_mod.IntegerMod_int'>,
sage: fs = [Int_to_IntegerMod(R) for R in Rs]
sage: [f(-1) for f in fs]
[1, 2047, 2097151, 2147483647, 2199023255551]
```

IntegerMod()

Create an integer modulo n with the given parent.

This is mainly for internal use.

class IntegerMod_abstract()

additive_order()

Returns the additive order of self.

This is the same as `self.order()`.

EXAMPLES:

```
sage: Integers(20)(2).additive_order()
10
sage: Integers(20)(7).additive_order()
20
sage: Integers(90308402384902)(2).additive_order()
45154201192451
```

charpoly()

Returns the characteristic polynomial of this element.

EXAMPLES:

```
sage: k = GF(3)
sage: a = k.gen()
sage: a.charpoly('x')
x + 2
sage: a + 2
0
```

AUTHORS:

- Craig Citro

crt()

Use the Chinese Remainder Theorem to find an element of the integers modulo the product of the moduli that reduces to `self` and to `other`. The modulus of `other` must be coprime to the modulus of `self`.

EXAMPLES:

```
sage: a = mod(3,5)
sage: b = mod(2,7)
sage: a.crt(b)
23

sage: a = mod(37,10^8)
sage: b = mod(9,3^8)
sage: a.crt(b)
125900000037

sage: b = mod(0,1)
sage: a.crt(b) == a
True
sage: a.crt(b).modulus()
100000000
```

AUTHORS:

- Robert Bradshaw

is_nilpotent()

Return `True` if `self` is nilpotent, i.e., some power of `self` is zero.

EXAMPLES:

```
sage: a = Integers(90384098234^3)
sage: factor(a.order())
2^3 * 191^3 * 236607587^3
sage: b = a(2*191)
sage: b.is_nilpotent()
False
sage: b = a(2*191*236607587)
sage: b.is_nilpotent()
True
```

ALGORITHM: Let $m \geq \log_2(n)$, where n is the modulus. Then $x \in \mathbf{Z}/n\mathbf{Z}$ is nilpotent if and only if $x^m = 0$.

PROOF: This is clear if you reduce to the prime power case, which you can do via the Chinese Remainder Theorem.

We could alternatively factor n and check to see if the prime divisors of n all divide x. This is asymptotically slower :-).

is_one()

is_square()

EXAMPLES:

```
sage: Mod(3,17).is_square()
False
sage: Mod(9,17).is_square()
True
sage: Mod(9,17*19^2).is_square()
True
sage: Mod(-1,17^30).is_square()
True
sage: Mod(1/9, next_prime(2^40)).is_square()
True
sage: Mod(1/25, next_prime(2^90)).is_square()
True
```

TESTS:

```
sage: Mod(1/25, 2^8).is_square()
True
sage: Mod(1/25, 2^40).is_square()
True
```

ALGORITHM: Calculate the Jacobi symbol $(\texttt{self}/p)$ at each prime p dividing n. It must be 1 or 0 for each prime, and if it is 0 mod p, where $p^k||n$, then $ord_p(\texttt{self})$ must be even or greater than k.

The case $p = 2$ is handled separately.

AUTHORS:

- Robert Bradshaw

is_unit()

log()

Return an integer x such that $b^x = a$, where a is `self`.

INPUT:

- `self` - unit modulo n
- `b` - a **generator** of the multiplicative group modulo n. If `b` is not given, `R.multiplicative_generator()` is used, where `R` is the parent of `self`.

OUTPUT: Integer x such that $b^x = a$.

Note: The base must not be too big or the current implementation, which is in PARI, will fail.

EXAMPLES:

```
sage: r = Integers(125)
sage: b = r.multiplicative_generator()^3
sage: a = b^17
sage: a.log(b)
17
sage: a.log()
63
```

A bigger example.

```
sage: FF = FiniteField(2^32+61)
sage: c = FF(4294967356)
sage: x = FF(2)
sage: a = c.log(x)
sage: a
2147483678
sage: x^a
4294967356
```

Things that can go wrong. E.g., if the base is not a generator for the multiplicative group, or not even a unit. You can also use the generic function `discrete_log`.

```
sage: a = Mod(9, 100); b = Mod(3,100)
sage: a.log(b)
...
ValueError: base (=3) for discrete log must generate multiplicative group
sage: sage.groups.generic.discrete_log(b^2,b)
2
sage: a = Mod(16, 100); b = Mod(4,100)
sage: a.log(b)
...
ValueError:  (8)
PARI failed to compute discrete log (perhaps base is not a generator or is too large)
sage: sage.groups.generic.discrete_log(a,b)
...
ZeroDivisionError: Inverse does not exist.
```

AUTHORS:

- David Joyner and William Stein (2005-11)
- William Stein (2007-01-27): update to use PARI as requested by David Kohel.

minimal_polynomial()

Returns the minimal polynomial of this element.

EXAMPLES: sage: GF(241, 'a')(1).minimal_polynomial(var = 'z') z + 240

minpoly()

Returns the minimal polynomial of this element.

EXAMPLES: sage: GF(241, 'a')(1).minpoly() x + 240

modulus()

EXAMPLES:

```
sage: Mod(3,17).modulus()
17
```

multiplicative_order()

Returns the multiplicative order of self.

EXAMPLES:

```
sage: Mod(-1,5).multiplicative_order()
2
sage: Mod(1,5).multiplicative_order()
1
sage: Mod(0,5).multiplicative_order()
...
ArithmeticError: multiplicative order of 0 not defined since it is not a unit modulo 5
```

norm()

Returns the norm of this element, which is itself. (This is here for compatibility with higher order finite fields.)

EXAMPLES:

```
sage: k = GF(691)
sage: a = k(389)
sage: a.norm()
389
```

AUTHORS:

- Craig Citro

nth_root()

Returns an nth root of `self`.

INPUT:

- `n` - integer ≥ 1 (must fit in C `int` type)
- `all` - bool (default: `False`); if `True`, return all nth roots of `self`, instead of just one.

OUTPUT: If self has an nth root, returns one (if `all` is false) or a list of all of them (if `all` is true). Otherwise, raises a `ValueError`.

AUTHORS:

- David Roe (2007-10-3)

EXAMPLES:

```
sage: k.<a> = GF(29)
sage: b = a^2 + 5*a + 1
sage: b.nth_root(5)
24
sage: b.nth_root(7)
...
ValueError: no nth root
sage: b.nth_root(4, all=True)
[21, 20, 9, 8]
```

pari()

polynomial()

Returns a constant polynomial representing this value.

EXAMPLES:

```
sage: k = GF(7)
sage: a = k.gen(); a
1
sage: a.polynomial()
1
sage: type(a.polynomial())
<type 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
```

rational_reconstruction()

EXAMPLES:

```
sage: R = IntegerModRing(97)
sage: a = R(2) / R(3)
sage: a
33
sage: a.rational_reconstruction()
2/3
```

sqrt()

Returns square root or square roots of `self` modulo n.

INPUT:

- `extend` - bool (default: `True`); if `True`, return a square root in an extension ring, if necessary. Otherwise, raise a `ValueError` if the square root is not in the base ring.
- `all` - bool (default: `False`); if `True`, return {all} square roots of self, instead of just one.

ALGORITHM: Calculates the square roots mod p for each of the primes p dividing the order of the ring, then lifts them p-adically and uses the CRT to find a square root mod n.

See also `square_root_mod_prime_power` and `square_root_mod_prime` (in this module) for more algorithmic details.

EXAMPLES:

```
sage: mod(-1, 17).sqrt()
4
sage: mod(5, 389).sqrt()
86
sage: mod(7, 18).sqrt()
5
sage: a = mod(14, 5^60).sqrt()
sage: a*a
14
sage: mod(15, 389).sqrt(extend=False)
...
ValueError: self must be a square
sage: Mod(1/9, next_prime(2^40)).sqrt()^(-2)
9
sage: Mod(1/25, next_prime(2^90)).sqrt()^(-2)
25

sage: a = Mod(3,5); a
3
sage: x = Mod(-1, 360)
sage: x.sqrt(extend=False)
...
ValueError: self must be a square
sage: y = x.sqrt(); y
sqrt359
sage: y.parent()
Univariate Quotient Polynomial Ring in sqrt359 over Ring of integers modulo 360 with modulus
sage: y^2
359
```

We compute all square roots in several cases:

```
sage: R = Integers(5*2^3*3^2); R
Ring of integers modulo 360
sage: R(40).sqrt(all=True)
[20, 160, 200, 340]
sage: [x for x in R if x^2 == 40]  # Brute force verification
[20, 160, 200, 340]
sage: R(1).sqrt(all=True)
[1, 19, 71, 89, 91, 109, 161, 179, 181, 199, 251, 269, 271, 289, 341, 359]
sage: R(0).sqrt(all=True)
[0, 60, 120, 180, 240, 300]

sage: R = Integers(5*13^3*37); R
Ring of integers modulo 406445
sage: v = R(-1).sqrt(all=True); v
[78853, 111808, 160142, 193097, 213348, 246303, 294637, 327592]
sage: [x^2 for x in v]
[406444, 406444, 406444, 406444, 406444, 406444, 406444, 406444]
sage: v = R(169).sqrt(all=True); min(v), -max(v), len(v)
(13, 13, 104)
sage: all([x^2==169 for x in v])
True
```

Modulo a power of 2:

```
sage: R = Integers(2^7); R
Ring of integers modulo 128
sage: a = R(17)
```

```
sage: a.sqrt()
23
sage: a.sqrt(all=True)
[23, 41, 87, 105]
sage: [x for x in R if x^2==17]
[23, 41, 87, 105]
```

square_root()

Returns square root or square roots of `self` modulo n.

INPUT:

- `extend` - bool (default: `True`); if `True`, return a square root in an extension ring, if necessary. Otherwise, raise a `ValueError` if the square root is not in the base ring.
- `all` - bool (default: `False`); if `True`, return {all} square roots of self, instead of just one.

ALGORITHM: Calculates the square roots mod p for each of the primes p dividing the order of the ring, then lifts them p-adically and uses the CRT to find a square root mod n.

See also `square_root_mod_prime_power` and `square_root_mod_prime` (in this module) for more algorithmic details.

EXAMPLES:

```
sage: mod(-1, 17).sqrt()
4
sage: mod(5, 389).sqrt()
86
sage: mod(7, 18).sqrt()
5
sage: a = mod(14, 5^60).sqrt()
sage: a*a
14
sage: mod(15, 389).sqrt(extend=False)
...
ValueError: self must be a square
sage: Mod(1/9, next_prime(2^40)).sqrt()^(-2)
9
sage: Mod(1/25, next_prime(2^90)).sqrt()^(-2)
25

sage: a = Mod(3,5); a
3
sage: x = Mod(-1, 360)
sage: x.sqrt(extend=False)
...
ValueError: self must be a square
sage: y = x.sqrt(); y
sqrt359
sage: y.parent()
Univariate Quotient Polynomial Ring in sqrt359 over Ring of integers modulo 360 with modulus
sage: y^2
359
```

We compute all square roots in several cases:

```
sage: R = Integers(5*2^3*3^2); R
Ring of integers modulo 360
sage: R(40).sqrt(all=True)
[20, 160, 200, 340]
sage: [x for x in R if x^2 == 40]  # Brute force verification
[20, 160, 200, 340]
```

```
sage: R(1).sqrt(all=True)
[1, 19, 71, 89, 91, 109, 161, 179, 181, 199, 251, 269, 271, 289, 341, 359]
sage: R(0).sqrt(all=True)
[0, 60, 120, 180, 240, 300]

sage: R = Integers(5*13^3*37); R
Ring of integers modulo 406445
sage: v = R(-1).sqrt(all=True); v
[78853, 111808, 160142, 193097, 213348, 246303, 294637, 327592]
sage: [x^2 for x in v]
[406444, 406444, 406444, 406444, 406444, 406444, 406444, 406444]
sage: v = R(169).sqrt(all=True); min(v), -max(v), len(v)
(13, 13, 104)
sage: all([x^2==169 for x in v])
True
```

Modulo a power of 2:

```
sage: R = Integers(2^7); R
Ring of integers modulo 128
sage: a = R(17)
sage: a.sqrt()
23
sage: a.sqrt(all=True)
[23, 41, 87, 105]
sage: [x for x in R if x^2==17]
[23, 41, 87, 105]
```

trace()

Returns the trace of this element, which is itself. (This is here for compatibility with higher order finite fields.)

EXAMPLES:

```
sage: k = GF(691)
sage: a = k(389)
sage: a.trace()
389
```

AUTHORS:

- Craig Citro

class IntegerMod_gmp()

Elements of $\mathbf{Z}/n\mathbf{Z}$ for n not small enough to be operated on in word size.

AUTHORS:

- Robert Bradshaw (2006-08-24)

is_one()

Returns `True` if this is 1, otherwise `False`.

EXAMPLES:

```
sage: mod(1,5^23).is_one()
True
sage: mod(0,5^23).is_one()
False
```

is_unit()

Return True iff this element is a unit.

EXAMPLES:

```
sage: mod(13, 5^23).is_unit()
True
sage: mod(25, 5^23).is_unit()
False
```

lift()

Lift an integer modulo n to the integers.

EXAMPLES:

```
sage: a = Mod(8943, 2^70); type(a)
<type 'sage.rings.integer_mod.IntegerMod_gmp'>
sage: lift(a)
8943
sage: a.lift()
8943
```

class IntegerMod_hom()

class IntegerMod_int()

Elements of $\mathbf{Z}/n\mathbf{Z}$ for n small enough to be operated on in 32 bits

AUTHORS:

- Robert Bradshaw (2006-08-24)

is_one()

Returns `True` if this is 1, otherwise `False`.

EXAMPLES:

```
sage: mod(6,5).is_one()
True
sage: mod(0,5).is_one()
False
```

is_unit()

Return True iff this element is a unit

EXAMPLES:

```
sage: a=Mod(23,100)
sage: a.is_unit()
True
sage: a=Mod(24,100)
sage: a.is_unit()
False
```

lift()

Lift an integer modulo n to the integers.

EXAMPLES:

```
sage: a = Mod(8943, 2^10); type(a)
<type 'sage.rings.integer_mod.IntegerMod_int'>
sage: lift(a)
751
sage: a.lift()
751
```

sqrt()

Returns square root or square roots of `self` modulo n.

INPUT:

- `extend` - bool (default: `True`); if `True`, return a square root in an extension ring, if necessary. Otherwise, raise a `ValueError` if the square root is not in the base ring.
- `all` - bool (default: `False`); if `True`, return {all} square roots of self, instead of just one.

ALGORITHM: Calculates the square roots mod p for each of the primes p dividing the order of the ring, then lifts them p-adically and uses the CRT to find a square root mod n.

See also `square_root_mod_prime_power` and `square_root_mod_prime` (in this module) for more algorithmic details.

EXAMPLES:

```
sage: mod(-1, 17).sqrt()
4
sage: mod(5, 389).sqrt()
86
sage: mod(7, 18).sqrt()
5
sage: a = mod(14, 5^60).sqrt()
sage: a*a
14
sage: mod(15, 389).sqrt(extend=False)
...
ValueError: self must be a square
sage: Mod(1/9, next_prime(2^40)).sqrt()^(-2)
9
sage: Mod(1/25, next_prime(2^90)).sqrt()^(-2)
25

sage: a = Mod(3,5); a
3
sage: x = Mod(-1, 360)
sage: x.sqrt(extend=False)
...
ValueError: self must be a square
sage: y = x.sqrt(); y
sqrt359
sage: y.parent()
Univariate Quotient Polynomial Ring in sqrt359 over Ring of integers modulo 360 with modulus
sage: y^2
359
```

We compute all square roots in several cases:

```
sage: R = Integers(5*2^3*3^2); R
Ring of integers modulo 360
sage: R(40).sqrt(all=True)
[20, 160, 200, 340]
sage: [x for x in R if x^2 == 40]  # Brute force verification
[20, 160, 200, 340]
sage: R(1).sqrt(all=True)
[1, 19, 71, 89, 91, 109, 161, 179, 181, 199, 251, 269, 271, 289, 341, 359]
sage: R(0).sqrt(all=True)
[0, 60, 120, 180, 240, 300]

sage: R = Integers(5*13^3*37); R
Ring of integers modulo 406445
sage: v = R(-1).sqrt(all=True); v
[78853, 111808, 160142, 193097, 213348, 246303, 294637, 327592]
sage: [x^2 for x in v]
[406444, 406444, 406444, 406444, 406444, 406444, 406444, 406444]
```

```
sage: v = R(169).sqrt(all=True); min(v), -max(v), len(v)
(13, 13, 104)
sage: all([x^2==169 for x in v])
True
```

Modulo a power of 2:

```
sage: R = Integers(2^7); R
Ring of integers modulo 128
sage: a = R(17)
sage: a.sqrt()
23
sage: a.sqrt(all=True)
[23, 41, 87, 105]
sage: [x for x in R if x^2==17]
[23, 41, 87, 105]
```

class IntegerMod_int64()

Elements of $\mathbf{Z}/n\mathbf{Z}$ for n small enough to be operated on in 64 bits

AUTHORS:

- Robert Bradshaw (2006-09-14)

is_one()

Returns `True` if this is 1, otherwise `False`.

EXAMPLES:

```
sage: (mod(-1,5^10)^2).is_one()
True
sage: mod(0,5^10).is_one()
False
```

is_unit()

Return True iff this element is a unit.

EXAMPLES:

```
sage: mod(13, 5^10).is_unit()
True
sage: mod(25, 5^10).is_unit()
False
```

lift()

Lift an integer modulo n to the integers.

EXAMPLES:

```
sage: a = Mod(8943, 2^25); type(a)
<type 'sage.rings.integer_mod.IntegerMod_int64'>
sage: lift(a)
8943
sage: a.lift()
8943
```

class IntegerMod_to_IntegerMod()

Very fast IntegerMod to IntegerMod homomorphism.

EXAMPLES:

```
sage: from sage.rings.integer_mod import IntegerMod_to_IntegerMod
sage: Rs = [Integers(3**k) for k in range(1,30,5)]
sage: [type(R(0)) for R in Rs]
```

```
[<type 'sage.rings.integer_mod.IntegerMod_int'>, <type 'sage.rings.integer_mod.IntegerMod_int'>,
sage: fs = [IntegerMod_to_IntegerMod(S, R) for R in Rs for S in Rs if S is not R and S.order() >
sage: all([f(-1) == f.codomain()(-1) for f in fs])
True
sage: [f(-1) for f in fs]
[2, 2, 2, 2, 2, 728, 728, 728, 728, 177146, 177146, 177146, 43046720, 43046720, 10460353202]
```

class Integer_to_IntegerMod()

Fast $\mathbf{Z} \to \mathbf{Z}/n\mathbf{Z}$ morphism.

EXAMPLES:

We make sure it works for every type.

```
sage: from sage.rings.integer_mod import Integer_to_IntegerMod
sage: Rs = [Integers(10), Integers(10^5), Integers(10^10)]
sage: [type(R(0)) for R in Rs]
[<type 'sage.rings.integer_mod.IntegerMod_int'>, <type 'sage.rings.integer_mod.IntegerMod_int64'
sage: fs = [Integer_to_IntegerMod(R) for R in Rs]
sage: [f(-1) for f in fs]
[9, 99999, 9999999999]
```

Mod()

Return the equivalence class of n modulo m as an element of $\mathbf{Z}/m\mathbf{Z}$.

EXAMPLES:

```
sage: x = Mod(12345678, 32098203845329048)
sage: x
12345678
sage: x^100
1017322209155072
```

You can also use the lowercase version:

```
sage: mod(12,5)
2
```

class NativeIntStruct()

We store the various forms of the modulus here rather than in the parent for efficiency reasons.

We may also store a cached table of all elements of a given ring in this class.

precompute_table()

Function to compute and cache all elements of this class.

If inverses==True, also computes and caches the inverses of the invertible elments

fast_lucas()

Return $V_k(P, 1)$ where V_k is the Lucas function defined by the recursive relation

$V_k(P, Q) = PV_{k-1}(P, Q) - QV_{k-2}(P, Q)$

with $V_0 = 2, V_1(P_Q) = P$.

REFERENCES:

•H. Postl. 'Fast evaluation of Dickson Polynomials' Contrib. to General Algebra, Vol. 6 (1988) pp. 223-225

AUTHORS:

•Robert Bradshaw

TESTS:

```
sage: from sage.rings.integer_mod import fast_lucas, slow_lucas
sage: all([fast_lucas(k, a) == slow_lucas(k, a)
...        for a in Integers(23)
...        for k in range(13)])
True
```

is_IntegerMod()

Return `True` if and only if x is an integer modulo n.

EXAMPLES:

```
sage: from sage.rings.integer_mod import is_IntegerMod
sage: is_IntegerMod(5)
False
sage: is_IntegerMod(Mod(5,10))
True
```

makeNativeIntStruct()

Function to convert a Sage Integer into class NativeIntStruct.

Note: This function seems completely redundant, and is not used anywhere.

mod()

Return the equivalence class of n modulo m as an element of $\mathbf{Z}/m\mathbf{Z}$.

EXAMPLES:

```
sage: x = Mod(12345678, 32098203845329048)
sage: x
12345678
sage: x^100
1017322209155072
```

You can also use the lowercase version:

```
sage: mod(12,5)
2
```

slow_lucas()

Lucas function defined using the standard definition, for consistency testing.

square_root_mod_prime()

Calculates the square root of a, where a is an integer mod p; if a is not a perfect square, this returns an (incorrect) answer without checking.

ALGORITHM: Several cases based on residue class of $p \bmod 16$.

- $p \bmod 2 = 0$: $p = 2$ so $\sqrt{a} = a$.
- $p \bmod 4 = 3$: $\sqrt{a} = a^{(p+1)/4}$.
- $p \bmod 8 = 5$: $\sqrt{a} = \zeta i a$ where $\zeta = (2a)^{(p-5)/8}$, $i = \sqrt{-1}$.
- $p \bmod 16 = 9$: Similar, work in a bi-quadratic extension of $\mathbf{F}_p$ for small p, Tonelli and Shanks for large p.
- $p \bmod 16 = 1$: Tonelli and Shanks.

REFERENCES:

- Siguna Muller. 'On the Computation of Square Roots in Finite Fields' Designs, Codes and Cryptography, Volume 31, Issue 3 (March 2004)

•A. Oliver L. Atkin. 'Probabilistic primality testing' (Chapter 30, Section 4) In Ph. Flajolet and P. Zimmermann, editors, Algorithms Seminar, 1991-1992. INRIA Research Report 1779, 1992, http://www.inria.fr/rrrt/rr-1779.html. Summary by F. Morain. http://citeseer.ist.psu.edu/atkin92probabilistic.html

•H. Postl. 'Fast evaluation of Dickson Polynomials' Contrib. to General Algebra, Vol. 6 (1988) pp. 223-225

AUTHORS:

•Robert Bradshaw

TESTS: Every case appears in the first hundred primes.

```
sage: from sage.rings.integer_mod import square_root_mod_prime   # sqrt() uses brute force for s
sage: all([square_root_mod_prime(a*a)^2 == a*a
...        for p in prime_range(100)
...        for a in Integers(p)])
True
```

square_root_mod_prime_power()

Calculates the square root of a, where a is an integer mod p^e.

ALGORITHM: Perform p-adically by stripping off even powers of p to get a unit and lifting $\sqrt{unit} \bmod p$ via Newton's method.

AUTHORS:

•Robert Bradshaw

EXAMPLES:

```
sage: from sage.rings.integer_mod import square_root_mod_prime_power
sage: a=Mod(17,2^20)
sage: b=square_root_mod_prime_power(a,2,20)
sage: b^2 == a
True


sage: a=Mod(72,97^10)
sage: b=square_root_mod_prime_power(a,97,10)
sage: b^2 == a
True
```

24.5 Field Q of Rational Numbers.

The class `RationalField` represents the field **Q** of (arbitrary precision) rational numbers. Each rational number is an instance of the class `Rational`.

Interactively, an instance of `RationalField` is available as `QQ`.

```
sage: QQ
Rational Field
```

Values of various types can be converted to rational numbers by using the `__call__` method of `RationalField` (that is, by treating `QQ` as a function).

```
sage: RealField(9).pi()
3.1
sage: QQ(RealField(9).pi())
22/7
sage: QQ(RealField().pi())
245850922/78256779
sage: QQ(35)
35
sage: QQ('12/347')
12/347
sage: QQ(exp(pi*I))
-1
sage: x = polygen(ZZ)
sage: QQ((3*x)/(4*x))
3/4
```

TEST:

```
sage: Q = RationalField()
sage: Q == loads(dumps(Q))
True
sage: RationalField() is RationalField()
True
```

class RationalField()

The class RationalField represents the field **Q** of rational numbers.

EXAMPLES:

```
sage: a = long(90182430982109382109381209381092830918309183209 1)
sage: b = QQ(a); b
901824309821093821093812093810928309183091832091
sage: QQ(b)
901824309821093821093812093810928309183091832091
sage: QQ(int(93820984323))
93820984323
sage: QQ(ZZ(90182430982109382109381209381092830918309183209 1))
901824309821093821093812093810928309183091832091
sage: QQ('-930482/9320842317')
-930482/9320842317
sage: QQ((-930482, 9320842317))
-930482/9320842317
sage: QQ([9320842317])
9320842317
sage: QQ(pari(3902938402384092830948284209843028439824398239 4))
390293840238409283094828420984302843982439823 94
sage: QQ('sage')
...
TypeError: unable to convert sage to a rational
```

Coercion from the reals to the rational is done by default using continued fractions.

```
sage: QQ(RR(3929329/32))
3929329/32
sage: QQ(-RR(3929329/32))
-3929329/32
sage: QQ(RR(1/7)) - 1/7
0
```

If you specify an optional second base argument, then the string representation of the float is used.

```
sage: QQ(23.2, 2)
6530219459687219/281474976710656
sage: 6530219459687219.0/281474976710656
23.199999999999999
sage: a = 23.2; a
23.200000000000000
sage: QQ(a, 10)
116/5
```

Here's a nice example involving elliptic curves:

```
sage: E = EllipticCurve('11a')
sage: L = E.lseries().at1(300)[0]; L
0.253841860855911
sage: O = E.period_lattice().omega(); O
1.26920930427955
sage: t = L/O; t
0.200000000000...
sage: QQ(RealField(45)(t))
1/5
```

absolute_degree()

EXAMPLES:

```
sage: QQ.absolute_degree()
1
```

characteristic()

Return 0, since the rational field has characteristic 0.

EXAMPLES:

```
sage: c = QQ.characteristic(); c
0
sage: parent(c)
Integer Ring
```

class_number()

Return the class number of the field of rational numbers, which is 1.

EXAMPLES:

```
sage: QQ.class_number()
1
```

completion(*p, prec, extras={}*)

complex_embedding(*prec=53*)

Return embedding of the rational numbers into the complex numbers.

EXAMPLES:

```
sage: QQ.complex_embedding()
Ring morphism:
  From: Rational Field
  To:   Complex Field with 53 bits of precision
  Defn: 1 |--> 1.00000000000000
sage: QQ.complex_embedding(20)
Ring morphism:
  From: Rational Field
```

```
  To:   Complex Field with 20 bits of precision
  Defn: 1 |--> 1.0000
```

construction()

degree()

EXAMPLES:

```
sage: QQ.degree()
1
```

discriminant()

Return the discriminant of the field of rational numbers, which is 1.

EXAMPLES:

```
sage: QQ.discriminant()
1
```

embeddings(*K*)

Return list of the one embedding of $\mathbf{Q}$ into K, if it exists.

EXAMPLES:

```
sage: QQ.embeddings(QQ)
[Ring Coercion endomorphism of Rational Field]
sage: QQ.embeddings(CyclotomicField(5))
[Ring Coercion morphism:
  From: Rational Field
  To:   Cyclotomic Field of order 5 and degree 4]
```

K must have characteristic 0:

```
sage: QQ.embeddings(GF(3))
...
ValueError: no embeddings of the rational field into K.
```

extension(*poly, names, check=True, embedding=None*)

EXAMPLES:

We make a single absolute extension:

```
sage: K.<a> = QQ.extension(x^3 + 5); K
Number Field in a with defining polynomial x^3 + 5
```

We make an extension generated by roots of two polynomials:

```
sage: K.<a,b> = QQ.extension([x^3 + 5, x^2 + 3]); K
Number Field in a with defining polynomial x^3 + 5 over its base field
sage: b^2
-3
sage: a^3
-5
```

gen(*n=0*)

EXAMPLES:

```
sage: QQ.gen()
1
```

gens()

EXAMPLES:

```
sage: QQ.gens()
(1,)
```

is_absolute()
$\mathbf{Q}$ is an absolute extension of $\mathbf{Q}$.
EXAMPLES:

```
sage: QQ.is_absolute()
True
```

is_atomic_repr()

is_field()
Return `True`, since the rational field is a field.
EXAMPLES:

```
sage: QQ.is_field()
True
```

is_finite()
Return `False`, since the rational field is not finite.
EXAMPLES:

```
sage: QQ.is_finite()
False
```

is_prime_field()
Return `True`, since $\mathbf{Q}$ is a prime field.
EXAMPLES:

```
sage: QQ.is_prime_field()
True
```

is_subring(K)
Return `True` if $\mathbf{Q}$ is a subring of K.
We are only able to determine this in some cases, e.g., when K is a field or of positive characteristic.
EXAMPLES:

```
sage: QQ.is_subring(QQ)
True
sage: QQ.is_subring(QQ['x'])
True
sage: QQ.is_subring(GF(7))
False
sage: QQ.is_subring(CyclotomicField(7))
True
sage: QQ.is_subring(ZZ)
False
sage: QQ.is_subring(Frac(ZZ))
True
```

maximal_order()
Return the maximal order of the rational numbers, i.e., the ring $\mathbf{Z}$ of integers.
EXAMPLES:

```
sage: QQ.maximal_order()
Integer Ring
sage: QQ.ring_of_integers ()
Integer Ring
```

ngens()
EXAMPLES:

```
sage: QQ.ngens()
1
```

number_field()

Return the number field associated to $\mathbf{Q}$. Since $\mathbf{Q}$ is a number field, this just returns $\mathbf{Q}$ again.

EXAMPLES:

```
sage: QQ.number_field() is QQ
True
```

order()

EXAMPLES:

```
sage: QQ.order()
+Infinity
```

power_basis()

Return a power basis for this number field over its base field.

The power basis is always [1] for the rational field. This method is defined to make the rational field behave more like a number field.

EXAMPLES:

```
sage: QQ.power_basis()
[1]
```

random_element(*num_bound=None, den_bound=None, distribution=None*)

EXAMPLES:

```
sage: QQ.random_element(10,10) # random output
-5/3
```

range_by_height(*start, end=None*)

Range function for rational numbers, ordered by height.

Returns a Python generator for the list of rational numbers with heights in `range(start, end)`. Follows the same convention as Python range, see range? for details.

See also QQ.__iter__?

EXAMPLES:

All rational numbers with height strictly less than 4:

```
sage: list(QQ.range_by_height(4))
[0, 1, -1, 1/2, -1/2, 2, -2, 1/3, -1/3, 3, -3, 2/3, -2/3, 3/2, -3/2]
sage: [a.height() for a in QQ.range_by_height(4)]
[1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3, 3, 3]
```

All rational numbers with height 2:

```
sage: list(QQ.range_by_height(2, 3))
[1/2, -1/2, 2, -2]
```

Nonsensical integer arguments will return an empty generator:

```
sage: list(QQ.range_by_height(3, 3))
[]
sage: list(QQ.range_by_height(10, 1))
[]
```

There are no rational numbers with height ≤ 0:

```
sage: list(QQ.range_by_height(-10, 1))
[]
```

signature()
Return the signature of the rational field, which is (1,0), since there are 1 real and no complex embeddings.
EXAMPLES:

```
sage: QQ.signature()
(1, 0)
```

zeta(*n=2*)
Return a root of unity in `self`.
INPUT:

- `n` - integer (default: 2) order of the root of unity

EXAMPLES:

```
sage: QQ.zeta()
-1
sage: QQ.zeta(2)
-1
sage: QQ.zeta(1)
1
sage: QQ.zeta(3)
...
ValueError: no n-th root of unity in rational field
```

frac(*n, d*)

is_RationalField(*x*)

24.6 Rational Numbers

AUTHORS:

- William Stein (2005): first version
- William Stein (2006-02-22): floor and ceil (pure fast GMP versions).
- Gonzalo Tornaria and William Stein (2006-03-02): greatly improved python/GMP conversion; hashing
- William Stein and Naqi Jaffery (2006-03-06): height, sqrt examples, and improve behavior of sqrt.
- David Harvey (2006-09-15): added nth_root
- Pablo De Napoli (2007-04-01): corrected the implementations of multiplicative_order, is_one; optimized __nonzero__ ; documented: lcm,gcd

TESTS:

```
sage: a = -2/3
sage: a == loads(dumps(a))
True
```

class Q_to_Z()
TESTS:

```
sage: type(ZZ.convert_map_from(QQ))
<type 'sage.rings.rational.Q_to_Z'>
```

section()

EXAMPLES:

```
sage: sage.rings.rational.Q_to_Z(QQ, ZZ).section()
Natural morphism:
  From: Integer Ring
  To:   Rational Field
```

class **Rational**()

A Rational number.

Rational numbers are implemented using the GMP C library.

EXAMPLES:

```
sage: a = -2/3
sage: type(a)
<type 'sage.rings.rational.Rational'>
sage: parent(a)
Rational Field
sage: Rational('1/0')
...
TypeError: unable to convert 1/0 to a rational
sage: Rational(1.5)
3/2
sage: Rational('9/6')
3/2
sage: Rational((2^99,2^100))
1/2
sage: Rational(("2", "10"), 16)
1/8
sage: Rational(QQbar(125/8).nth_root(3))
5/2
sage: Rational(AA(209735/343 - 17910/49*golden_ratio).nth_root(3) + 3*golden_ratio)
53/7
sage: QQ(float(1.5))
3/2
sage: QQ(RDF(1.2))
6/5
```

Conversion from PARI:

```
sage: Rational(pari('-939082/3992923'))
-939082/3992923
```

additive_order()

Return the additive order of self.

OUTPUT: integer or infinity

EXAMPLES:

```
sage: QQ(0).additive_order()
1
sage: QQ(1).additive_order()
+Infinity
```

ceil()

Return the ceiling of this rational number.

OUTPUT: Integer

If this rational number is an integer, this returns this number, otherwise it returns the floor of this number +1.

EXAMPLES:

```
sage: n = 5/3; n.ceil()
2
sage: n = -17/19; n.ceil()
0
sage: n = -7/2; n.ceil()
-3
sage: n = 7/2; n.ceil()
4
sage: n = 10/2; n.ceil()
5
```

charpoly()

Return the characteristic polynomial of this rational number. This will always be just var - self; this is really here so that code written for number fields won't crash when applied to rational numbers.

INPUT:

- var - a string

OUTPUT: Polynomial

EXAMPLES:

```
sage: (1/3).charpoly('x')
 x - 1/3
```

AUTHORS:

- Craig Citro

conjugate()

Return the complex conjugate of this rational number, which is the number itself.

EXAMPLES:

```
sage: n = 23/11
sage: n.conjugate()
23/11
```

content()

Return the content of self and other, i.e. the unique positive rational number c such that self/c and other/c are coprime integers.

other can be a rational number or a list of rational numbers.

EXAMPLES:

```
sage: a = 2/3
sage: a.content(2/3)
2/3
sage: a.content(1/5)
1/15
sage: a.content([2/5, 4/9])
2/45
```

copy()

Return a copy of self.

OUTPUT: Rational

EXAMPLES:

```
sage: a = -17/37
sage: a.copy() is a
False
```

Coercion does not make a new copy:

```
sage: QQ(a) is a
True
```

The constructor also makes a new copy:

```
sage: Rational(a) is a
False
```

denom()

Returns the denominator of this rational number.

EXAMPLES:

```
sage: x = 5/13
sage: x.denom()
13
sage: x = -9/3
sage: x.denom()
1
```

denominator()

Returns the denominator of this rational number.

EXAMPLES:

```
sage: x = -5/11
sage: x.denominator()
11
sage: x = 9/3
sage: x.denominator()
1
```

factor()

Return the factorization of this rational number.

OUTPUT: Factorization

EXAMPLES:

```
sage: (-4/17).factor()
-1 * 2^2 * 17^-1
```

Trying to factor 0 gives an arithmetic error:

```
sage: (0/1).factor()
...
ArithmeticError: Prime factorization of 0 not defined.
```

floor()

Return the floor of this rational number as an integer.

OUTPUT: Integer

EXAMPLES:

```
sage: n = 5/3; n.floor()
1
sage: n = -17/19; n.floor()
-1
sage: n = -7/2; n.floor()
-4
sage: n = 7/2; n.floor()
3
sage: n = 10/2; n.floor()
5
```

gamma()

Return the gamma function evaluated at self. This value is exact for integers and half-integers, otherwise a numerical approximation is returned.

EXAMPLES:

```
sage: gamma(1/2)
sqrt(pi)
sage: gamma(7/2)
15/8*sqrt(pi)
sage: gamma(-3/2)
4/3*sqrt(pi)
sage: gamma(6/1)
120
sage: gamma(1/3)
2.67893853470775
```

This function accepts an optional precision argument:

```
sage: (1/3).gamma(prec=100)
2.6789385347077476336556929410
sage: (1/2).gamma(prec=100)
1.7724538509055160272981674833
```

gcd()

Return a gcd of the rational numbers self and other.

If self = other = 0, this is by convention 0. In all other cases it can (mathematically) be any nonzero rational number, but for simplicity we choose to always return 1.

EXAMPLES:

```
sage: gcd(1/3, 2/1)
1
sage: gcd(1/1, 0/1)
1
sage: gcd(0/1, 0/1)
0
```

height()

The max absolute value of the numerator and denominator of self, as an Integer.

OUTPUT: Integer

EXAMPLES:

```
sage: a = 2/3
sage: a.height()
3
sage: a = 34/3
sage: a.height()
34
sage: a = -97/4
sage: a.height()
97
```

AUTHORS:

- Naqi Jaffery (2006-03-05): examples

is_S_integral()

Determine if the rational number is S-integral.

x is S-integral if x.valuation(p)>=0 for all p not in S, i.e., the denominator of x is divisible only by the primes in S.

INPUT:

- S - list or tuple of primes.

OUTPUT: bool

Note: Primality of the entries in S is not checked.

EXAMPLES:

```
sage: QQ(1/2).is_S_integral()
False
sage: QQ(1/2).is_S_integral([2])
True
sage: [a for a in range(1,11) if QQ(101/a).is_S_integral([2,5])]
[1, 2, 4, 5, 8, 10]
```

is_S_unit()

Determine if the rational number is an S-unit.

x is an S-unit if x.valuation(p)==0 for all p not in S, i.e., the numerator and denominator of x are divisible only by the primes in S.

INPUT:

- S - list or tuple of primes.

OUTPUT: bool

Note: Primality of the entries in S is not checked.

EXAMPLES:

```
sage: QQ(1/2).is_S_unit()
False
sage: QQ(1/2).is_S_unit([2])
True
sage: [a for a in range(1,11) if QQ(10/a).is_S_unit([2,5])]
[1, 2, 4, 5, 8, 10]
```

is_integral()

Determine if a rational number is integral (i.e is in **Z**).

OUTPUT: bool

EXAMPLES:

```
sage: QQ(1/2).is_integral()
False
sage: QQ(4/4).is_integral()
True
```

is_nth_power()

Returns True if self is an nth power, else False.

INPUT:

- n - integer (must fit in C int type)

Note: Use this function when you nede test test if a rational number is an n'th power, but do not need to know the value of its n'th root. If the value is needed, use nth_root().

AUTHORS:

- John Cremona (2009-04-04)

EXAMPLES:

```
sage: QQ(25/4).is_nth_power(2)
True
sage: QQ(125/8).is_nth_power(3)
True
sage: QQ(-125/8).is_nth_power(3)
```

```
True
sage: QQ(25/4).is_nth_power(-2)
True

sage: QQ(9/2).is_nth_power(2)
False
sage: QQ(-25).is_nth_power(2)
False
```

is_one()

Determine if a rational number is one.

OUTPUT: bool

EXAMPLES:

```
sage: QQ(1/2).is_one()
False
sage: QQ(4/4).is_one()
True
```

is_padic_square()

Determines whether this rational number is a square in Q_p (or in R when p = infinity).

INPUT:

- p - a prime number, or infinity

EXAMPLES: sage: QQ(2).is_padic_square(7) True sage: QQ(98).is_padic_square(7) True sage: QQ(2).is_padic_square(5) False

TESTS: sage: QQ(5/7).is_padic_square(int(2)) False

is_perfect_power()

Returns True if self is a perfect power.

INPUT:

- **expected_value - (bool) whether or not this rational is expected** be a perfect power. This does not affect the correctness of the output, only the runtime.

If expected_value is False (default) it will check the smallest of the numerator and denominator is a perfect power as a first step, which is often faster than checking if the quotient is a perfect power.

EXAMPLES:

```
sage: (4/9).is_perfect_power()
True
sage: (144/1).is_perfect_power()
True
sage: (4/3).is_perfect_power()
False
sage: (2/27).is_perfect_power()
False
sage: (4/27).is_perfect_power()
False
sage: (-1/25).is_perfect_power()
False
sage: (-1/27).is_perfect_power()
True
sage: (0/1).is_perfect_power()
True
```

The second parameter does not change the result, but may change the runtime.

```
sage: (-1/27).is_perfect_power(True)
True
sage: (-1/25).is_perfect_power(True)
False
sage: (2/27).is_perfect_power(True)
False
sage: (144/1).is_perfect_power(True)
True
```

This test makes sure we workaround a bug in GMP (see trac #4612):

```
sage: [ -a for a in srange(100) if not QQ(-a^3).is_perfect_power() ]
[]
sage: [ -a for a in srange(100) if not QQ(-a^3).is_perfect_power(True) ]
[]
```

is_square()

Return whether or not this rational number is a square.

OUTPUT: bool

EXAMPLES:

```
sage: x = 9/4
sage: x.is_square()
True
sage: x = (7/53)^100
sage: x.is_square()
True
sage: x = 4/3
sage: x.is_square()
False
sage: x = -1/4
sage: x.is_square()
False
```

lcm()

Return the least common multiple of self and other.

One way to define this notion is the following:

Note that each rational positive rational number can be written as a product of primes with integer (positive or negative) powers in a unique way.

Then, the LCM of two rational numbers x,y can be defined by specifying that the exponent of every prime p in lcm(x,y) is the supremum of the exponents of p in x, and the exponent of p in y (The primes that does not appear in the decomposition of x or y are considered to have exponent zero).

This definition is consistent with the definition of the LCM in the rational integers. Our hopefully interesting notion of LCM for rational numbers is illustrated in the examples below.

EXAMPLES:

```
sage: lcm(2/3,1/5)
2
```

This is consistent with the definition above, since:

$$2/3 = 2^1 * 3^{-1} * 5^0$$

$$1/5 = 2^0 * 3^0 * 5^{-1}$$

and hence,

$$lcm(2/3, 1/5) = 2^1 * 3^0 * 5^0 = 2.$$

```
sage: lcm(2/3,7/5)
14
```

In this example:

$$2/3 = 2^1 * 3^{-1} * 5^0 * 7^0$$

$$7/5 = 2^0 * 3^0 * 5^{-1} * 7^1$$

$$lcm(2/3, 7/5) = 2^1 * 3^0 * 5^0 * 7^1 = 14$$

```
sage: lcm(1/3,1/5)
1
```

In this example:

$$1/3 = 3^{-1} * 5^0$$

$$1/5 = 3^0 * 5^{-1}$$

$$lcm(1/3, 1/5) = 3^0 * 5^0 = 1$$

```
sage: lcm(1/3,1/6)
1/3
```

In this example:

$$1/3 = 2^0 * 3^{-1}$$

$$1/6 = 2^{-1} * 3^{-1}$$

$$lcm(1/3, 1/6) = 2^0 * 3^{-1} = 1/3$$

list()

Return a list with the rational element in it, to be compatible with the method for number fields.

OUTPUT:

- `list` - the list [self]

EXAMPLES:

```
sage: m = 5/3
sage: m.list()
[5/3]
```

minpoly()

Return the minimal polynomial of this rational number. This will always be just x - self; this is really here so that code written for number fields won't crash when applied to rational numbers.

INPUT:

- `var` - a string

OUTPUT: Polynomial

EXAMPLES:

```
sage: (1/3).minpoly('x')
 x - 1/3
```

AUTHORS:

- Craig Citro

mod_ui()

Return the remainder upon division of self by the unsigned long integer n.

INPUT:

- n - an unsigned long integer

OUTPUT: integer

EXAMPLES:

```
sage: (-4/17).mod_ui(3)
1
sage: (-4/17).mod_ui(17)
...
ArithmeticError: The inverse of 0 modulo 17 is not defined.
```

multiplicative_order()

Return the multiplicative order of self.

OUTPUT: Integer or infinity

EXAMPLES:

```
sage: QQ(1).multiplicative_order()
1
sage: QQ('1/-1').multiplicative_order()
2
sage: QQ(0).multiplicative_order()
+Infinity
sage: QQ('2/3').multiplicative_order()
+Infinity
sage: QQ('1/2').multiplicative_order()
+Infinity
```

norm()

Returns the norm from Q to Q of x (which is just x). This was added for compatibility with NumberFields.

OUTPUT:

- Rational - reference to self

EXAMPLES:

```
sage: (1/3).norm()
 1/3
```

AUTHORS:

- Craig Citro

nth_root()

Computes the nth root of self, or raises a ValueError if self is not a perfect nth power.

INPUT:

- n - integer (must fit in C int type)

AUTHORS:

- David Harvey (2006-09-15)

EXAMPLES:

```
sage: (25/4).nth_root(2)
5/2
sage: (125/8).nth_root(3)
5/2
sage: (-125/8).nth_root(3)
-5/2
sage: (25/4).nth_root(-2)
2/5

sage: (9/2).nth_root(2)
...
ValueError: not a perfect 2nd power

sage: (-25/4).nth_root(2)
...
ValueError: cannot take even root of negative number
```

numer()

Return the numerator of this rational number.

EXAMPLE:

```
sage: x = -5/11
sage: x.numer()
-5
```

numerator()

Return the numerator of this rational number.

EXAMPLE:

```
sage: x = 5/11
sage: x.numerator()
5

sage: x = 9/3
sage: x.numerator()
3
```

period()

Return the period of the repeating part of the decimal expansion of this rational number.

ALGORITHM: When a rational number n/d with $(n, d) == 1$ is expanded, the period begins after s terms and has length t, where s and t are the smallest numbers satisfying $10^s = 10^(s + t)(modd)$. When d is coprime to 10, this becomes a purely periodic decimal with $10^t = 1(modd)$. (Lehmer 1941 and Mathworld).

EXAMPLES:

```
sage: (1/7).period()
6
sage: RR(1/7)
0.142857142857143
sage: (1/8).period()
1
sage: RR(1/8)
0.125000000000000
sage: RR(1/6)
0.166666666666667
sage: (1/6).period()
1
sage: x = 333/106
```

```
sage: x.period()
13
sage: RealField(200)(x)
3.14150943396226415094339622641509433962264150943396226415094
```

prime_to_S_part()

Returns self with all powers of all primes in S removed.

INPUT:

- S - list or tuple of primes.

OUTPUT: rational

Note: Primality of the entries in S is not checked.

EXAMPLES:

```
sage: QQ(3/4).prime_to_S_part()
3/4
sage: QQ(3/4).prime_to_S_part([2])
3
sage: QQ(-3/4).prime_to_S_part([3])
-1/4
sage: QQ(700/99).prime_to_S_part([2,3,5])
7/11
sage: QQ(-700/99).prime_to_S_part([2,3,5])
-7/11
sage: QQ(0).prime_to_S_part([2,3,5])
0
sage: QQ(-700/99).prime_to_S_part([])
-700/99
```

round()

Returns the nearest integer to self.

INPUT:

- self - a rational number
- mode - a rounding mode for half integers:
  - 'toward' (default) rounds toward zero
  - 'away' rounds away from zero
  - 'up' rounds up
  - 'down' rounds down
  - 'even' rounds toward the even integer
  - 'odd' rounds toward the odd integer

OUTPUT: Integer

EXAMPLES:

```
sage: n = 4/3; n.round()
1
sage: n = -17/4; n.round()
-4
sage: n = -5/2; n.round()
-2
sage: n.round("away")
-3
sage: n.round("up")
-2
sage: n.round("down")
-3
```

```
sage: n.round("even")
-2
sage: n.round("odd")
-3
```

sqrt()

The square root function.

INPUT:

- prec - integer (default: None): if None, returns an exact square root; otherwise returns a numerical square root if necessary, to the given bits of precision.
- extend - bool (default: True); if True, return a square root in an extension ring, if necessary. Otherwise, raise a ValueError if the square is not in the base ring.
- all - bool (default: False); if True, return all square roots of self, instead of just one.

EXAMPLES:

```
sage: x = 25/9
sage: x.sqrt()
5/3
sage: sqrt(x)
5/3
sage: x = 64/4
sage: x.sqrt()
4
sage: x = 100/1
sage: x.sqrt()
10
sage: x.sqrt(all=True)
[10, -10]
sage: x = 81/5
sage: x.sqrt()
9*sqrt(1/5)
sage: x = -81/3
sage: x.sqrt()
3*sqrt(-3)

sage: n = 2/3
sage: n.sqrt()
sqrt(2/3)
sage: n.sqrt(prec=10)
0.82
sage: n.sqrt(prec=100)
0.81649658092772603273242802490
sage: n.sqrt(prec=100)^2
0.66666666666666666666666666667
sage: n.sqrt(prec=53, all=True)
[0.816496580927726, -0.816496580927726]
sage: n.sqrt(extend=False, all=True)
...
ValueError: square root of 2/3 not a rational number
sage: sqrt(-2/3, all=True)
[sqrt(-2/3), -sqrt(-2/3)]
sage: sqrt(-2/3, prec=53)
0.816496580927726*I
sage: sqrt(-2/3, prec=53, all=True)
[0.816496580927726*I, -0.816496580927726*I]
```

AUTHORS:

•Naqi Jaffery (2006-03-05): some examples

sqrt_approx()

Return numerical approximation with given number of bits of precision to this rational number. If all is given, return both approximations.

INPUT:

•`prec` - integer

•`all` - bool

EXAMPLES:

```
sage: (5/3).sqrt_approx()
doctest:1172: DeprecationWarning: This function is deprecated.  Use sqrt with a given number
1.29099444873581
sage: (990829038092384908234098239048230984/4).sqrt_approx()
4.9770197862083713747374920870362581922510725585130996993055116540856385e17
sage: (5/3).sqrt_approx(prec=200)
1.2909944487358056283930884665941332036109739017638636088625
sage: (9/4).sqrt_approx()
3/2
```

squarefree_part()

Return the square free part of x, i.e., an integer z such that $x = zy^2$, for a perfect square y^2.

EXAMPLES:

```
sage: a = 1/2
sage: a.squarefree_part()
2
sage: b = a/a.squarefree_part()
sage: b, b.is_square()
(1/4, True)
sage: a = 24/5
sage: a.squarefree_part()
30
```

str()

INPUT:

•`base` - integer (default: 10); base must be between 2 and 36.

OUTPUT: string

EXAMPLES:

```
sage: (-4/17).str()
'-4/17'
sage: (-4/17).str(2)
'-100/10001'
```

Note that the base must be at most 36.

```
sage: (-4/17).str(40)
...
ValueError: base (=40) must be between 2 and 36
sage: (-4/17).str(1)
...
ValueError: base (=1) must be between 2 and 36
```

support()

Return a sorted list of the primes where this rational number has non-zero valuation.

OUTPUT: The set of primes appearing in the factorization of this rational with nonzero exponent, as a sorted list.

EXAMPLES:

```
sage: (-4/17).support()
[2, 17]
```

Trying to find the support of 0 gives an arithmetic error:

```
sage: (0/1).support()
...
ArithmeticError: Support of 0 not defined.
```

trace()

Returns the trace from Q to Q of x (which is just x). This was added for compatibility with NumberFields.

OUTPUT:

- Rational - reference to self

EXAMPLES:

```
sage: (1/3).trace()
 1/3
```

AUTHORS:

- Craig Citro

val_unit()

Returns a pair: the p-adic valuation of self, and the p-adic unit of self, as a Rational.

We do not require the p be prime, but it must be at least 2. For more documentation see Integer.val_unit

INPUT:

- p - a prime

OUTPUT:

- int - the p-adic valuation of this rational
- Rational - p-adic unit part of self

EXAMPLES:

```
sage: (-4/17).val_unit(2)
(2, -1/17)
sage: (-4/17).val_unit(17)
(-1, -4)
sage: (0/1).val_unit(17)
(+Infinity, 1)
```

AUTHORS:

- David Roe (2007-04-12)

valuation()

Return the largest power of p that divides self.

INPUT:

- p - a prime number

EXAMPLES:

```
sage: x = -5/9
sage: x.valuation(5)
1
sage: x.valuation(3)
-2
sage: x.valuation(2)
0
```

Some edge cases:

```
sage: (0/1).valuation(4)
+Infinity
sage: (7/16).valuation(4)
-2
```

class **Z_to_Q**()

section()
EXAMPLES:

```
sage: QQ.coerce_map_from(ZZ).section()
Generic map:
  From: Rational Field
  To:   Integer Ring
```

clear_mpz_globals()

gmp_randrange()

init_mpz_globals()

class **int_to_Q**()

integer_rational_power()
Compute a^b as an integer, if it is integral, or return None. The positive real root is taken for even denominators.

INPUT:

```
a -- an Integer
b -- a positive Rational
```

OUTPUT:

```
`a^b` as an Integer or None
```

EXAMPLES:

```
sage: from sage.rings.rational import integer_rational_power
sage: integer_rational_power(49, 1/2)
7
sage: integer_rational_power(27, 1/3)
3
sage: integer_rational_power(-27, 1/3) is None
True
sage: integer_rational_power(-27, 2/3) is None
True
sage: integer_rational_power(512, 7/9)
128

sage: integer_rational_power(27, 1/4) is None
True
sage: integer_rational_power(-16, 1/4) is None
True

sage: integer_rational_power(0, 7/9)
0
sage: integer_rational_power(1, 7/9)
1
sage: integer_rational_power(-1, 7/9) is None
True
```

```
sage: integer_rational_power(-1, 8/9) is None
True
sage: integer_rational_power(-1, 9/8) is None
True
```

make_rational()

Make a rational number from s (a string in base 32)

INPUT:

- s - string in base 32

OUTPUT: Rational

EXAMPLES:

```
sage: (-7/15).str(32)
'-7/f'
sage: sage.rings.rational.make_rational('-7/f')
-7/15
```

pyrex_rational_reconstruction()

Find the rational reconstruction of a mod m, if it exists.

INPUT:

- a - Integer
- m - Integer

OUTPUT:

- x - rings.rational.Rational

EXAMPLES:

```
sage: Integers(100)(2/3)
34
sage: sage.rings.rational.pyrex_rational_reconstruction(34, 100)
2/3
```

rational_power_parts()

Compute rationals or integers c and d such that $a^b = c * d^b$ with d small. This is used for simplifying radicals.

INPUT:

```
a -- a Rational or Integer
b -- a Rational
factor_limit -- the limit used in factoring a
```

EXAMPLES:

```
sage: from sage.rings.rational import rational_power_parts
sage: rational_power_parts(27, 1/2)
(3, 3)
sage: rational_power_parts(-128, 3/4)
(8, -8)
sage: rational_power_parts(-4, 1/2)
(2, -1)
sage: rational_power_parts(-4, 1/3)
(1, -4)
```

```
sage: rational_power_parts(9/1000, 1/2)
(3/10, 1/10)
```

24.7 Finite Fields

Sage supports arithmetic in finite prime and extension fields. Several implementation for prime fields are implemented natively in Sage for several sizes of primes p. These implementations are

- `sage.rings.integer_mod.IntegerMod_int`,
- `sage.rings.integer_mod.IntegerMod_int64`, and
- `sage.rings.integer_mod.IntegerMod_gmp`.

Small extension fields of cardinality $< 2^{16}$ are implemented using tables of Zech logs via the Givaro C++ library (`sage.rings.finite_field_givaro.FiniteField_givaro`). While this representation is very fast it is limited to finite fields of small cardinality. Larger finite extension fields of order $q >= 2^{16}$ are internally represented as polynomials over smaller finite prime fields. If the characteristic of such a field is 2 then NTL is used internally to represent the field (`sage.rings.finite_field_ntl_gf2e.FiniteField_ntl_gf2e`). In all other case the PARI C library is used (`sage.rings.finite_field_ext_pari.FiniteField_ext_pari`).

However, this distinction is internal only and the user usually does not have to worry about it because consistency across all implementations is aimed for. In all extension field implementations the user may either specify a minimal polynomial or leave the choice to Sage.

For small finite fields the default choice are Conway polynomials.

The Conway polynomial C_n is the lexicographically first monic irreducible, primitive polynomial of degree n over $GF(p)$ with the property that for a root α of C_n we have that $\beta = \alpha^{(p^n-1)/(p^m-1)}$ is a root of C_m for all m dividing n. Sage contains a database of Conway polynomials which also can be queried independently of finite field construction.

While Sage supports basic arithmetic in finite fields some more advanced features for computing with finite fields are still not implemented. For instance, Sage does not calculate embeddings of finite fields yet.

EXAMPLES:

```
sage: k = GF(5); type(k)
<class 'sage.rings.finite_field_prime_modn.FiniteField_prime_modn'>


sage: k = GF(5^2,'c'); type(k)
<type 'sage.rings.finite_field_givaro.FiniteField_givaro'>


sage: k = GF(2^16,'c'); type(k)
<type 'sage.rings.finite_field_ntl_gf2e.FiniteField_ntl_gf2e'>


sage: k = GF(3^16,'c'); type(k)
<class 'sage.rings.finite_field_ext_pari.FiniteField_ext_pari'>
```

Finite Fields support iteration, starting with 0.

```
sage: k = GF(9, 'a')
sage: for i,x in enumerate(k):  print i,x
0 0
1 2*a
2 a + 1
3 a + 2
4 2
5 a
6 2*a + 2
7 2*a + 1
8 1
sage: for a in GF(5):
...    print a
0
1
2
3
4
```

We output the base rings of several finite fields.

```
sage: k = GF(3); type(k)
<class 'sage.rings.finite_field_prime_modn.FiniteField_prime_modn'>
sage: k.base_ring()
Finite Field of size 3
```

```
sage: k = GF(9,'alpha'); type(k)
<type 'sage.rings.finite_field_givaro.FiniteField_givaro'>
sage: k.base_ring()
Finite Field of size 3
```

```
sage: k = GF(3^40,'b'); type(k)
<class 'sage.rings.finite_field_ext_pari.FiniteField_ext_pari'>
sage: k.base_ring()
Finite Field of size 3
```

Further examples:

```
sage: GF(2).is_field()
True
sage: GF(next_prime(10^20)).is_field()
True
sage: GF(19^20,'a').is_field()
True
sage: GF(8,'a').is_field()
True
```

AUTHORS:

- William Stein: initial version
- Robert Bradshaw: prime field implementation
- Martin Albrecht: Givaro and ntl.GF2E implementations

class FiniteFieldFactory()

Return the globally unique finite field of given order with generator labeled by the given name and possibly with given modulus.

INPUT:

- `order` - int
- `name` - string; must be specified if not a prime field
- `modulus` - (optional) defining polynomial for field, i.e., generator of the field will be a root of this polynomial; if not specified the choice of defininining polynomials can be arbitrary.
- `elem_cache` - cache all elements to avoid creation time (default: order < 500)
- `check_irreducible` - verify that the polynomial modulus is irreducible
- `args` - additional parameters passed to finite field implementations
- `kwds` - additional keyword parameters passed to finite field implementations

ALIAS: You can also use GF instead of FiniteField - they are identical.

EXAMPLES:

```
sage: k.<a> = FiniteField(9); k
Finite Field in a of size 3^2
sage: parent(a)
Finite Field in a of size 3^2
sage: charpoly(a, 'y')
y^2 + 2*y + 2
```

```
sage: F.<x> = GF(5)[]
sage: K.<a> = GF(5**5, name='a', modulus=x^5 - x +1 )
sage: f = K.modulus(); f
x^5 + 4*x + 1
sage: type(f)
 <type 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
```

The modulus must be irreducible:

```
sage: K.<a> = GF(5**5, name='a', modulus=x^5 - x )
...
ValueError: finite field modulus must be irreducible but it is not
```

You can't accidently fool the constructor into thinking the modulus is irreducible when it isn't mod p, since it actually tests irreducibility modulo p.

```
sage: F.<x> = QQ[]
sage: factor(x^5+2)
x^5 + 2
sage: K.<a> = GF(5**5, name='a', modulus=x^5 + 2 )
...
ValueError: finite field modulus must be irreducible but it is not
```

If you wish to live dangerously, you can tell the constructor not to test irreducibility using check_irreducible=False, but this can easily lead to crashes and hangs - so do not do it unless you know that the modulus really is irreducible!

```
sage: F.<x> = GF(5)[]
sage: K.<a> = GF(5**2, name='a', modulus=x^2 + 2, check_irreducible=False)
```

For example, you may print finite field elements as integers. This currently only works if the order of field is $< 2^{16}$, though.

```
sage: k.<a> = GF(2^8,repr='int')
sage: a
2
```

The order of a finite field must be a prime power:

```
sage: GF(100)
...
ValueError: order of finite field must be a prime power
```

Finite fields with random modulus are not cached:

```
sage: k.<a> = GF(2^17,modulus='random')
sage: n.<a> = GF(2^17,modulus='random')
sage: n is k
False
```

We check that various ways of creating the same finite field yield the same object.

```
sage: K = GF(7, 'a')
sage: L = GF(7, 'b')
sage: K is L
True
sage: K = GF(4,'a'); K.modulus()
x^2 + x + 1
sage: L = GF(4,'a', K.modulus())
sage: K is L
True
sage: M = GF(4,'a', K.modulus().change_variable_name('y'))
sage: K is M
True
```

create_key_and_extra_args(*order, name=None, modulus=None, names=None, impl=None, **kwds*)

EXAMPLES:

```
sage: GF.create_key_and_extra_args(9, 'a')
((9, ('a',), None, None, '{}'), {})
sage: GF.create_key_and_extra_args(9, 'a', foo='value')
((9, ('a',), None, None, "{'foo': 'value'}"), {'foo': 'value'})
```

create_object(*version, key, check_irreducible=True, elem_cache=None, names=None, **kwds*)

EXAMPLES:

```
sage: K = GF(19)
sage: loads(dumps(K)) is K
True
```

other_keys(*key, K*)

EXAMPLES:

```
sage: key, extra = GF.create_key_and_extra_args(9, 'a'); key
(9, ('a',), None, None, '{}')
sage: K = GF.create_object(0, key); K
Finite Field in a of size 3^2
sage: GF.other_keys(key, K)
[(9, ('a',), x^2 + 2*x + 2, None, '{}'),
 (9, ('a',), x^2 + 2*x + 2, 'givaro', '{}')]
```

conway_polynomial(*p, n*)

Return the Conway polynomial of degree n over GF(p), which is loaded from a table.

If the requested polynomial is not known, this function raises a RuntimeError exception.

INPUT:

- p - int
- n - int

OUTPUT:

- Polynomial - a polynomial over the prime finite field GF(p).

Note: The first time this function is called a table is read from disk, which takes a fraction of a second. Subsequent calls do not require reloading the table.

See also the ConwayPolynomials() object, which is a table of Conway polynomials. For example, if c=ConwayPolynomials(), then c.primes() is a list of all primes for which the polynomials are known, and for a given prime p, c.degree(p) is a list of all degrees for which the Conway polynomials are known.

EXAMPLES:

```
sage: conway_polynomial(2,5)
x^5 + x^2 + 1
sage: conway_polynomial(101,5)
x^5 + 2*x + 99
sage: conway_polynomial(97,101)
...
RuntimeError: requested conway polynomial not in database.
```

exists_conway_polynomial(*p, n*)

Return True if the Conway polynomial over F_p of degree n is in the database and False otherwise.

If the Conway polynomial is in the database, to obtain it use the command conway_polynomial(p,n).

EXAMPLES:

```
sage: exists_conway_polynomial(2,3)
True
sage: exists_conway_polynomial(2,-1)
False
sage: exists_conway_polynomial(97,200)
False
sage: exists_conway_polynomial(6,6)
False
```

is_PrimeFiniteField(*x*)

Returns True if x is a prime finite field.

EXAMPLES:

```
sage: from sage.rings.finite_field import is_PrimeFiniteField
sage: is_PrimeFiniteField(QQ)
False
sage: is_PrimeFiniteField(GF(7))
True
sage: is_PrimeFiniteField(GF(7^2,'a'))
False
sage: is_PrimeFiniteField(GF(next_prime(10^90,proof=False)))
True
```

24.8 Elements of Finite Fields

EXAMPLES:

```
sage: K = FiniteField(2)
sage: V = VectorSpace(K,3)
sage: w = V([0,1,2])
sage: K(1)*w
(0, 1, 0)
```

We do some arithmetic involving a bigger field and a Conway polynomial, i.e., we verify compatibility condition.

```
sage: f = conway_polynomial(2,63)
sage: K.<a> = GF(2**63, name='a', modulus=f)
sage: n = f.degree()
sage: m = 3;
sage: e = (2^n - 1) / (2^m - 1)
sage: c = a^e
sage: conway = conway_polynomial(2,m)
sage: conway(c) == 0
True
```

class **FiniteField_ext_pariElement** (*parent, value, check=True*)

An element of a finite field.

Create elements by first defining the finite field F, then use the notation F(n), for n an integer. or let a = F.gen() and write the element in terms of a.

EXAMPLES:

```
sage: K = FiniteField(10007^10, 'a')
sage: a = K.gen(); a
a
sage: loads(a.dumps()) == a
True
sage: K = GF(10007)
sage: a = K(938); a
938
sage: loads(a.dumps()) == a
True
```

TESTS:

```
sage: K.<a> = GF(2^16)
sage: K(0).is_zero()
True
sage: (a - a).is_zero()
True
sage: a - a
0
sage: a == a
True
sage: a - a == 0
True
sage: a - a == K(0)
True
```

copy()
Return a copy of this element.
EXAMPLES:

```
sage: from sage.rings.finite_field_ext_pari import FiniteField_ext_pari
sage: k = FiniteField_ext_pari(3**3,'a')
sage: a = k(5)
sage: a
2
sage: a.copy()
2
sage: b = a.copy()
sage: a == b
True
sage: a is b
False
sage: a is a
True
```

is_square()
Returns True if and only if this element is a perfect square.
EXAMPLES:

```
sage: from sage.rings.finite_field_ext_pari import FiniteField_ext_pari
sage: k = FiniteField_ext_pari(3**2, 'a')
sage: a = k.gen()
sage: a.is_square()
False
sage: (a**2).is_square()
True
sage: k = FiniteField_ext_pari(2**2,'a')
sage: a = k.gen()
sage: (a**2).is_square()
True
sage: k = FiniteField_ext_pari(17**5,'a'); a = k.gen()
sage: (a**2).is_square()
True
sage: a.is_square()
False

sage: k(0).is_square()
True
```

lift()
If this element lies in a prime finite field, return a lift of this element to an integer.
EXAMPLES:

```
sage: from sage.rings.finite_field_ext_pari import FiniteField_ext_pari
sage: k = GF(next_prime(10**10))
sage: a = k(17)/k(19)
sage: b = a.lift(); b
7894736858
sage: b.parent()
Integer Ring
```

log(*base*)
Return x such that $b^x = a$, where x is a and b is the base.
INPUT:

- `self` - finite field element

- b - finite field element that generates the multiplicative group.

OUTPUT: Integer x such that $a^x = b$, if it exists. Raises a ValueError exception if no such x exists.

EXAMPLES:

```
sage: F = GF(17)
sage: F(3^11).log(F(3))
11
sage: F = GF(113)
sage: F(3^19).log(F(3))
19
sage: F = GF(next_prime(10000))
sage: F(23^997).log(F(23))
997

sage: F = FiniteField(2^10, 'a')
sage: g = F.gen()
sage: b = g; a = g^37
sage: a.log(b)
37
sage: b^37; a
a^8 + a^7 + a^4 + a + 1
a^8 + a^7 + a^4 + a + 1
```

AUTHORS:

- David Joyner and William Stein (2005-11)

multiplicative_order()

Returns the *multiplicative* order of this element, which must be nonzero.

EXAMPLES:

```
sage: from sage.rings.finite_field_ext_pari import FiniteField_ext_pari
sage: a = FiniteField_ext_pari(5**3, 'a').0
sage: a.multiplicative_order()
124
sage: a**124
1
```

nth_root(*n, extend=False, all=False*)

Returns an nth root of self.

INPUT:

- n - integer = 1 (must fit in C int type)
- extend - bool (default: True); if True, return an nth root in an extension ring, if necessary. Otherwise, raise a ValueError if the root is not in the base ring. Warning: this option is not implemented!
- all - bool (default: False); if True, return all nth roots of self, instead of just one.

OUTPUT: If self has an nth root, returns one (if all == False) or a list of all of them (if all == True). Otherwise, raises a ValueError (if extend = False) or a NotImplementedError (if extend = True).

Warning: The 'extend' option is not implemented (yet).

AUTHORS:

- David Roe (2007-10-3)

EXAMPLES:

```
sage: k.<a> = GF(29^5)
sage: b = a^2 + 5*a + 1
sage: b.nth_root(5)
```

```
19*a^4 + 2*a^3 + 2*a^2 + 16*a + 3
sage: b.nth_root(7)
...
ValueError: no nth root
sage: b.nth_root(4, all=True)
[]
```

polynomial()

Elements of a finite field are represented as a polynomial modulo a modulus. This function returns the representing polynomial as an element of the polynomial ring over the prime finite field, with the same variable as the finite field.

EXAMPLES:

The default variable is a:

```
sage: from sage.rings.finite_field_ext_pari import FiniteField_ext_pari
sage: k = FiniteField_ext_pari(3**2,'a')
sage: k.gen().polynomial()
a
```

The variable can be any string.

```
sage: k = FiniteField(3**4, "alpha")
sage: a = k.gen()
sage: a.polynomial()
alpha
sage: (a**2 + 1).polynomial()
alpha^2 + 1
sage: (a**2 + 1).polynomial().parent()
Univariate Polynomial Ring in alpha over Finite Field of size 3
```

rational_reconstruction()

If the parent field is a prime field, uses rational reconstruction to try to find a lift of this element to the rational numbers.

EXAMPLES:

```
sage: from sage.rings.finite_field_ext_pari import FiniteField_ext_pari
sage: k = GF(97)
sage: a = k(RationalField()('2/3'))
sage: a
33
sage: a.rational_reconstruction()
2/3
```

sqrt(*extend=False, all=False*)

See self.square_root().

INPUT:

- `extend` - ignored

square_root(*extend=False, all=False*)

The square root function.

INPUT:

- `extend` - bool (default: True); if True, return a square root in an extension ring, if necessary. Otherwise, raise a ValueError if the root is not in the base ring. Warning: this option is not implemented!
- `all` - bool (default: False); if True, return all square roots of self, instead of just one.

> **Warning:** The 'extend' option is not implemented (yet).

EXAMPLES:

```
sage: from sage.rings.finite_field_ext_pari import FiniteField_ext_pari
sage: F = FiniteField_ext_pari(7^2, 'a')
sage: F(2).square_root()
4
sage: F(3).square_root()
5*a + 1
sage: F(3).square_root()**2
3
sage: F(4).square_root()
5
sage: K = FiniteField_ext_pari(7^3, 'alpha')
sage: K(3).square_root()
...
ValueError: must be a perfect square.
```

is_FiniteFieldElement(*x*)

Returns if x is a finite field element.

EXAMPLE:

```
sage: from sage.rings.finite_field_element import is_FiniteFieldElement
sage: is_FiniteFieldElement(1)
False
sage: is_FiniteFieldElement(IntegerRing())
False
sage: is_FiniteFieldElement(GF(5)(2))
True
```

CHAPTER

TWENTYFIVE

FIXED AND ARBITRARY PRECISION NUMERICAL FIELDS

Sage supports two optimized fixed precision fields for numerical computation, the real double (RealDoubleField) and complex double fields (ComplexDoubleField). Sage also supports arbitrary precision real (RealField) and complex fields (ComplexField), and real and complex interval arithmetic (RealIntervalField and ComplexIntervalField).

Real and complex double elements are optimized implementations that use the GNU Scientific Library for arithmetic and some special functions. Arbitrary precision real and complex numbers are implemented using the MPFR library, which builds on GMP. (Note that Sage doesn't currently use the MPC library.) The interval arithmetic field is implemented using the MPFI library.

In many cases the PARI C-library is used to compute special functions when implementations aren't otherwise available.

25.1 Double Precision Real Numbers

EXAMPLES:

We create the real double vector space of dimension 3:

```
sage: V = RDF^3; V
Vector space of dimension 3 over Real Double Field
```

Notice that this space is unique.

```
sage: V is RDF^3
True
sage: V is FreeModule(RDF, 3)
True
sage: V is VectorSpace(RDF, 3)
True
```

Also, you can instantly create a space of large dimension.

```
sage: V = RDF^10000
```

class RealDoubleElement()

An approximation to a real number using double precision floating point numbers. Answers derived from calculations with such approximations may differ from what they would be if those calculations were performed with true real numbers. This is due to the rounding errors inherent to finite precision calculations.

NaN()

EXAMPLES:

```
sage: RDF.NaN()
NaN
```

abs()

Returns the absolute value of self.

EXAMPLES:

```
sage: RDF(1e10).abs()
10000000000.0
sage: RDF(-1e10).abs()
10000000000.0
```

acosh()

Returns the hyperbolic inverse cosine of this number

EXAMPLES:

```
sage: q = RDF.pi()/2
sage: i = q.cosh() ; i
2.50917847866
sage: abs(i.acosh()-q) < 1e-15
True
```

agm()

Return the arithmetic-geometric mean of self and other. The arithmetic-geometric mean is the common limit of the sequences u_n and v_n, where u_0 is self, v_0 is other, u_{n+1} is the arithmetic mean of u_n and v_n, and v_{n+1} is the geometric mean of u_n and v_n. If any operand is negative, the return value is `NaN`.

EXAMPLES:

```
sage: a = RDF(1.5)
sage: b = RDF(2.3)
sage: a.agm(b)
1.87864845581
```

The arithmetic-geometric mean always lies between the geometric and arithmetic mean.

```
sage: sqrt(a*b) < a.agm(b) < (a+b)/2
True
```

algdep()

Returns a polynomial of degree at most n which is approximately satisfied by this number. Note that the returned polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

EXAMPLE:

```
sage: r = RDF(2).sqrt(); r
1.41421356237
sage: r.algdep(5)
x^2 - 2
```

algebraic_dependency()

Returns a polynomial of degree at most n which is approximately satisfied by this number. Note that the returned polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

EXAMPLE:

```
sage: r = sqrt(RDF(2)); r
1.41421356237
sage: r.algdep(5)
x^2 - 2
```

arccos()
Returns the inverse cosine of this number
EXAMPLES:

```
sage: q = RDF.pi()/3
sage: i = q.cos()
sage: i.arccos() == q
True
```

arcsin()
Returns the inverse sine of this number
EXAMPLES:

```
sage: q = RDF.pi()/5
sage: i = q.sin()
sage: i.arcsin() == q
True
```

arcsinh()
Returns the hyperbolic inverse sine of this number
EXAMPLES:

```
sage: q = RDF.pi()/2
sage: i = q.sinh() ; i
2.30129890231
sage: abs(i.arcsinh()-q) < 1e-15
True
```

arctan()
Returns the inverse tangent of this number
EXAMPLES:

```
sage: q = RDF.pi()/5
sage: i = q.tan()
sage: i.arctan() == q
True
```

arctanh()
Returns the hyperbolic inverse tangent of this number
EXAMPLES:

```
sage: q = RDF.pi()/2
sage: i = q.tanh() ; i
0.917152335667
sage: i.arctanh() - q      # output is random, depending on arch.
-4.4408920985e-16
```

ceil()
Returns the ceiling of this number
OUTPUT: integer
EXAMPLES:

```
sage: RDF(2.99).ceil()
3
```

```
sage: RDF(2.00).ceil()
2
sage: RDF(-5/2).ceil()
-2
```

ceiling()
Returns the ceiling of this number
OUTPUT: integer
EXAMPLES:

```
sage: RDF(2.99).ceil()
3
sage: RDF(2.00).ceil()
2
sage: RDF(-5/2).ceil()
-2
```

cos()
Returns the cosine of this number
EXAMPLES:

```
sage: t=RDF.pi()/2
sage: t.cos()
6.12323399574e-17
```

cosh()
Returns the hyperbolic cosine of this number
EXAMPLES:

```
sage: q = RDF.pi()/12
sage: q.cosh()
1.0344656401
```

coth()
This function returns the hyperbolic cotangent.
EXAMPLES:

```
sage: RDF(pi).coth()
1.0037418732
sage: CDF(pi).coth()
1.0037418732
```

csch()
This function returns the hyperbolic cosecant.
EXAMPLES:

```
sage: RDF(pi).csch()
0.08658953753
sage: CDF(pi).csch()
0.08658953753
```

cube_root()
Return the cubic root (defined over the real numbers) of self.
EXAMPLES:

```
sage: r = RDF(125.0); r.cube_root()
5.0
sage: r = RDF(-119.0)
sage: r.cube_root()^3 - r          # output is random, depending on arch.
0.0
```

erf()
Returns the value of the error function on self.
EXAMPLES:

```
sage: RDF(6).erf()
1.0
```

exp()
Returns $e^{\texttt{self}}$
EXAMPLES:

```
sage: r = RDF(0.0)
sage: r.exp()
1.0

sage: r = RDF('32.3')
sage: a = r.exp(); a
1.06588847275e+14
sage: a.log()
32.3

sage: r = RDF('-32.3')
sage: r.exp()
9.3818445885e-15

sage: RDF(1000).exp()
+infinity
```

exp10()
Returns $10^{\texttt{self}}$
EXAMPLES:

```
sage: r = RDF(0.0)
sage: r.exp10()
1.0

sage: r = RDF(32.0)
sage: r.exp10()
1e+32

sage: r = RDF(-32.3)
sage: r.exp10()
5.01187233627e-33
```

exp2()
Returns $2^{\texttt{self}}$
EXAMPLES:

```
sage: r = RDF(0.0)
sage: r.exp2()
1.0

sage: r = RDF(32.0)
sage: r.exp2()
4294967296.0

sage: r = RDF(-32.3)
sage: r.exp2()
1.89117248253e-10
```

floor()
Returns the floor of this number
EXAMPLES:

```
sage: RDF(2.99).floor()
2
sage: RDF(2.00).floor()
2
sage: RDF(-5/2).floor()
-3
```

frac()
frac returns a real number > -1 and < 1. it satisfies the relation: x = x.trunc() + x.frac()
EXAMPLES:

```
sage: RDF(2.99).frac()
0.99
sage: RDF(2.50).frac()
0.5
sage: RDF(-2.79).frac()
-0.79
```

gamma()
The Euler gamma function. Return gamma of self.
EXAMPLES:

```
sage: RDF(6).gamma()
120.0
sage: RDF(1.5).gamma()
0.886226925453
```

hypot()
Computes the value $\sqrt{(self^2 + other^2)}$ in such a way as to avoid overflow.
EXAMPLES:

```
sage: x = RDF(4e300); y = RDF(3e300);
sage: x.hypot(y)
5e+300
sage: sqrt(x^2+y^2) # overflow
+infinity
```

imag()
Returns the imaginary part of this number. (hint: it's zero.)
EXAMPLES:

```
sage: a = RDF(3)
sage: a.imag()
0.0
```

integer_part()
If in decimal this number is written n.defg, returns n.
EXAMPLES:

```
sage: r = RDF('-1.6')
sage: a = r.integer_part(); a
-1
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: r = RDF(0.0/0.0)
sage: a = r.integer_part()
```

```
...
TypeError: Attempt to get integer part of NaN
```

is_NaN()
EXAMPLES:

```
sage: RDF(1).is_NaN()
False
sage: a = RDF(0)/RDF(0)
sage: a.is_NaN()
True
```

is_infinity()
EXAMPLES:

```
sage: a = RDF(2); b = RDF(0)
sage: (a/b).is_infinity()
True
sage: (b/a).is_infinity()
False
```

is_negative_infinity()
EXAMPLES:

```
sage: a = RDF(2)/RDF(0)
sage: a.is_negative_infinity()
False
sage: a = RDF(-3)/RDF(0)
sage: a.is_negative_infinity()
True
```

is_positive_infinity()
EXAMPLES:

```
sage: a = RDF(1)/RDF(0)
sage: a.is_positive_infinity()
True
sage: a = RDF(-1)/RDF(0)
sage: a.is_positive_infinity()
False
```

is_square()
Returns whether or not this number is a square in this field. For the real numbers, this is True if and only if self is non-negative.
EXAMPLES:

```
sage: RDF(3.5).is_square()
True
sage: RDF(0).is_square()
True
sage: RDF(-4).is_square()
False
```

log()
EXAMPLES:

```
sage: RDF(2).log()
0.69314718056
sage: RDF(2).log(2)
1.0
sage: RDF(2).log(pi)
```

```
0.605511561398
sage: RDF(2).log(10)
0.301029995664
sage: RDF(2).log(1.5)
1.70951129135
sage: RDF(0).log()
-infinity
sage: RDF(-1).log()
NaN
```

log10()

Returns log to the base 10 of self

EXAMPLES:

```
sage: r = RDF('16.0'); r.log10()
1.20411998266
sage: r.log() / RDF(log(10))
1.20411998266
sage: r = RDF('39.9'); r.log10()
1.60097289569
```

log2()

Returns log to the base 2 of self

EXAMPLES:

```
sage: r = RDF(16.0)
sage: r.log2()
4.0

sage: r = RDF(31.9); r.log2()
4.99548451888
```

logpi()

Returns log to the base pi of self

EXAMPLES:

```
sage: r = RDF(16); r.logpi()
2.42204624559
sage: r.log() / RDF(log(pi))
2.42204624559
sage: r = RDF('39.9'); r.logpi()
3.22030233461
```

multiplicative_order()

Returns n such that self^n == 1.

Only ± 1 have finite multiplicative order.

EXAMPLES:

```
sage: RDF(1).multiplicative_order()
1
sage: RDF(-1).multiplicative_order()
2
sage: RDF(3).multiplicative_order()
+Infinity
```

nan()

EXAMPLES:

```
sage: RDF.nan()
NaN
```

nth_root()

Returns the n^{th} root of self.

INPUT:

- n - an integer

OUTPUT: an real or complex double

The output is complex if self is negative and n is even.

EXAMPLES:

```
sage: r = RDF(-125.0); r.nth_root(3)
-5.0
sage: r.nth_root(5)
-2.6265278044
sage: RDF(-2).nth_root(5)^5
-2.0
sage: RDF(-1).nth_root(5)^5
-1.0
sage: RDF(3).nth_root(10)^10
3.0
sage: RDF(-1).nth_root(2)
6.12323399574e-17 + 1.0*I
sage: RDF(-1).nth_root(4)
0.707106781187 + 0.707106781187*I
```

parent()

Return the real double field, which is the parent of self.

EXAMPLES:

```
sage: a = RDF(2.3)
sage: a.parent()
Real Double Field
sage: parent(a)
Real Double Field
```

prec()

Returns the precision of this number in bits.

Always returns 53.

EXAMPLES:

```
sage: RDF(0).prec()
53
```

real()

Returns itself - we're already real.

EXAMPLES:

```
sage: a = RDF(3)
sage: a.real()
3.0
```

restrict_angle()

Returns a number congruent to self mod 2π that lies in the interval $(-\pi, \pi]$.

Specifically, it is the unique $x \in (-\pi, \pi]$ such that `self = x + 2pi n` for some $n \in \mathbf{Z}$.

EXAMPLES:

```
sage: RDF(pi).restrict_angle()
3.14159265359
sage: RDF(pi + 1e-10).restrict_angle()
-3.14159265349
sage: RDF(1+10^10*pi).restrict_angle()
0.9999977606...
```

round()

Given real number x, rounds up if fractional part is greater than .5, rounds down if fractional part is less than .5.

EXAMPLES:

```
sage: RDF(0.49).round()
0
sage: a=RDF(0.51).round(); a
1
```

sech()

This function returns the hyperbolic secant.

EXAMPLES:

```
sage: RDF(pi).sech()
0.0862667383341
sage: CDF(pi).sech()
0.0862667383341
```

sign()

Returns -1,0, or 1 if self is negative, zero, or positive; respectively.

EXAMPLES:

```
sage: RDF(-1.5).sign()
-1
sage: RDF(0).sign()
0
sage: RDF(2.5).sign()
1
```

sin()

Returns the sine of this number

EXAMPLES:

```
sage: RDF(2).sin()
0.909297426826
```

sincos()

Returns a pair consisting of the sine and cosine.

EXAMPLES:

```
sage: t = RDF.pi()/6
sage: t.sincos()
(0.5, 0.866025403784)
```

sinh()

Returns the hyperbolic sine of this number

EXAMPLES:

```
sage: q = RDF.pi()/12
sage: q.sinh()
0.264800227602
```

sqrt()

The square root function.

INPUT:

- `extend` - bool (default: True); if True, return a square root in a complex field if necessary if self is negative; otherwise raise a ValueError
- `all` - bool (default: False); if True, return a list of all square roots.

EXAMPLES:

```
sage: r = RDF(4.0)
sage: r.sqrt()
2.0
sage: r.sqrt()^2 == r
True

sage: r = RDF(4344)
sage: r.sqrt()
65.9090282131
sage: r.sqrt()^2 - r
0.0

sage: r = RDF(-2.0)
sage: r.sqrt()
1.41421356237*I

sage: RDF(2).sqrt(all=True)
[1.41421356237, -1.41421356237]
sage: RDF(0).sqrt(all=True)
[0.0]
sage: RDF(-2).sqrt(all=True)
[1.41421356237*I, -1.41421356237*I]
```

str()

Return string representation of self.

EXAMPLES:

```
sage: a = RDF('4.5'); a.str()
'4.5'
sage: a = RDF('49203480923840.2923904823048'); a.str()
'4.92034809238e+13'
sage: a = RDF(1)/RDF(0); a.str()
'+infinity'
sage: a = -RDF(1)/RDF(0); a.str()
'-infinity'
sage: a = RDF(0)/RDF(0); a.str()
'NaN'
```

We verify consistency with RR (mpfr reals):

```
sage: str(RR(RDF(1)/RDF(0))) == str(RDF(1)/RDF(0))
True
sage: str(RR(-RDF(1)/RDF(0))) == str(-RDF(1)/RDF(0))
True
sage: str(RR(RDF(0)/RDF(0))) == str(RDF(0)/RDF(0))
True
```

tan()

Returns the tangent of this number

EXAMPLES:

```
sage: q = RDF.pi()/3
sage: q.tan()
1.73205080757
sage: q = RDF.pi()/6
sage: q.tan()
0.57735026919
```

tanh()

Returns the hyperbolic tangent of this number

EXAMPLES:

```
sage: q = RDF.pi()/12
sage: q.tanh()
0.255977789246
```

trunc()

Truncates this number (returns integer part).

EXAMPLES:

```
sage: RDF(2.99).trunc()
2
sage: RDF(-2.00).trunc()
-2
sage: RDF(0.00).trunc()
0
```

ulp()

Returns the unit of least precision of self, which is the weight of the least significant bit of self. Unless self is exactly a power of two, it is gap between this number and the next closest distinct number that can be represented.

EXAMPLES:

```
sage: a = RDF(1)
sage: a - a.ulp() == a
False
sage: a - a.ulp()/2 == a
True

sage: a = RDF.pi()
sage: b = a + a.ulp()
sage: (a+b)/2 in [a,b]
True

sage: a = RDF(1)/RDF(0); a
+infinity
sage: a.ulp()
+infinity
sage: (-a).ulp()
+infinity
sage: a = RR('nan')
sage: a.ulp() is a
True
```

zeta()

Return the Riemann zeta function evaluated at this real number.

Note: PARI is vastly more efficient at computing the Riemann zeta function. See the example below for how to use it.

EXAMPLES:

```
sage: RDF(2).zeta()
1.64493406685
sage: RDF.pi()^2/6
1.64493406685
sage: RDF(-2).zeta()        # slightly random-ish arch dependent output
-2.37378795339e-18
sage: RDF(1).zeta()
+infinity
```

RealDoubleField()

Return the unique instance of the Real Double Field.

EXAMPLES:

```
sage: RealDoubleField() is RealDoubleField()
True
```

class **RealDoubleField_class**()

An approximation to the field of real numbers using double precision floating point numbers. Answers derived from calculations in this approximation may differ from what they would be if those calculations were performed in the true field of real numbers. This is due to the rounding errors inherent to finite precision calculations.

EXAMPLES:

```
sage: RR == RDF
False
sage: RDF == RealDoubleField()     # RDF is the shorthand
True


sage: RDF(1)
1.0
sage: RDF(2/3)
0.666666666667
```

A TypeError is raised if the coercion doesn't make sense:

```
sage: RDF(QQ['x'].0)
...
TypeError: cannot coerce nonconstant polynomial to float
sage: RDF(QQ['x'](3))
3.0
```

One can convert back and forth between double precision real numbers and higher-precision ones, though of course there may be loss of precision:

```
sage: a = RealField(200)(2).sqrt(); a
1.41421356237309504880168872420969807856967187537694807317667
sage: b = RDF(a); b
1.41421356237
sage: a.parent()(b)
1.41421356237309514547462185873882845044136047363281250000000
sage: a.parent()(b) == b
True
sage: b == RR(a)
True
```

NaN()

EXAMPLES:

```
sage: RDF.NaN()
NaN
```

algebraic_closure()
Returns the algebraic closure of self, ie, the complex double field.
EXAMPLES:

```
sage: RDF.algebraic_closure()
Complex Double Field
```

characteristic()
Returns 0, since the field of real numbers has characteristic 0.
EXAMPLES:

```
sage: RDF.characteristic()
0
```

complex_field()
Returns the complex field with the same precision as self, ie, the complex double field.
EXAMPLES:

```
sage: RDF.complex_field()
Complex Double Field
```

construction()
Returns the functorial construction of self, namely, completion of the rational numbers with respect to the prime at ∞.
Also preserves other information that makes this field unique (i.e. the Real Double Field).
EXAMPLES:

```
sage: c, S = RDF.construction(); S
Rational Field
sage: RDF == c(S)
True
```

euler_constant()
Returns Euler's gamma constant to double precision
EXAMPLES:

```
sage: RDF.euler_constant()
0.577215664902
```

factorial()
Return the factorial of the integer n as a real number.
EXAMPLES:

```
sage: RDF.factorial(100)
9.33262154439e+157
```

gen()
Return the generator of the real double field.
EXAMPLES:

```
sage: RDF.0
1.0
sage: RDF.gens()
(1.0,)
```

is_atomic_repr()
Returns True, to signify that elements of this field print without sums, so parenthesis aren't required, e.g., in coefficients of polynomials.
EXAMPLES:

```
sage: RDF.is_atomic_repr()
True
```

is_exact()
Returns False, because doubles are not exact.
EXAMPLE:

```
sage: RDF.is_exact()
False
```

is_finite()
Returns False, since the field of real numbers is not finite. Technical note: There exists an upper bound on the double representation.
EXAMPLES:

```
sage: RDF.is_finite()
False
```

log2()
Returns log(2) to the precision of this field.
EXAMPLES:

```
sage: RDF.log2()
0.69314718056
sage: RDF(2).log()
0.69314718056
```

name()

nan()
EXAMPLES:

```
sage: RDF.nan()
NaN
```

ngens()

pi()
Returns pi to double-precision.
EXAMPLES:

```
sage: RDF.pi()
3.14159265359
sage: RDF.pi().sqrt()/2
0.886226925453
```

prec()
Return the precision of this real double field in bits.
Always returns 53.
EXAMPLES:

```
sage: RDF.prec()
53
```

random_element()
Return a random element of this real double field in the interval [min, max].
EXAMPLES:

```
sage: RDF.random_element()
0.736945423566
sage: RDF.random_element(min=100, max=110)
102.815947352
```

to_prec()

Returns the real field to the specified precision. As doubles have fixed precision, this will only return a real double field if prec is exactly 53.

EXAMPLES:

```
sage: RDF.to_prec(52)
Real Field with 52 bits of precision
sage: RDF.to_prec(53)
Real Double Field
```

zeta()

Return an n-th root of unity in the real field, if one exists, or raise a ValueError otherwise.

EXAMPLES:

```
sage: RDF.zeta()
-1.0
sage: RDF.zeta(1)
1.0
sage: RDF.zeta(5)
...
ValueError: No 5th root of unity in self
```

class ToRDF()

is_RealDoubleElement()

is_RealDoubleField()

Returns True if x is the field of real double precision numbers.

EXAMPLE:

```
sage: from sage.rings.real_double import is_RealDoubleField
sage: is_RealDoubleField(RDF)
True
sage: is_RealDoubleField(RealField(53))
False
```

pool_stats()

time_alloc()

time_alloc_list()

25.2 Double Precision Complex Numbers

Sage supports arithmetic using double-precision complex numbers. A double-precision complex number is a complex number x + I*y with x, y 64-bit (8 byte) floating point numbers (double precision).

The field `ComplexDoubleField` implements the field of all double-precision complex numbers. You can refer to this field by the shorthand CDF. Elements of this field are of type `ComplexDoubleElement`. If x and y are coercible to doubles, you can create a complex double element using `ComplexDoubleElement(x,y)`. You can coerce more general objects z to complex doubles by typing either `ComplexDoubleField(x)` or `CDF(x)`.

EXAMPLES:

```
sage: ComplexDoubleField()
Complex Double Field
sage: CDF
Complex Double Field
sage: type(CDF.0)
<type 'sage.rings.complex_double.ComplexDoubleElement'>
sage: ComplexDoubleElement(sqrt(2),3)
1.41421356237 + 3.0*I
sage: parent(CDF(-2))
Complex Double Field
```

```
sage: CC == CDF
False
sage: CDF is ComplexDoubleField()      # CDF is the shorthand
True
sage: CDF == ComplexDoubleField()
True
```

The underlying arithmetic of complex numbers is implemented using functions and macros in GSL (the GNU Scientific Library), and should be very fast. Also, all standard complex trig functions, log, exponents, etc., are implemented using GSL, and are also robust and fast. Several other special functions, e.g. eta, gamma, incomplete gamma, etc., are implemented using the PARI C library.

AUTHORS:

- William Stein (2006-09): first version

class ComplexDoubleElement()

An approximation to a complex number using double precision floating point numbers. Answers derived from calculations with such approximations may differ from what they would be if those calculations were performed with true complex numbers. This is due to the rounding errors inherent to finite precision calculations.

abs()

This function returns the magnitude of the complex number z, $|z|$.

EXAMPLES:

```
sage: CDF(2,3).abs()    # slightly random-ish arch dependent output
3.6055512754639891
```

abs2()

This function returns the squared magnitude of the complex number z, $|z|^2$.

EXAMPLES:

```
sage: CDF(2,3).abs2()
13.0
```

agm()

Return the arithmetic geometry mean of self and right.

The principal square root is always chosen.

EXAMPLES:

```
sage: i = CDF(I)
sage: (1+i).agm(2-i)
1.62780548487 + 0.136827548397*I
```

algdep()

Returns a polynomial of degree at most n which is approximately satisfied by this complex number. Note

that the returned polynomial need not be irreducible, and indeed usually won't be if z is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

EXAMPLE:

```
sage: z = (1/2)*(1 + RDF(sqrt(3)) *CDF.0); z
0.5 + 0.866025403784*I
sage: p = z.algdep(5); p
x^5 + x^2
sage: p.factor()
(x + 1) * x^2 * (x^2 - x + 1)
sage: z^2 - z + 1
2.22044604925e-16...

sage: CDF(0,2).algdep(10)
x^2 + 4
sage: CDF(1,5).algdep(2)
x^2 - 2*x + 26
```

arccos()

This function returns the complex arccosine of the complex number z, $\arccos(z)$. The branch cuts are on the real axis, less than -1 and greater than 1.

EXAMPLES:

```
sage: CDF(1,1).arccos()
0.904556894302 - 1.06127506191*I
```

arccosh()

This function returns the complex hyperbolic arccosine of the complex number z, $\operatorname{arccosh}(z)$. The branch cut is on the real axis, less than 1.

EXAMPLES:

```
sage: CDF(1,1).arccosh()
1.06127506191 + 0.904556894302*I
```

arccot()

This function returns the complex arccotangent of the complex number z, $\operatorname{arccot}(z) = \arctan(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccot()
0.553574358897 - 0.402359478109*I
```

arccoth()

This function returns the complex hyperbolic arccotangent of the complex number z, $\operatorname{arccoth}(z) = \operatorname{arctanh}(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccoth()
0.402359478109 - 0.553574358897*I
```

arccsc()

This function returns the complex arccosecant of the complex number z, $\operatorname{arccsc}(z) = \arcsin(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccsc()
0.452278447151 - 0.530637530953*I
```

arccsch()

This function returns the complex hyperbolic arccosecant of the complex number z, $\operatorname{arccsch}(z) = \arcsin(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arccsch()
0.530637530953 - 0.452278447151*I
```

arcsech()

This function returns the complex hyperbolic arcsecant of the complex number z, $\text{arcsech}(z) = \text{arccosh}(1/z)$.

EXAMPLES:

```
sage: CDF(1,1).arcsech()
0.530637530953 - 1.11851787964*I
```

arcsin()

This function returns the complex arcsine of the complex number z, $\arcsin(z)$. The branch cuts are on the real axis, less than -1 and greater than 1.

EXAMPLES:

```
sage: CDF(1,1).arcsin()
0.666239432493 + 1.06127506191*I
```

arcsinh()

This function returns the complex hyperbolic arcsine of the complex number z, $\text{arcsinh}(z)$. The branch cuts are on the imaginary axis, below -i and above i.

EXAMPLES:

```
sage: CDF(1,1).arcsinh()
1.06127506191 + 0.666239432493*I
```

arctan()

This function returns the complex arctangent of the complex number z, $\arctan(z)$. The branch cuts are on the imaginary axis, below $-i$ and above i.

EXAMPLES:

```
sage: CDF(1,1).arctan()
1.0172219679 + 0.402359478109*I
```

arctanh()

This function returns the complex hyperbolic arctangent of the complex number z, $\text{arctanh}(z)$. The branch cuts are on the real axis, less than -1 and greater than 1.

EXAMPLES:

```
sage: CDF(1,1).arctanh()
0.402359478109 + 1.0172219679*I
```

arg()

This function returns the argument of the complex number z, $\arg(z)$, where $-\pi < \arg(z) <= \pi$.

EXAMPLES:

```
sage: CDF(1,0).arg()
0.0
sage: CDF(0,1).arg()
1.57079632679
sage: CDF(0,-1).arg()
-1.57079632679
sage: CDF(-1,0).arg()
3.14159265359
```

argument()

This function returns the argument of the self, in the interval $-\pi < arg(self) \leq \pi$.

EXAMPLES:

```
sage: CDF(6).argument()
0.0
sage: CDF(i).argument()
1.57079632679
sage: CDF(-1).argument()
3.14159265359
sage: CDF(-1 - 0.000001*i).argument()
-3.14159165359
```

conj()

This function returns the complex conjugate of the complex number z, $\overline{z} = x - iy$.

EXAMPLES:

```
sage: z = CDF(2,3); z.conj()
2.0 - 3.0*I
```

conjugate()

This function returns the complex conjugate of the complex number z, $\overline{z} = x - iy$.

EXAMPLES:

```
sage: z = CDF(2,3); z.conjugate()
2.0 - 3.0*I
```

cos()

This function returns the complex cosine of the complex number z, $\cos(z) = (\exp(iz) + \exp(-iz))/2$.

EXAMPLES:

```
sage: CDF(1,1).cos()
0.833730025131 - 0.988897705763*I
```

cosh()

This function returns the complex hyperbolic cosine of the complex number z, $\cosh(z) = (\exp(z) + \exp(-z))/2$.

EXAMPLES:

```
sage: CDF(1,1).cosh()
0.833730025131 + 0.988897705763*I
```

cot()

This function returns the complex cotangent of the complex number z, $\cot(z) = 1/\tan(z)$.

EXAMPLES:

```
sage: CDF(1,1).cot()
0.217621561854 - 0.868014142896*I
```

coth()

This function returns the complex hyperbolic cotangent of the complex number z, $\coth(z) = 1/\tanh(z)$.

EXAMPLES:

```
sage: CDF(1,1).coth()
0.868014142896 - 0.217621561854*I
```

csc()

This function returns the complex cosecant of the complex number z, $\csc(z) = 1/\sin(z)$.

EXAMPLES:

```
sage: CDF(1,1).csc()
0.621518017171 - 0.303931001628*I
```

csch()

This function returns the complex hyperbolic cosecant of the complex number z, $\mathrm{csch}(z) = 1/\sinh(z)$.

EXAMPLES:

```
sage: CDF(1,1).csch()
0.303931001628 - 0.62151801717*I
```

dilog()

Returns the principal branch of the dilogarithm of x, i.e., analytic continuation of the power series

$$\log_2(x) = \sum_{n \geq 1} x^n/n^2.$$

EXAMPLES:

```
sage: CDF(1,2).dilog()
-0.0594747986738 + 2.07264797177*I
sage: CDF(10000000,10000000).dilog()
-134.411774491 + 38.793962999*I
```

eta()

Return the value of the Dedekind η function on self, intelligently computed using $\mathbb{SL}(2, \mathbf{Z})$ transformations.

INPUT:

- `self` - element of the upper half plane (if not, raises a ValueError).
- `omit_frac` - (bool, default: False), if True, omit the $e^{\pi i z/12}$ factor.

OUTPUT: a complex double number

ALGORITHM: Uses the PARI C library, but with some modifications so it always works instead of failing on easy cases involving large input (like PARI does).

The η function is

$$\eta(z) = e^{\pi i z/12} \prod_{n=1}^{\infty} (1 - e^{2\pi i n z})$$

EXAMPLES: We compute a few values of eta:

```
sage: CDF(0,1).eta()
0.768225422326
sage: CDF(1,1).eta()
0.742048775837 + 0.19883137023*I
sage: CDF(25,1).eta()
0.742048775837 + 0.19883137023*I
```

Eta works even if the inputs are large.

```
sage: CDF(0,10^15).eta()
0
sage: CDF(10^15,0.1).eta()      # slightly random-ish arch dependent output
-0.121339721991 - 0.19619461894*I
```

We compute a few values of eta, but with the fractional power of e omited.

```
sage: CDF(0,1).eta(True)
0.998129069926
```

We compute eta to low precision directly from the definition.

```
sage: z = CDF(1,1); z.eta()
0.742048775837 + 0.19883137023*I
sage: i = CDF(0,1); pi = CDF(pi)
```

```
sage: exp(pi * i * z / 12) * prod([1-exp(2*pi*i*n*z) for n in range(1,10)])
0.742048775837 + 0.19883137023*I
```

The optional argument allows us to omit the fractional part:

```
sage: z = CDF(1,1)
sage: z.eta(omit_frac=True)
0.998129069926
sage: pi = CDF(pi)
sage: prod([1-exp(2*pi*i*n*z) for n in range(1,10)])      # slightly random-ish arch depende
0.998129069926 + 4.5908467128e-19*I
```

We illustrate what happens when z is not in the upper half plane.

```
sage: z = CDF(1)
sage: z.eta()
...
ValueError: value must be in the upper half plane
```

You can also use functional notation.

```
sage: z = CDF(1,1) ; eta(z)
0.742048775837 + 0.19883137023*I
```

exp()

This function returns the complex exponential of the complex number z, $\exp(z)$.

EXAMPLES:

```
sage: CDF(1,1).exp()
1.46869393992 + 2.28735528718*I
```

We numerically verify a famous identity to the precision of a double.

```
sage: z = CDF(0, 2*pi); z
6.28318530718*I
sage: exp(z)          # somewhat random-ish output depending on platform
1.0 - 2.44921270764e-16*I
```

gamma()

Return the Gamma function evaluated at this complex number.

EXAMPLES:

```
sage: CDF(5,0).gamma()
24.0
sage: CDF(1,1).gamma()
0.498015668118 - 0.154949828302*I
sage: CDF(0).gamma()
Infinity
```

gamma_inc()

Return the incomplete Gamma function evaluated at this complex number.

EXAMPLES:

```
sage: CDF(1,1).gamma_inc(CDF(2,3))
0.00209691486365 - 0.0599819136554*I
sage: CDF(1,1).gamma_inc(5)
-0.00137813093622 + 0.00651982002312*I
sage: CDF(2,0).gamma_inc(CDF(1,1))
0.707092096346 - 0.42035364096*I
```

imag()

Return the imaginary part of this complex double.

EXAMPLES:

```
sage: a = CDF(3,-2)
sage: a.imag()
-2.0
```

is_square()

This function always returns true as $\mathbf{C}$ is algebraically closed.

EXAMPLES:

```
sage: CDF(-1).is_square()
True
```

log()

This function returns the complex natural logarithm to the given base of the complex number z, $\log(z)$. The branch cut is the negative real axis.

INPUT:

•`base` - default: e, the base of the natural logarithm

EXAMPLES:

```
sage: CDF(1,1).log()
0.34657359028 + 0.785398163397*I
```

log10()

This function returns the complex base-10 logarithm of the complex number z, $\log_{10}(z)$.

The branch cut is the negative real axis.

EXAMPLES:

```
sage: CDF(1,1).log10()
0.150514997832 + 0.34109408846*I
```

log_b()

This function returns the complex base-b logarithm of the complex number z, $\log_b(z)$. This quantity is computed as the ratio $\log(z)/\log(b)$.

The branch cut is the negative real axis.

EXAMPLES:

```
sage: CDF(1,1).log_b(10)
0.150514997832 + 0.34109408846*I
```

logabs()

This function returns the natural logarithm of the magnitude of the complex number z, $\log|z|$.

This allows for an accurate evaluation of $\log|z|$ when $|z|$ is close to 1. The direct evaluation of `log(abs(z))` would lead to a loss of precision in this case.

EXAMPLES:

```
sage: CDF(1.1,0.1).logabs()
0.0994254293726
sage: log(abs(CDF(1.1,0.1)))
0.0994254293726

sage: log(abs(ComplexField(200)(1.1,0.1)))
0.099425429372582595066319157757531449594489450091985182495705
```

norm()

This function returns the squared magnitude of the complex number z, $|z|^2$.

EXAMPLES:

```
sage: CDF(2,3).norm()
13.0
```

nth_root()

The n-th root function.

INPUT:

- `all` - bool (default: False); if True, return a list of all n-th roots.

EXAMPLES:

```
sage: a = CDF(125)
sage: a.nth_root(3)
5.0
sage: a = CDF(10, 2)
sage: [r^5 for r in a.nth_root(5, all=True)]
[10.0 + 2.0*I, 10.0 + 2.0*I, 10.0 + 2.0*I, 10.0 + 2.0*I, 10.0 + 2.0*I]
sage: abs(sum(a.nth_root(111, all=True))) # random but close to zero
6.00659385991e-14
```

parent()

Return the complex double field, which is the parent of self.

EXAMPLES:

```
sage: a = CDF(2,3)
sage: a.parent()
Complex Double Field
sage: parent(a)
Complex Double Field
```

prec()

Returns the precision of this number (to be more similar to ComplexNumber). Always returns 53.

EXAMPLES:

```
sage: CDF(0).prec()
53
```

real()

Return the real part of this complex double.

EXAMPLES:

```
sage: a = CDF(3,-2)
sage: a.real()
3.0
```

sec()

This function returns the complex secant of the complex number z, $\sec(z) = 1/\cos(z)$.

EXAMPLES:

```
sage: CDF(1,1).sec()
0.498337030555 + 0.591083841721*I
```

sech()

This function returns the complex hyperbolic secant of the complex number z, $\mathrm{sech}(z) = 1/\cosh(z)$.

EXAMPLES:

```
sage: CDF(1,1).sech()
0.498337030555 - 0.591083841721*I
```

sin()

This function returns the complex sine of the complex number z, $\sin(z) = (\exp(iz) - \exp(-iz))/(2i)$.

EXAMPLES:

```
sage: CDF(1,1).sin()
1.29845758142 + 0.634963914785*I
```

sinh()

This function returns the complex hyperbolic sine of the complex number z, $\sinh(z) = (\exp(z) - \exp(-z))/2$.

EXAMPLES:

```
sage: CDF(1,1).sinh()
0.634963914785 + 1.29845758142*I
```

sqrt()

The square root function.

INPUT:

- `all` - bool (default: False); if True, return a list of all square roots.

If all is False, the branch cut is the negative real axis. The result always lies in the right half of the complex plane.

EXAMPLES: We compute several square roots.

```
sage: a = CDF(2,3)
sage: b = a.sqrt(); b
1.67414922804 + 0.89597747613*I
sage: b^2
2.0 + 3.0*I
sage: a^(1/2)
1.67414922804 + 0.89597747613*I
```

We compute the square root of -1.

```
sage: a = CDF(-1)
sage: a.sqrt()
1.0*I
```

We compute all square roots:

```
sage: CDF(-2).sqrt(all=True)
[1.41421356237*I, -1.41421356237*I]
sage: CDF(0).sqrt(all=True)
[0]
```

tan()

This function returns the complex tangent of the complex number z, $\tan(z) = \sin(z)/\cos(z)$.

EXAMPLES:

```
sage: CDF(1,1).tan()
0.27175258532 + 1.08392332734*I
```

tanh()

This function returns the complex hyperbolic tangent of the complex number z, $\tanh(z) = \sinh(z)/\cosh(z)$.

EXAMPLES:

```
sage: CDF(1,1).tanh()
1.08392332734 + 0.27175258532*I
```

zeta()

Return the Riemann zeta function evaluated at this complex number.

EXAMPLES:

```
sage: z = CDF(1, 1)
sage: z.zeta()
0.582158059752 - 0.926848564331*I
sage: zeta(z)
0.582158059752 - 0.926848564331*I
```

ComplexDoubleField()

Returns the field of double precision complex numbers.

EXAMPLE:

```
sage: ComplexDoubleField()
Complex Double Field
sage: ComplexDoubleField() is CDF
True
```

class **ComplexDoubleField_class**()

An approximation to the field of complex numbers using double precision floating point numbers. Answers derived from calculations in this approximation may differ from what they would be if those calculations were performed in the true field of complex numbers. This is due to the rounding errors inherent to finite precision calculations.

ALGORITHMS: Arithmetic is done using GSL (the GNU Scientific Library).

characteristic()

Return the characteristic of this complex double field, which is 0.

EXAMPLES:

```
sage: CDF.characteristic()
0
```

construction()

Returns the functorial construction of self, namely, algebraic closure of the real double field.

EXAMPLES:

```
sage: c, S = CDF.construction(); S
Real Double Field
sage: CDF == c(S)
True
```

gen()

Return the generator of the complex double field.

EXAMPLES:

```
sage: CDF.0
1.0*I
sage: CDF.gens()
(1.0*I,)
```

is_exact()

Returns whether or not this field is exact, which is always false.

EXAMPLE:

```
sage: CDF.is_exact()
False
```

ngens()

The number of generators of this complex field as an RR-algebra.

There is one generator, namely sqrt(-1).

EXAMPLES:

```
sage: CDF.ngens()
1
```

pi()

Returns pi as a double precision complex number.

EXAMPLES:

```
sage: CDF.pi()
3.14159265359
```

prec()

Return the precision of this complex double field (to be more similar to ComplexField). Always returns 53.

EXAMPLES:

```
sage: CDF.prec()
53
```

random_element()

Return a random element this complex double field with real and imaginary part bounded by xmin, xmax, ymin, ymax.

EXAMPLES:

```
sage: CDF.random_element()
-0.436810529675 + 0.736945423566*I
sage: CDF.random_element(-10,10,-10,10)
-7.08874026302 - 9.54135400334*I
sage: CDF.random_element(-10^20,10^20,-2,2)
-7.58765473764e+19 + 0.925549022839*I
```

real_double_field()

The real double field, which you may view as a subfield of this complex double field.

EXAMPLES:

```
sage: CDF.real_double_field()
Real Double Field
```

to_prec()

Returns the complex field to the specified precision. As doubles have fixed precision, this will only return a complex double field if prec is exactly 53.

EXAMPLES:

```
sage: CDF.to_prec(53)
Complex Double Field
sage: CDF.to_prec(250)
Complex Field with 250 bits of precision
```

zeta()

Return a primitive n-th root of unity in this CDF, for $n \geq 1$.

INPUT:

- n - a positive integer (default: 2)

OUTPUT: a complex n-th root of unity.

EXAMPLES:

```
sage: CDF.zeta(7)
0.623489801859 + 0.781831482468*I
sage: CDF.zeta(1)
1.0
```

```
sage: CDF.zeta()
-1.0
sage: CDF.zeta() == CDF.zeta(2)
True

sage: CDF.zeta(0.5)
...
ValueError: n must be a positive integer
sage: CDF.zeta(0)
...
ValueError: n must be a positive integer
sage: CDF.zeta(-1)
...
ValueError: n must be a positive integer
```

class FloatToCDF()

Fast morphism from anything with a __float__ method to an RDF element.

EXAMPLES:

```
sage: f = CDF.coerce_map_from(ZZ); f
Native morphism:
  From: Integer Ring
  To:   Complex Double Field
sage: f(4)
4.0
sage: f = CDF.coerce_map_from(QQ); f
Native morphism:
  From: Rational Field
  To:   Complex Double Field
sage: f(1/2)
0.5
sage: f = CDF.coerce_map_from(int); f
Native morphism:
  From: Set of Python objects of type 'int'
  To:   Complex Double Field
sage: f(3r)
3.0
sage: f = CDF.coerce_map_from(float); f
Native morphism:
  From: Set of Python objects of type 'float'
  To:   Complex Double Field
sage: f(3.5)
3.5
```

is_ComplexDoubleElement()

Return True if x is a is_ComplexDoubleElement.

EXAMPLES:

```
sage: from sage.rings.complex_double import is_ComplexDoubleElement
sage: is_ComplexDoubleElement(0)
False
sage: is_ComplexDoubleElement(CDF(0))
True
```

is_ComplexDoubleField()

Return True if x is the complex double field.

EXAMPLE:

```
sage: from sage.rings.complex_double import is_ComplexDoubleField
sage: is_ComplexDoubleField(CDF)
True
sage: is_ComplexDoubleField(ComplexField(53))
False
```

25.3 Field of Arbitrary Precision Real Numbers

AUTHORS:

- Kyle Schalm (2005-09)
- William Stein: bug fixes, examples, maintenance
- Didier Deshommes (2006-03-19): examples
- David Harvey (2006-09-20): compatibility with Element._parent
- William Stein (2006-10): default printing truncates to avoid base-2 rounding confusing (fix suggested by Bill Hart)
- Didier Deshommes: special constructor for QD numbers
- Paul Zimmermann (2008-01): added new functions from mpfr-2.3.0, replaced some, e.g., sech = 1/cosh, by their original mpfr version.
- Carl Witty (2008-02): define floating-point rank and associated functions; add some documentation
- Robert Bradshaw (2009-09): decimal literals, optimizations

This is a binding for the MPFR arbitrary-precision floating point library.

We define a class `RealField`, where each instance of `RealField` specifies a field of floating-point numbers with a specified precision and rounding mode. Individual floating-point numbers are of class `RealNumber`.

In Sage (as in MPFR), floating-point numbers of precision p are of the form $sm2^{e-p}$, where $s \in \{-1, 1\}$, $2^{p-1} \leq m < 2^p$, and $-2^{30} + 1 \leq e \leq 2^{30} - 1$; plus the special values `+0`, `-0`, `+infinity`, `-infinity`, and `NaN` (which stands for Not-a-Number).

Operations in this module which are direct wrappers of MPFR functions are "correctly rounded"; we briefly describe what this means. Assume that you could perform the operation exactly, on real nunbers, to get a result r. If this result can be represented as a floating-point number, then we return that number.

Otherwise, the result r is between two floating-point numbers. For the directed rounding modes (round to plus infinity, round to minus infinity, round to zero), we return the floating-point number in the indicated direction from r. For round to nearest, we return the floating-point number which is nearest to r.

This leaves one case unspecified: in round to nearest mode, what happens if r is exactly halfway between the two nearest floating-point numbers? In that case, we round to the number with an even mantissa (the mantissa is the number m in the representation above).

Consider the ordered set of floating-point numbers of precision p. (Here we identify `+0` and `-0`, and ignore `NaN`.) We can give a bijection between these floating-point numbers and a segment of the integers, where 0 maps to 0 and adjacent floating-point numbers map to adjacent integers. We call the integer corresponding to a given floating-point number the "floating-point rank" of the number. (This is not standard terminology; I just made it up.)

EXAMPLES: A difficult conversion:

```
sage: RR(sys.maxint)
9.22337203685478e18      # 64-bit
2.14748364700000e9       # 32-bit
```

TESTS:

```
sage: -1e30
-1.00000000000000e30
```

Make sure we don't have a new field for every new literal:

```
sage: parent(2.0) is parent(2.0)
True
sage: RealField(100, rnd='RNDZ') is RealField(100, rnd='RNDD')
False
sage: RealField(100, rnd='RNDZ') is RealField(100, rnd='RNDZ')
True
```

class QQtoRR()

class RRtoRR()

section()

EXAMPLES:

```
sage: from sage.rings.real_mpfr import RRtoRR
sage: R10 = RealField(10)
sage: R100 = RealField(100)
sage: f = RRtoRR(R100, R10)
sage: f.section()
Generic map:
  From: Real Field with 10 bits of precision
  To:   Real Field with 100 bits of precision
```

class RealField()

An approximation to the field of real numbers using floating point numbers with any specified precision. Answers derived from calculations in this approximation may differ from what they would be if those calculations were performed in the true field of real numbers. This is due to the rounding errors inherent to finite precision calculations.

See the documentation for the module sage.rings.real_mpfr for more details.

algebraic_closure()

Returns the algebraic closure of self, ie the complex field with the same precision.

catalan_constant()

Returns Catalan's constant to the precision of this field.

EXAMPLES:

```
sage: RealField(100).catalan_constant()
0.91596559417721901505460351493
```

characteristic()

Returns 0, since the field of real numbers has characteristic 0.

EXAMPLES:

```
sage: RealField(10).characteristic()
0
```

complex_field()
Return complex field of the same precision.

construction()
Returns the functorial construction of self, namely, completion of the rational numbers with respect to the prime at ∞.

Also preserves other information that makes this field unique (e.g. precision, rounding, print mode).

EXAMPLES:

```
sage: R = RealField(100, rnd='RNDU')
sage: c, S = R.construction(); S
Rational Field
sage: R == c(S)
True
```

euler_constant()
Returns Euler's gamma constant to the precision of this field.

EXAMPLES:

```
sage: RealField(100).euler_constant()
0.57721566490153286060651209008
```

factorial()
Return the factorial of the integer n as a real number.

gen()

gens()

is_atomic_repr()
Returns True, to signify that elements of this field print without sums, so parenthesis aren't required, e.g., in coefficients of polynomials.

EXAMPLES:

```
sage: RealField(10).is_atomic_repr()
True
```

is_exact()

is_finite()
Returns False, since the field of real numbers is not finite.

EXAMPLES:

```
sage: RealField(10).is_finite()
False
```

log2()
Returns log(2) to the precision of this field.

EXAMPLES:

```
sage: R=RealField(100)
sage: R.log2()
0.69314718055994530941723212146
sage: R(2).log()
0.69314718055994530941723212146
```

name()

ngens()

pi()
Returns pi to the precision of this field.

EXAMPLES:

```
sage: R = RealField(100)
sage: R.pi()
3.1415926535897932384626433833
sage: R.pi().sqrt()/2
0.88622692545275801364908374167
sage: R = RealField(150)
sage: R.pi().sqrt()/2
0.886226925452758013649083741670572591398774737
```

prec()

precision()

random_element()

Returns a uniformly distributed random number between min and max (default -1 to 1).

EXAMPLES:

```
sage: RealField(100).random_element(-5, 10)
1.9305310520925994224072377281
sage: RealField(10).random_element()
-0.84
```

TESTS:

```
sage: RealField(31).random_element()
-0.207006278
sage: RealField(32).random_element()
-0.757827933
sage: RealField(33).random_element()
-0.530834221
sage: RealField(63).random_element()
0.918013195263849341
sage: RealField(64).random_element()
-0.805114150788947694
sage: RealField(65).random_element()
0.2035927570696802284
sage: RealField(10).random_element()
-0.59
sage: RealField(10).random_element()
0.57
sage: RR.random_element()
0.931242676441124
sage: RR.random_element()
0.979095507956490
```

rounding_mode()

scientific_notation()

Set or return the scientific notation printing flag. If this flag is True then real numbers with this space as parent print using scientific notation.

INPUT:

- `status` - (bool -) optional flag

to_prec()

Returns the real field that is identical to self, except at the specified precision.

EXAMPLES:

```
sage: RR.to_prec(212)
Real Field with 212 bits of precision
sage: R = RealField(30, rnd="RNDZ")
```

```
sage: R.to_prec(300)
Real Field with 300 bits of precision and rounding RNDZ
```

zeta()

Return an n-th root of unity in the real field, if one exists, or raise a ValueError otherwise.

EXAMPLES:

```
sage: R = RealField()
sage: R.zeta()
-1.00000000000000
sage: R.zeta(1)
1.00000000000000
sage: R.zeta(5)
...
ValueError: No 5th root of unity in self
```

RealField_constructor()

RealField(prec, sci_not, rnd):

INPUT:

- `prec` - (integer) precision; default = 53 prec is the number of bits used to represent the mantissa of a floating-point number. The precision can be any integer between mpfr_prec_min() and mpfr_prec_max(). In the current implementation, mpfr_prec_min() is equal to 2.
- `sci_not` - (default: False) if True, always display using scientific notation; if False, display using scientific notation only for very large or very small numbers
- `rnd` - (string) the rounding mode
- RNDN - (default) round to nearest: Knuth says this is the best choice to prevent "floating point drift".
- RNDD - round towards minus infinity
- RNDZ - round towards zero
- RNDU - round towards plus infinity

EXAMPLES:

```
sage: RealField(10)
Real Field with 10 bits of precision
sage: RealField()
Real Field with 53 bits of precision
sage: RealField(100000)
Real Field with 100000 bits of precision
```

Note: The default precision is 53, since according to the MPFR manual: 'mpfr should be able to exactly reproduce all computations with double-precision machine floating-point numbers (double type in C), except the default exponent range is much wider and subnormal numbers are not implemented.'

class **RealLiteral**()

base

literal

class **RealNumber**()

A floating point approximation to a real number using any specified precision. Answers derived from calculations with such approximations may differ from what they would be if those calculations were performed with true real numbers. This is due to the rounding errors inherent to finite precision calculations.

The approximation is printed to slightly fewer digits than its internal precision, in order to avoid confusing roundoff issues that occur because numbers are stored internally in binary.

agm()

Return the arithmetic-geometric mean of self and other. The arithmetic-geometric mean is the common limit of the sequences u_n and v_n, where u_0 is self, v_0 is other, u_{n+1} is the arithmetic mean of u_n and v_n, and v_{n+1} is the geometric mean of u_n and v_n. If any operand is negative, the return value is `NaN`.

algdep()

Returns a polynomial of degree at most n which is approximately satisfied by this number. Note that the returned polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

EXAMPLE:

```
sage: r = sqrt(2.0); r
1.41421356237310
sage: r.algdep(5)
x^2 - 2
```

algebraic_dependency()

Returns a polynomial of degree at most n which is approximately satisfied by this number. Note that the returned polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

EXAMPLE:

```
sage: r = sqrt(2.0); r
1.41421356237310
sage: r.algdep(5)
x^2 - 2
```

arccos()

Returns the inverse cosine of this number

EXAMPLES:

```
sage: q = RR.pi()/3
sage: i = q.cos()
sage: i.arccos() == q
True
```

arccosh()

Returns the hyperbolic inverse cosine of this number

EXAMPLES:

```
sage: q = RR.pi()/2
sage: i = q.cosh() ; i
2.50917847865806
sage: q == i.arccosh()
True
```

arccoth()

Returns the inverse hyperbolic cotangent of this number

EXAMPLES:

```
sage: q = RR.pi()/5
sage: i = q.coth()
sage: i.arccoth() == q
True
```

arccsch()
Returns the inverse hyperbolic cosecant of this number
EXAMPLES:

```
sage: i = RR.pi()/5
sage: q = i.csch()
sage: q.arccsch() == i
True
```

arcsech()
Returns the inverse hyperbolic secant of this number
EXAMPLES:

```
sage: i = RR.pi()/3
sage: q = i.sech()
sage: q.arcsech() == i
True
```

arcsin()
Returns the inverse sine of this number
EXAMPLES:

```
sage: q = RR.pi()/5
sage: i = q.sin()
sage: i.arcsin() == q
True
sage: i.arcsin() - q
0.000000000000000
```

arcsinh()
Returns the hyperbolic inverse sine of this number
EXAMPLES:

```
sage: q = RR.pi()/7
sage: i = q.sinh() ; i
0.464017630492991
sage: i.arcsinh() - q
0.000000000000000
```

arctan()
Returns the inverse tangent of this number
EXAMPLES:

```
sage: q = RR.pi()/5
sage: i = q.tan()
sage: i.arctan() == q
True
```

arctanh()
Returns the hyperbolic inverse tangent of this number
EXAMPLES:

```
sage: q = RR.pi()/7
sage: i = q.tanh() ; i
0.420911241048535
sage: i.arctanh() - q
0.000000000000000
```

ceil()
Returns the ceiling of this number

OUTPUT: integer

EXAMPLES:

```
sage: (2.99).ceil()
3
sage: (2.00).ceil()
2
sage: (2.01).ceil()
3

sage: ceil(10^16 * 1.0)
10000000000000000
sage: ceil(10^17 * 1.0)
100000000000000000
sage: ceil(RR(+infinity))
...
ValueError: Calling ceil() on infinity or NaN
```

ceiling()

conjugate()

Return the complex conjugate of this real number, which is the number itself.

EXAMPLES: sage: x = RealField(100)(1.238) sage: x.conjugate() 1.2380000000000000000000000000

cos()

Returns the cosine of this number

EXAMPLES:

```
sage: t=RR.pi()/2
sage: t.cos()
6.12323399573677e-17
```

cosh()

Returns the hyperbolic cosine of this number

EXAMPLES:

```
sage: q = RR.pi()/12
sage: q.cosh()
1.03446564009551
```

cot()

Returns the cotangent of this number

EXAMPLES:

```
sage: RealField(100)(2).cot()
-0.45765755436028576375027741043
```

coth()

Returns the hyperbolic cotangent of this number

EXAMPLES:

```
sage: RealField(100)(2).coth()
1.0373147207275480958778097648
```

csc()

Returns the cosecant of this number

EXAMPLES:

```
sage: RealField(100)(2).csc()
1.0997501702946164667566973970
```

csch()
Returns the hyperbolic cosecant of this number
EXAMPLES:

```
sage: RealField(100)(2).csch()
0.27572056477178320775835148216
```

cube_root()
Return the cubic root (defined over the real numbers) of self.
EXAMPLES:

```
sage: r = 125.0; r.cube_root()
5.00000000000000
sage: r = -119.0
sage: r.cube_root()^3 - r       # illustrates precision loss
-1.42108547152020e-14
```

eint()
Returns the exponential integral of this number.
EXAMPLES:

```
sage: r = 1.0
sage: r.eint()
1.89511781635594

sage: r = -1.0
sage: r.eint()
NaN
```

erf()
Returns the value of the error function on self.
EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).erf()
0.995322265018953
sage: R(6).erf()
1.00000000000000
```

erfc()
Returns the value of the complementary error function on self, i.e., $1 -$ `erf(self)`.
EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).erfc()
0.00467773498104727
sage: R(6).erfc()
2.15197367124989e-17
```

exact_rational()
Returns the exact rational representation of this floating-point number.
EXAMPLES:

```
sage: RR(0).exact_rational()
0
sage: RR(1/3).exact_rational()
6004799503160661/18014398509481984
sage: RR(37/16).exact_rational()
37/16
sage: RR(3^60).exact_rational()
```

```
42391158275216203520420085760
sage: RR(3^60).exact_rational() - 3^60
6125652559
sage: RealField(5)(-pi).exact_rational()
-25/8
```

TESTS:

```
sage: RR('nan').exact_rational()
...
ValueError: Cannot convert NaN or infinity to rational number
sage: RR('-infinity').exact_rational()
...
ValueError: Cannot convert NaN or infinity to rational number
```

exp()

Returns $e^{\texttt{self}}$

EXAMPLES:

```
sage: r = 0.0
sage: r.exp()
1.00000000000000

sage: r = 32.3
sage: a = r.exp(); a
1.06588847274864e14
sage: a.log()
32.3000000000000

sage: r = -32.3
sage: r.exp()
9.38184458849869e-15
```

exp10()

Returns $10^{\texttt{self}}$

EXAMPLES:

```
sage: r = 0.0
sage: r.exp10()
1.00000000000000

sage: r = 32.0
sage: r.exp10()
1.00000000000000e32

sage: r = -32.3
sage: r.exp10()
5.01187233627276e-33
```

exp2()

Returns $2^{\texttt{self}}$

EXAMPLES:

```
sage: r = 0.0
sage: r.exp2()
1.00000000000000

sage: r = 32.0
sage: r.exp2()
4.29496729600000e9
```

```
sage: r = -32.3
sage: r.exp2()
1.89117248253021e-10
```

expm1()

Returns $e^{\texttt{self}} - 1$, avoiding cancellation near 0.

EXAMPLES:

```
sage: r = 1.0
sage: r.expm1()
1.71828182845905

sage: r = 1e-16
sage: exp(r)-1
0.000000000000000
sage: r.expm1()
1.00000000000000e-16
```

floor()

Returns the floor of this number

EXAMPLES:

```
sage: R = RealField()
sage: (2.99).floor()
2
sage: (2.00).floor()
2
sage: floor(RR(-5/2))
-3
sage: floor(RR(+infinity))
...
ValueError: Calling floor() on infinity or NaN
```

fp_rank()

Returns the floating-point rank of this number. That is, if you list the floating-point numbers of this precision in order, and number them starting with $0.0 \to 0$ and extending the list to positive and negative infinity, returns the number corresponding to this floating-point number.

EXAMPLES:

```
sage: RR(0).fp_rank()
0
sage: RR(0).nextabove().fp_rank()
1
sage: RR(0).nextbelow().nextbelow().fp_rank()
-2
sage: RR(1).fp_rank()
48357032784585166988247 05
sage: RR(-1).fp_rank()
-48357032784585166988247 05
sage: RR(1).fp_rank() - RR(1).nextbelow().fp_rank()
1
sage: RR(-infinity).fp_rank()
-96714065524134337702789 13
sage: RR(-infinity).fp_rank() - RR(-infinity).nextabove().fp_rank()
-1
```

fp_rank_delta()

Return the floating-point rank delta between `self` and `other`. That is, if the return value is positive, this is the number of times you have to call `.nextabove()` to get from `self` to `other`.

EXAMPLES:

```
sage: [x.fp_rank_delta(x.nextabove()) for x in
...       (RR(-infinity), -1.0, 0.0, 1.0, RR(pi), RR(infinity))]
[1, 1, 1, 1, 1, 0]
```

In the 2-bit floating-point field, one subsegment of the floating-point numbers is: 1, 1.5, 2, 3, 4, 6, 8, 12, 16, 24, 32

```
sage: R2 = RealField(2)
sage: R2(1).fp_rank_delta(R2(2))
2
sage: R2(2).fp_rank_delta(R2(1))
-2
sage: R2(1).fp_rank_delta(R2(1048576))
40
sage: R2(24).fp_rank_delta(R2(4))
-5
sage: R2(-4).fp_rank_delta(R2(-24))
-5
```

There are lots of floating-point numbers around 0:

```
sage: R2(-1).fp_rank_delta(R2(1))
4294967298
```

frac()

frac returns a real number > -1 and < 1. it satisfies the relation: x = x.trunc() + x.frac()

EXAMPLES:

```
sage: (2.99).frac()
0.990000000000000
sage: (2.50).frac()
0.500000000000000
sage: (-2.79).frac()
-0.790000000000000
```

gamma()

The Euler gamma function. Return gamma of self.

EXAMPLES:

```
sage: R = RealField()
sage: R(6).gamma()
120.000000000000
sage: R(1.5).gamma()
0.886226925452758
```

integer_part()

If in decimal this number is written n.defg, returns n.

OUTPUT: a Sage Integer

EXAMPLE:

```
sage: a = 119.41212
sage: a.integer_part()
119
sage: a = -123.4567
sage: a.integer_part()
-123
```

A big number with no decimal point:

```
sage: a = RR(10^17); a
1.00000000000000e17
sage: a.integer_part()
100000000000000000
```

is_NaN()

is_infinity()

EXAMPLES:

```
sage: a = RR('1.494') / RR(0); a
+infinity
sage: a.is_infinity()
True
sage: a = -RR('1.494') / RR(0); a
-infinity
sage: a.is_infinity()
True
sage: RR(1.5).is_infinity()
False
```

is_negative_infinity()

EXAMPLES:

```
sage: a = RR('1.494') / RR(0); a
+infinity
sage: a.is_negative_infinity()
False
sage: a = -RR('1.494') / RR(0); a
-infinity
sage: RR(1.5).is_negative_infinity()
False
sage: a.is_negative_infinity()
True
```

is_positive_infinity()

EXAMPLES:

```
sage: a = RR('1.494') / RR(0); a
+infinity
sage: a.is_positive_infinity()
True
sage: a = -RR('1.494') / RR(0); a
-infinity
sage: RR(1.5).is_positive_infinity()
False
sage: a.is_positive_infinity()
False
```

is_real()

is_square()

Returns whether or not this number is a square in this field. For the real numbers, this is True if and only if self is non-negative.

EXAMPLES:

```
sage: r = 3.5
sage: r.is_square()
True
sage: r = 0.0
sage: r.is_square()
```

```
True
sage: r = -4.0
sage: r.is_square()
False
```

is_unit()

j0()

Returns the value of the Bessel J function of order 0 at self.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).j0()
0.223890779141236
```

j1()

Returns the value of the Bessel J function of order 1 at self.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).j1()
0.576724807756873
```

jn()

Returns the value of the Bessel J function of order n at self.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).jn(3)
0.128943249474402
sage: R(2).jn(-17)
-2.65930780516787e-15
```

lngamma()

Return the logarithm of gamma of self.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(6).lngamma()
4.78749174278205
sage: R(1e10).lngamma()
2.20258509288811e11
```

log()

EXAMPLES:

```
sage: R = RealField()
sage: R(2).log()
0.693147180559945
```

log10()

Returns log to the base 10 of self

EXAMPLES:

```
sage: r = 16.0; r.log10()
1.20411998265592
sage: r.log() / log(10.0)
1.20411998265592
```

```
sage: r = 39.9; r.log10()
1.60097289568675

sage: r = 0.0
sage: r.log10()
-infinity

sage: r = -1.0
sage: r.log10()
NaN
```

log1p()

Returns log of 1 + self

EXAMPLES:

```
sage: r = 15.0; r.log1p()
2.77258872223978
sage: (r+1).log()
2.77258872223978

sage: r = 38.9; r.log1p()
3.68637632389582

sage: r = -1.0
sage: r.log1p()
-infinity

sage: r = -2.0
sage: r.log1p()
NaN
```

log2()

Returns log to the base 2 of self

EXAMPLES:

```
sage: r = 16.0
sage: r.log2()
4.00000000000000

sage: r = 31.9; r.log2()
4.99548451887751

sage: r = 0.0
sage: r.log2()
-infinity
```

multiplicative_order()

nearby_rational()

Find a rational near to self. Exactly one of max_error or max_denominator must be specified. If max_error is specified, then this returns the simplest rational in the range [self-max_error .. self+max_error]. If max_denominator is specified, then this returns the rational closest to self with denominator at most max_denominator. (In case of ties, we pick the simpler rational.)

EXAMPLES:

```
sage: (0.333).nearby_rational(max_error=0.001)
1/3
sage: (0.333).nearby_rational(max_error=1)
0
```

```
sage: (-0.333).nearby_rational(max_error=0.0001)
-257/772

sage: (0.333).nearby_rational(max_denominator=100)
1/3
sage: RR(1/3 + 1/1000000).nearby_rational(max_denominator=2999999)
777780/2333333
sage: RR(1/3 + 1/1000000).nearby_rational(max_denominator=3000000)
1000003/3000000
sage: (-0.333).nearby_rational(max_denominator=1000)
-333/1000
sage: RR(3/4).nearby_rational(max_denominator=2)
1
sage: RR(pi).nearby_rational(max_denominator=120)
355/113
sage: RR(pi).nearby_rational(max_denominator=10000)
355/113
sage: RR(pi).nearby_rational(max_denominator=100000)
312689/99532
sage: RR(pi).nearby_rational(max_denominator=1)
3
sage: RR(-3.5).nearby_rational(max_denominator=1)
-3
```

TESTS:

```
sage: RR('nan').nearby_rational(max_denominator=1000)
...
ValueError: Cannot convert NaN or infinity to rational number
sage: RR('nan').nearby_rational(max_error=0.01)
...
ValueError: Cannot convert NaN or infinity to rational number
sage: RR('infinity').nearby_rational(max_denominator=1000)
...
ValueError: Cannot convert NaN or infinity to rational number
sage: RR('infinity').nearby_rational(max_error=0.01)
...
ValueError: Cannot convert NaN or infinity to rational number
```

nextabove()

Returns the next floating-point number larger than self.

EXAMPLES:

```
sage: RR('-infinity').nextabove()
-2.09857871646739e323228496
sage: RR(0).nextabove()
2.38256490488795e-323228497
sage: RR('+infinity').nextabove()
+infinity
sage: RR(-sqrt(2)).str(truncate=False)
'-1.4142135623730951'
sage: RR(-sqrt(2)).nextabove().str(truncate=False)
'-1.4142135623730949'
```

nextbelow()

Returns the next floating-point number smaller than self.

EXAMPLES:

```
sage: RR('-infinity').nextbelow()
-infinity
sage: RR(0).nextbelow()
-2.38256490488795e-323228497
sage: RR('+infinity').nextbelow()
2.09857871646739e323228496
sage: RR(-sqrt(2)).str(truncate=False)
'-1.4142135623730951'
sage: RR(-sqrt(2)).nextbelow().str(truncate=False)
'-1.4142135623730954'
```

nexttoward()

Returns the floating-point number adjacent to self which is closer to other. If self or other is NaN, returns NaN; if self equals other, returns self.

EXAMPLES:

```
sage: (1.0).nexttoward(2).str(truncate=False)
'1.0000000000000002'
sage: (1.0).nexttoward(RR('-infinity')).str(truncate=False)
'0.99999999999999989'
sage: RR(infinity).nexttoward(0)
2.09857871646739e323228496
sage: RR(pi).str(truncate=False)
'3.1415926535897931'
sage: RR(pi).nexttoward(22/7).str(truncate=False)
'3.1415926535897936'
sage: RR(pi).nexttoward(21/7).str(truncate=False)
'3.1415926535897927'
```

nth_root()

Returns an n^{th} root of self.

INPUT:

- n - A positive number, rounded down to the nearest integer. Note that n should be less than 'sys.maxint'.
- algorithm - Set this to 1 to call mpfr directly, set this to 2 to use interval arithmetic and logarithms, or leave it at the default of 0 to choose the algorithm which is estimated to be faster.

AUTHORS:

- Carl Witty (2007-10)

EXAMPLES:

```
sage: R = RealField()
sage: R(8).nth_root(3)
2.00000000000000
sage: R(8).nth_root(3.7)     # illustrate rounding down
2.00000000000000
sage: R(-8).nth_root(3)
-2.00000000000000
sage: R(0).nth_root(3)
0.000000000000000
sage: R(32).nth_root(-1)
...
ValueError: n must be positive
sage: R(32).nth_root(1.0)
32.0000000000000
sage: R(4).nth_root(4)
1.41421356237310
```

```
sage: R(4).nth_root(40)
1.03526492384138
sage: R(4).nth_root(400)
1.00347174850950
sage: R(4).nth_root(4000)
1.00034663365385
sage: R(4).nth_root(4000000)
1.00000034657365
sage: R(-27).nth_root(3)
-3.00000000000000
sage: R(-4).nth_root(3999999)
-1.00000034657374
```

Note that for negative numbers, any even root throws an exception

```
sage: R(-2).nth_root(6)
...
ValueError: taking an even root of a negative number
```

The n^{th} root of 0 is defined to be 0, for any n

```
sage: R(0).nth_root(6)
0.000000000000000

sage: R(0).nth_root(7)
0.000000000000000
```

TESTS: The old and new algorithms should give exactly the same results in all cases.

```
sage: def check(x, n):
...       answers = []
...       for sign in (1, -1):
...           if is_even(n) and sign == -1:
...               continue
...           for rounding in ('RNDN', 'RNDD', 'RNDU', 'RNDZ'):
...               fld = RealField(x.prec(), rnd=rounding)
...               fx = fld(sign * x)
...               alg_mpfr = fx.nth_root(n, algorithm=1)
...               alg_mpfi = fx.nth_root(n, algorithm=2)
...               assert(alg_mpfr == alg_mpfi)
...               if sign == 1: answers.append(alg_mpfr)
...       return answers
```

Check some perfect powers (and nearby numbers).

```
sage: check(16.0, 4)
[2.00000000000000, 2.00000000000000, 2.00000000000000, 2.00000000000000]
sage: check((16.0).nextabove(), 4)
[2.00000000000000, 2.00000000000000, 2.00000000000001, 2.00000000000000]
sage: check((16.0).nextbelow(), 4)
[2.00000000000000, 1.99999999999999, 2.00000000000000, 1.99999999999999]
sage: check(((9.0 * 256)^7), 7)
[2304.00000000000, 2304.00000000000, 2304.00000000000, 2304.00000000000]
sage: check(((9.0 * 256)^7).nextabove(), 7)
[2304.00000000000, 2304.00000000000, 2304.00000000001, 2304.00000000000]
sage: check(((9.0 * 256)^7).nextbelow(), 7)
[2304.00000000000, 2303.99999999999, 2304.00000000000, 2303.99999999999]
sage: check(((5.0 / 512)^17), 17)
[0.00976562500000000, 0.00976562500000000, 0.00976562500000000, 0.00976562500000000]
sage: check(((5.0 / 512)^17).nextabove(), 17)
[0.00976562500000000, 0.00976562500000000, 0.00976562500000001, 0.00976562500000000]
```

```
sage: check(((5.0 / 512)^17).nextbelow(), 17)
[0.00976562500000000, 0.00976562499999999, 0.00976562500000000, 0.00976562499999999]
```

And check some non-perfect powers:

```
sage: check(2.0, 3)
[1.25992104989487, 1.25992104989487, 1.25992104989488, 1.25992104989487]
sage: check(2.0, 4)
[1.18920711500272, 1.18920711500272, 1.18920711500273, 1.18920711500272]
sage: check(2.0, 5)
[1.14869835499704, 1.14869835499703, 1.14869835499704, 1.14869835499703]
```

And some different precisions:

```
sage: check(RealField(20)(22/7), 19)
[1.0621, 1.0621, 1.0622, 1.0621]
sage: check(RealField(200)(e), 4)
[1.2840254166877414840734205680624364583362808652814630892175, 1.2840254166877414840734205680
```

parent()

EXAMPLES:

```
sage: R = RealField()
sage: a = R('1.2456')
sage: a.parent()
Real Field with 53 bits of precision
```

prec()

real()

Return the real part of self.

(Since self is a real number, this simply returns self.)

round()

Rounds self to the nearest integer. The rounding mode of the parent field has no effect on this function.

EXAMPLES:

```
sage: RR(0.49).round()
0
sage: RR(0.5).round()
1
sage: RR(-0.49).round()
0
sage: RR(-0.5).round()
-1
```

sec()

Returns the secant of this number

EXAMPLES:

```
sage: RealField(100)(2).sec()
-2.4029979617223809897546004014
```

sech()

Returns the hyperbolic secant of this number

EXAMPLES:

```
sage: RealField(100)(2).sech()
0.26580222883407969212086273982
```

sign()

simplest_rational()

Returns the simplest rational which is equal to self (in the Sage sense). Recall that Sage defines the equality operator by coercing both sides to a single type and then comparing; thus, this finds the simplest rational which (when coerced to this RealField) is equal to self.

Given rationals a/b and c/d (both in lowest terms), the former is simpler if b<d or if b==d and $|a| < |c|$.

The effect of rounding modes is slightly counter-intuitive. Consider the case of round-toward-minus-infinity. This rounding is performed when coercing a rational to a floating-point number; so the simplest_rational() of a round-to-minus-infinity number will be either exactly equal to or slightly larger than the number.

EXAMPLES:

```
sage: RRd = RealField(53, rnd='RNDD')
sage: RRz = RealField(53, rnd='RNDZ')
sage: RRu = RealField(53, rnd='RNDU')
sage: def check(x):
...       rx = x.simplest_rational()
...       assert(x == rx)
...       return rx
sage: RRd(1/3) < RRu(1/3)
True
sage: check(RRd(1/3))
1/3
sage: check(RRu(1/3))
1/3
sage: check(RRz(1/3))
1/3
sage: check(RR(1/3))
1/3
sage: check(RRd(-1/3))
-1/3
sage: check(RRu(-1/3))
-1/3
sage: check(RRz(-1/3))
-1/3
sage: check(RR(-1/3))
-1/3
sage: check(RealField(20)(pi))
355/113
sage: check(RR(pi))
245850922/78256779
sage: check(RR(2).sqrt())
131836323/93222358
sage: check(RR(1/2^210))
1/16455045573212058594672645161945060119317354277663745537946419 21
sage: check(RR(2^210))
16455045573212059508111168493759181172524338208658911348528250 88
sage: (RR(17).sqrt()).simplest_rational()^2 - 17
-1/348729667233025
sage: (RR(23).cube_root()).simplest_rational()^3 - 23
-1404915133/2647433958420390848915 84
sage: RRd5 = RealField(5, rnd='RNDD')
sage: RRu5 = RealField(5, rnd='RNDU')
sage: RR5 = RealField(5)
sage: below1 = RR5(1).nextbelow()
sage: check(RRd5(below1))
31/32
sage: check(RRu5(below1))
16/17
```

```
sage: check(below1)
21/22
sage: below1.exact_rational()
31/32
sage: above1 = RR5(1).nextabove()
sage: check(RRd5(above1))
10/9
sage: check(RRu5(above1))
17/16
sage: check(above1)
12/11
sage: above1.exact_rational()
17/16
sage: check(RR(1234))
1234
sage: check(RR5(1234))
1185
sage: check(RR5(1184))
1120
sage: RRd2 = RealField(2, rnd='RNDD')
sage: RRu2 = RealField(2, rnd='RNDU')
sage: RR2 = RealField(2)
sage: check(RR2(8))
7
sage: check(RRd2(8))
8
sage: check(RRu2(8))
7
sage: check(RR2(13))
11
sage: check(RRd2(13))
12
sage: check(RRu2(13))
13
sage: check(RR2(16))
14
sage: check(RRd2(16))
16
sage: check(RRu2(16))
13
sage: check(RR2(24))
21
sage: check(RRu2(24))
17
sage: check(RR2(-24))
-21
sage: check(RRu2(-24))
-24
```

TESTS:

```
sage: RR('nan').simplest_rational()
...
ValueError: Cannot convert NaN or infinity to rational number
sage: RR('-infinity').simplest_rational()
...
ValueError: Cannot convert NaN or infinity to rational number
```

sin()

Returns the sine of this number

EXAMPLES:

```
sage: R = RealField(100)
sage: R(2).sin()
0.90929742682568169539601986591
```

sincos()

Returns a pair consisting of the sine and cosine.

EXAMPLES:

```
sage: R = RealField()
sage: t = R.pi()/6
sage: t.sincos()
(0.500000000000000, 0.866025403784439)
```

sinh()

Returns the hyperbolic sine of this number

EXAMPLES:

```
sage: q = RR.pi()/12
sage: q.sinh()
0.264800227602271
```

sqrt()

The square root function.

INPUT:

- `extend` - bool (default: True); if True, return a square root in a complex field if necessary if self is negative; otherwise raise a ValueError
- `all` - bool (default: False); if True, return a list of all square roots.

EXAMPLES:

```
sage: r = -2.0
sage: r.sqrt()
1.41421356237310*I

sage: r = 4.0
sage: r.sqrt()
2.00000000000000
sage: r.sqrt()^2 == r
True

sage: r = 4344
sage: r.sqrt()
2*sqrt(1086)

sage: r = 4344.0
sage: r.sqrt()^2 == r
True
sage: r.sqrt()^2 - r
0.000000000000000

sage: r = -2.0
sage: r.sqrt()
1.41421356237310*I
```

str()

INPUT:

- `base` - base for output
- `no_sci` - if 2, never print using scientific notation; if 1 or True, print using scientific notation only for very large or very small numbers; if 0 or False always print with scientific notation; if None (the default), print how the parent prints.
- `e` - symbol used in scientific notation; defaults to 'e' for base=10, and '@' otherwise
- `truncate` - if True, round off the last digits in printing to lessen confusing base-2 roundoff issues.
- `skip_zeroes` - if True, skip trailing zeroes in mantissa

EXAMPLES:

```
sage: a = 61/3.0; a
20.3333333333333
sage: a.str(truncate=False)
'20.333333333333332'
sage: a.str(2)
'10100.01010101010101010101010101010101010101010101010101'
sage: a.str(no_sci=False)
'2.03333333333333e1'
sage: a.str(16, no_sci=False)
'1.4555555555555@1'
sage: b = 2.0^99
sage: b.str()
'6.33825300114115e29'
sage: b.str(no_sci=False)
'6.33825300114115e29'
sage: b.str(no_sci=True)
'6.33825300114115e29'
sage: c = 2.0^100
sage: c.str()
'1.26765060022823e30'
sage: c.str(no_sci=False)
'1.26765060022823e30'
sage: c.str(no_sci=True)
'1.26765060022823e30'
sage: c.str(no_sci=2)
'126765060022823000000000000000.'
sage: 0.5^53
1.11022302462516e-16
sage: 0.5^54
5.55111512312578e-17
sage: (0.01).str()
'0.0100000000000000'
sage: (0.01).str(skip_zeroes=True)
'0.01'
sage: (-10.042).str()
'-10.0420000000000'
sage: (-10.042).str(skip_zeroes=True)
'-10.042'
sage: (389.0).str(skip_zeroes=True)
'389.'
```

tan()

Returns the tangent of this number

EXAMPLES:

```
sage: q = RR.pi()/3
sage: q.tan()
1.73205080756888
sage: q = RR.pi()/6
```

```
sage: q.tan()
0.577350269189626
```

tanh()

Returns the hyperbolic tangent of this number

EXAMPLES:

```
sage: q = RR.pi()/11
sage: q.tanh()
0.278079429295850
```

trunc()

Truncates this number

EXAMPLES:

```
sage: (2.99).trunc()
2
sage: (-0.00).trunc()
0
sage: (0.00).trunc()
0
```

ulp()

Returns the unit of least precision of self, which is the weight of the least significant bit of self. Unless self is exactly a power of two, it is gap between this number and the next closest distinct number that can be represented.

EXAMPLES: sage: a = 1.0 sage: a + a.ulp() == a False sage: a + a.ulp()/2 == a True
sage: a = RealField(500).pi() sage: b = a + a.ulp() sage: (a+b)/2 in [a,b] True
sage: a = RR(infinity) sage: a.ulp() +infinity sage: (-a).ulp() +infinity sage: a = RR('nan') sage: a.ulp() is a True

y0()

Returns the value of the Bessel Y function of order 0 at self.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).y0()
0.510375672649745
```

y1()

Returns the value of the Bessel Y function of order 1 at self.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).y1()
-0.107032431540938
```

yn()

Returns the value of the Bessel Y function of order n at self.

EXAMPLES:

```
sage: R = RealField(53)
sage: R(2).yn(3)
-1.12778377684043
sage: R(2).yn(-17)
7.09038821729481e12
```

zeta()

Return the Riemann zeta function evaluated at this real number.

EXAMPLES:

```
sage: R = RealField()
sage: R(2).zeta()
1.64493406684823
sage: R.pi()^2/6
1.64493406684823
sage: R(-2).zeta()
0.000000000000000
sage: R(1).zeta()
+infinity
```

Computing zeta using PARI is much more efficient in difficult cases. Here's how to compute zeta with at least a given precision:

```
sage: z = pari(2).zeta(precision=53); z
1.64493406684823
sage: pari(2).zeta(precision=128).python().prec()
128
sage: pari(2).zeta(precision=65).python().prec()
128                                                    # 64-bit
96                                                     # 32-bit
```

Note that the number of bits of precision in the constructor only effects the internal precision of the pari number, which is rounded up to the nearest multiple of 32 or 64. To increase the number of digits that gets displayed you must use `pari.set_real_precision`.

```
sage: type(z)
<type 'sage.libs.pari.gen.gen'>
sage: R(z)
1.64493406684823
```

class ZZtoRR()

create_RealField()

create_RealNumber()

Return the real number defined by the string s as an element of `RealField(prec=n)`, where n potentially has slightly more (controlled by pad) bits than given by s.

INPUT:

- `s` - a string that defines a real number (or something whose string representation defines a number)
- `base` - an integer between 2 and 36
- `pad` - an integer = 0.
- `rnd` - rounding mode: RNDN, RNDZ, RNDU, RNDD
- `min_prec` - number will have at least this many bits of precision, no matter what.

EXAMPLES:

```
sage: RealNumber('2.3')
2.30000000000000
sage: RealNumber(10)
10.0000000000000
sage: RealNumber('1.000000000000000000000000000000000')
1.00000000000000000000000000000000
sage: RealField(200)(1.2)
1.20000000000000000000000000000000000000000000000000000000000000
sage: (1.2).parent() is RR
True
```

class double_toRR()

class int_toRR()

is_RealField()

is_RealNumber()
Return True if x is of type RealNumber, meaning that it is an element of the MPFR real field with some precision.

EXAMPLES:

```
sage: from sage.rings.real_mpfr import is_RealNumber
sage: is_RealNumber(2.5)
True
sage: is_RealNumber(float(2.3))
False
sage: is_RealNumber(RDF(2))
False
sage: is_RealNumber(pi)
False
```

mpfr_prec_max()

mpfr_prec_min()
Return the mpfr variable MPFR_PREC_MIN.

25.4 Field of Arbitrary Precision Complex Numbers

AUTHORS:

- William Stein (2006-01-26): complete rewrite

ComplexField(*prec=53, names=None*)
Return the complex field with real and imaginary parts having prec *bits* of precision.

EXAMPLES:

```
sage: ComplexField()
Complex Field with 53 bits of precision
sage: ComplexField(100)
Complex Field with 100 bits of precision
sage: ComplexField(100).base_ring()
Real Field with 100 bits of precision
sage: i = ComplexField(200).gen()
sage: i^2
-1.0000000000000000000000000000000000000000000000000000000000000000000000000000
```

class ComplexField_class(*prec=53*)
An approximation to the field of complex numbers using floating point numbers with any specified precision. Answers derived from calculations in this approximation may differ from what they would be if those calculations were performed in the true field of complex numbers. This is due to the rounding errors inherent to finite precision calculations.

EXAMPLES:

```
sage: C = ComplexField(); C
Complex Field with 53 bits of precision
sage: Q = RationalField()
sage: C(1/3)
0.333333333333333
```

```
sage: C(1/3, 2)
0.333333333333333 + 2.00000000000000*I
sage: C(RR.pi())
3.14159265358979
sage: C(RR.log2(), RR.pi())
0.693147180559945 + 3.14159265358979*I
```

We can also coerce rational numbers and integers into C, but coercing a polynomial will raise an exception.

```
sage: Q = RationalField()
sage: C(1/3)
0.333333333333333
sage: S = PolynomialRing(Q, 'x')
sage: C(S.gen())
...
TypeError: unable to coerce to a ComplexNumber: <class 'sage.rings.polynomial.polynomial_element
```

This illustrates precision.

```
sage: CC = ComplexField(10); CC(1/3, 2/3)
0.33 + 0.67*I
sage: CC
Complex Field with 10 bits of precision
sage: CC = ComplexField(100); CC
Complex Field with 100 bits of precision
sage: z = CC(1/3, 2/3); z
0.33333333333333333333333333333 + 0.66666666666666666666666666667*I
```

We can load and save complex numbers and the complex field.

```
sage: loads(z.dumps()) == z
True
sage: loads(CC.dumps()) == CC
True
sage: k = ComplexField(100)
sage: loads(dumps(k)) == k
True
```

This illustrates basic properties of a complex field.

```
sage: CC = ComplexField(200)
sage: CC.is_field()
True
sage: CC.characteristic()
0
sage: CC.precision()
200
sage: CC.variable_name()
'I'
sage: CC == ComplexField(200)
True
sage: CC == ComplexField(53)
False
sage: CC == 1.1
False
```

characteristic()

construction()
Returns the functorial construction of self, namely, algebraic closure of the real field with the same precision.

EXAMPLES:

```
sage: c, S = CC.construction(); S
Real Field with 53 bits of precision
sage: CC == c(S)
True
```

gen(*n=0*)

is_exact()

is_field()
Return True, since the complex numbers are a field.

EXAMPLES:

```
sage: CC.is_field()
True
```

is_finite()
Return False, since the complex numbers are infinite.

EXAMPLES:

```
sage: CC.is_finite()
False
```

ngens()

pi()

prec()

precision()

random_element(*component_max=1*)
Returns a uniformly distributed random number inside a square centered on the origin (by default, the square [-1,1]x[-1,1]).

EXAMPLES:

```
sage: [CC.random_element() for _ in range(5)]
[-0.306077326077253 - 0.0759291930543202*I, -0.838081254900233 - 0.207006276657392*I, -0.757
sage: CC6 = ComplexField(6)
sage: [CC6.random_element(2^-20) for _ in range(5)]
[-5.7e-7 + 5.4e-7*I, 8.6e-7 + 9.2e-7*I, -5.7e-7 + 6.9e-7*I, -1.2e-7 - 6.9e-7*I, 2.7e-7 + 8.3
sage: [CC6.random_element(pi^20) for _ in range(5)]
[-5.0e9*I, 2.8e9 - 5.1e9*I, 2.7e8 + 6.3e9*I, 2.7e8 - 6.4e9*I, 6.7e8 + 1.7e9*I]
```

scientific_notation(*status=None*)

to_prec(*prec*)
Returns the complex field to the specified precision.

EXAMPLES:

```
sage: CC.to_prec(10)
Complex Field with 10 bits of precision
sage: CC.to_prec(100)
Complex Field with 100 bits of precision
```

zeta(*n=2*)
Return a primitive n-th root of unity.

INPUT:

- n - an integer (default: 2)

OUTPUT: a complex n-th root of unity.

is_ComplexField(*x*)

late_import()

25.5 Arbitrary Precision Complex Numbers

AUTHORS:

- William Stein (2006-01-26): complete rewrite
- Joel B. Mohler (2006-12-16): naive rewrite into pyrex
- William Stein(2007-01): rewrite of Mohler's rewrite

class CCtoCDF()

class ComplexNumber()

A floating point approximation to a complex number using any specified precision. Answers derived from calculations with such approximations may differ from what they would be if those calculations were performed with true complex numbers. This is due to the rounding errors inherent to finite precision calculations.

EXAMPLES:

```
sage: I = CC.0
sage: b = 1.5 + 2.5*I
sage: loads(b.dumps()) == b
True
```

additive_order()

EXAMPLES:

```
sage: CC(0).additive_order()
1
sage: CC.gen().additive_order()
+Infinity
```

agm()

EXAMPLES:

```
sage: (1+CC(I)).agm(2-I)
1.62780548487271 + 0.136827548397369*I
```

algdep()

Returns a polynomial of degree at most n which is approximately satisfied by this complex number. Note that the returned polynomial need not be irreducible, and indeed usually won't be if z is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

INPUT: Type algdep? at the top level prompt. All additional parameters are passed onto the top-level algdep command.

EXAMPLE:

```
sage: C = ComplexField()
sage: z = (1/2)*(1 + sqrt(3.0) *C.0); z
0.500000000000000 + 0.866025403784439*I
```

```
sage: p = z.algdep(5); p
x^5 + x^2
sage: p.factor()
(x + 1) * x^2 * (x^2 - x + 1)
sage: z^2 - z + 1
1.11022302462516e-16
```

algebraic_dependancy()

Returns a polynomial of degree at most n which is approximately satisfied by this complex number. Note that the returned polynomial need not be irreducible, and indeed usually won't be if z is a good approximation to an algebraic number of degree less than n.

ALGORITHM: Uses the PARI C-library algdep command.

INPUT: Type algdep? at the top level prompt. All additional parameters are passed onto the top-level algdep command.

EXAMPLE:

```
sage: C = ComplexField()
sage: z = (1/2)*(1 + sqrt(3.0) *C.0); z
0.500000000000000 + 0.866025403784439*I
sage: p = z.algebraic_dependancy(5); p
x^5 + x^2
sage: p.factor()
(x + 1) * x^2 * (x^2 - x + 1)
sage: z^2 - z + 1
1.11022302462516e-16
```

arccos()

EXAMPLES:

```
sage: (1+CC(I)).arccos()
0.904556894302381 - 1.06127506190504*I
```

arccosh()

EXAMPLES:

```
sage: (1+CC(I)).arccosh()
1.06127506190504 + 0.904556894302381*I
```

arccoth()

EXAMPLES:

```
sage: ComplexField(100)(1,1).arccoth()
0.40235947810852509365018983331 - 0.55357435889704525150853273009*I
```

arccsch()

EXAMPLES:

```
sage: ComplexField(100)(1,1).arccsch()
0.53063753095251782601650945811 - 0.45227844715119068206365839783*I
```

arcsech()

EXAMPLES:

```
sage: ComplexField(100)(1,1).arcsech()
-0.53063753095251782601650945811 + 1.1185178796437059371676632938*I
```

arcsin()

EXAMPLES:

```
sage: (1+CC(I)).arcsin()
0.666239432492515 + 1.06127506190504*I
```

arcsinh()
EXAMPLES:

```
sage: (1+CC(I)).arcsinh()
1.06127506190504 + 0.666239432492515*I
```

arctan()
EXAMPLES:

```
sage: (1+CC(I)).arctan()
1.01722196789785 + 0.402359478108525*I
```

arctanh()
EXAMPLES:

```
sage: (1+CC(I)).arctanh()
0.402359478108525 + 1.01722196789785*I
```

arg()
Same as argument.
EXAMPLES:

```
sage: i = CC.0
sage: (i^2).arg()
3.14159265358979
```

argument()
The argument (angle) of the complex number, normalized so that $-\pi < \theta \leq \pi$.
EXAMPLES:

```
sage: i = CC.0
sage: (i^2).argument()
3.14159265358979
sage: (1+i).argument()
0.785398163397448
sage: i.argument()
1.57079632679490
sage: (-i).argument()
-1.57079632679490
sage: (RR('-0.001') - i).argument()
-1.57179632646156
```

conjugate()
Return the complex conjugate of this complex number.
EXAMPLES:

```
sage: i = CC.0
sage: (1+i).conjugate()
1.00000000000000 - 1.00000000000000*I
```

cos()
EXAMPLES:

```
sage: (1+CC(I)).cos()
0.833730025131149 - 0.988897705762865*I
```

cosh()
EXAMPLES:

```
sage: (1+CC(I)).cosh()
0.833730025131149 + 0.988897705762865*I
```

cotan()

EXAMPLES:

```
sage: (1+CC(I)).cotan()
0.217621561854403 - 0.868014142895925*I
sage: i = ComplexField(200).0
sage: (1+i).cotan()
0.21762156185440268136513424360523807352075436916785404091068 - 0.86801414289592494863584920
sage: i = ComplexField(220).0
sage: (1+i).cotan()
0.21762156185440268136513424360523807352075436916785404091068124239 - 0.86801414289592494863
```

coth()

EXAMPLES:

```
sage: ComplexField(100)(1,1).coth()
0.86801414289592494863584920892 - 0.21762156185440268136513424361*I
```

csc()

EXAMPLES:

```
sage: ComplexField(100)(1,1).csc()
0.62151801717042842123490780586 - 0.30393100162842645033448560451*I
```

csch()

EXAMPLES:

```
sage: ComplexField(100)(1,1).csch()
0.30393100162842645033448560451 - 0.62151801717042842123490780586*I
```

dilog()

Returns the complex dilogarithm of self. The complex dilogarithm, or Spence's function, is defined by

$Li_2(z) = -\int_0^z \frac{\log|1-\zeta|}{\zeta} d(\zeta)$

$= \sum_{k=1}^{\infty} \frac{z^k}{k}$

Note that the series definition can only be used for $|z| < 1$

EXAMPLES:

```
sage: a = ComplexNumber(1,0)
sage: a.dilog()
1.64493406684823
sage: float(pi^2/6)
1.6449340668482262

sage: b = ComplexNumber(0,1)
sage: b.dilog()
-0.205616758356028 + 0.915965594177219*I

sage: c = ComplexNumber(0,0)
sage: c.dilog()
0
```

eta()

Return the value of the Dedekind η function on self, intelligently computed using $\mathbb{SL}(2, \mathbf{Z})$ transformations.

INPUT:

- `self` - element of the upper half plane (if not, raises a ValueError).

- `omit_frac` - (bool, default: False), if True, omit the $e^{\pi iz/12}$ factor.

OUTPUT: a complex number

The η function is

$$\eta(z) = e^{\pi iz/12} \prod_{n=1}^{\infty} (1 - e^{2\pi inz})$$

ALGORITHM: Uses the PARI C library.

EXAMPLES:

First we compute $\eta(1 + i)$:

```
sage: i = CC.0
sage: z = 1+i; z.eta()
0.742048775836565 + 0.198831370229911*I
```

We compute eta to low precision directly from the definition.

```
sage: z = 1 + i; z.eta()
0.742048775836565 + 0.198831370229911*I
sage: pi = CC(pi)        # otherwise we will get a symbolic result.
sage: exp(pi * i * z / 12) * prod([1-exp(2*pi*i*n*z) for n in range(1,10)])
0.742048775836565 + 0.198831370229911*I
```

The optional argument allows us to omit the fractional part:

```
sage: z = 1 + i
sage: z.eta(omit_frac=True)
0.998129069925959 - 8.12769318...e-22*I
sage: prod([1-exp(2*pi*i*n*z) for n in range(1,10)])
0.998129069925958 + 4.59099857829247e-19*I
```

We illustrate what happens when z is not in the upper half plane.

```
sage: z = CC(1)
sage: z.eta()
...
ValueError: value must be in the upper half plane
```

You can also use functional notation.

```
sage: eta(1+CC(I))
0.742048775836565 + 0.198831370229911*I
```

exp()

Compute exp(z).

EXAMPLES:

```
sage: i = ComplexField(300).0
sage: z = 1 + i
sage: z.exp()
1.46869393991588515713896759732660426132695673662900872279767567631093696585951213872272450
```

gamma()

Return the Gamma function evaluated at this complex number.

EXAMPLES:

```
sage: i = ComplexField(30).0
sage: (1+i).gamma()
0.49801567 - 0.15494983*I
```

TESTS:

```
sage: CC(0).gamma()
Infinity

sage: CC(-1).gamma()
Infinity
```

gamma_inc()

Return the incomplete Gamma function evaluated at this complex number.

EXAMPLES:

```
sage: C, i = ComplexField(30).objgen()
sage: (1+i).gamma_inc(2 + 3*i)
0.0020969149 - 0.059981914*I
sage: (1+i).gamma_inc(5)
-0.0013781309 + 0.0065198200*I
sage: C(2).gamma_inc(1 + i)
0.70709210 - 0.42035364*I
sage: gamma_inc(2, 1 + i)
0.70709210 - 0.42035364*I
sage: gamma_inc(2, 5)
0.0404276819945128
```

imag()

Return imaginary part of self.

EXAMPLES:

```
sage: i = ComplexField(100).0
sage: z = 2 + 3*i
sage: x = z.imag(); x
3.0000000000000000000000000000
sage: x.parent()
Real Field with 100 bits of precision
```

is_imaginary()

Return True if self is imaginary, i.e. has real part zero.

EXAMPLES:

```
sage: CC(1.23*i).is_imaginary()
True
sage: CC(1+i).is_imaginary()
False
```

is_real()

Return True if self is real, i.e. has imaginary part zero.

EXAMPLES:

```
sage: CC(1.23).is_real()
True
sage: CC(1+i).is_real()
False
```

is_square()

This function always returns true as $\mathbf{C}$ is algebraically closed.

EXAMPLES:

```
sage: a = ComplexNumber(2,1)
sage: a.is_square()
True
```

$\mathbf{C}$ is algebraically closed, hence every element is a square:

```
sage: b = ComplexNumber(5)
sage: b.is_square()
True
```

log()

Complex logarithm of z with branch chosen as follows: Write $z = \rho e^{i\theta} with` - \pi <= \theta < pi$. Then $\log(z) = \log(\rho) + i\theta$.

Warning: Currently the real log is computed using floats, so there is potential precision loss.

EXAMPLES:

```
sage: a = ComplexNumber(2,1)
sage: a.log()
0.804718956217050 + 0.463647609000806*I
sage: log(a.abs())
0.804718956217050
sage: a.argument()
0.463647609000806

sage: b = ComplexNumber(float(exp(42)),0)
sage: b.log()
41.99999999999971
```

multiplicative_order()

Return the multiplicative order of this complex number, if known, or raise a NotImplementedError.

EXAMPLES:

```
sage: C.<i> = ComplexField()
sage: i.multiplicative_order()
4
sage: C(1).multiplicative_order()
1
sage: C(-1).multiplicative_order()
2
sage: C(i^2).multiplicative_order()
2
sage: C(-i).multiplicative_order()
4
sage: C(2).multiplicative_order()
+Infinity
sage: w = (1+sqrt(-3.0))/2; w
0.500000000000000 + 0.866025403784439*I
sage: abs(w)
1.00000000000000
sage: w.multiplicative_order()
...
NotImplementedError: order of element not known
```

norm()

Returns the norm of self.

$norm(a + bi) = a^2 + b^2$

EXAMPLES: This indeed acts as the square function when the imaginary component of self is equal to zero:

```
sage: a = ComplexNumber(2,1)
sage: a.norm()
5.00000000000000
```

```
sage: b = ComplexNumber(4.2,0)
sage: b.norm()
17.6400000000000
sage: b^2
17.6400000000000
```

nth_root()

The n-th root function.

INPUT:

- `all` - bool (default: False); if True, return a list of all n-th roots.

EXAMPLES:

```
sage: a = CC(27)
sage: a.nth_root(3)
3.00000000000000
sage: a.nth_root(3, all=True)
[3.00000000000000, -1.50000000000000 + 2.59807621135332*I, -1.50000000000000 - 2.59807621135
sage: a = ComplexField(20)(2,1)
sage: [r^7 for r in a.nth_root(7, all=True)]
[2.0000 + 1.0000*I, 2.0000 + 1.0000*I, 2.0000 + 1.0000*I, 2.0000 + 1.0000*I, 2.0000 + 1.0000
```

prec()

Return precision of this complex number.

EXAMPLES:

```
sage: i = ComplexField(2000).0
sage: i.prec()
2000
```

real()

Return real part of self.

EXAMPLES:

```
sage: i = ComplexField(100).0
sage: z = 2 + 3*i
sage: x = z.real(); x
2.0000000000000000000000000000
sage: x.parent()
Real Field with 100 bits of precision
```

sec()

EXAMPLES:

```
sage: ComplexField(100)(1,1).sec()
0.49833703055518678521380589177 + 0.59108384172104504805039169297*I
```

sech()

EXAMPLES:

```
sage: ComplexField(100)(1,1).sech()
0.49833703055518678521380589177 - 0.59108384172104504805039169297*I
```

sin()

EXAMPLES:

```
sage: (1+CC(I)).sin()
1.29845758141598 + 0.634963914784736*I
```

sinh()

EXAMPLES:

```
sage: (1+CC(I)).sinh()
0.634963914784736 + 1.29845758141598*I
```

sqrt()

The square root function, taking the branch cut to be the negative real axis.

INPUT:

- `all` - bool (default: False); if True, return a list of all square roots.

EXAMPLES:

```
sage: C.<i> = ComplexField(30)
sage: i.sqrt()
0.70710678 + 0.70710678*I
sage: (1+i).sqrt()
1.0986841 + 0.45508986*I
sage: (C(-1)).sqrt()
1.0000000*I
sage: (1 + 1e-100*i).sqrt()^2
1.0000000 + 1.0000000e-100*I
sage: i = ComplexField(200).0
sage: i.sqrt()
0.70710678118654752440084436210484903928483593768847403658834 + 0.70710678118654752440084436
```

str()

Return a string representation of this number.

INPUTS:

- `base` - The base to use for printing (default 10)
- `truncate` - (default: `True`) Whether to print fewer digits than are available, to mask errors in the last bits.

EXAMPLES:

```
sage: a = CC(pi + I*e)
sage: a.str()
'3.14159265358979 + 2.71828182845905*I'
sage: a.str(truncate=False)
'3.1415926535897931 + 2.7182818284590451*I'
sage: a.str(base=2)
'11.001001000011111101101010100010001000010110100011000 + 10.101101111100001010100010110001
sage: CC(0.5 + 0.625*I).str(base=2)
'0.10000000000000000000000000000000000000000000000000000 + 0.10100000000000000000000000000000
sage: a.str(base=16)
'3.243f6a8885a30 + 2.b7e151628aed2*I'
sage: a.str(base=36)
'3.53i5ab8p5fc + 2.puw5nggjf8f*I'
```

tan()

EXAMPLES:

```
sage: (1+CC(I)).tan()
0.271752585319512 + 1.08392332733869*I
```

tanh()

EXAMPLES:

```
sage: (1+CC(I)).tanh()
1.08392332733869 + 0.271752585319512*I
```

zeta()

Return the Riemann zeta function evaluated at this complex number.

EXAMPLES:

```
sage: i = ComplexField(30).gen()
sage: z = 1 + i
sage: z.zeta()
0.58215806 - 0.92684856*I
sage: zeta(z)
0.58215806 - 0.92684856*I
```

class RRtoCC()

create_ComplexNumber()

Return the complex number defined by the strings s_real and s_imag as an element of ComplexField(prec=n), where n potentially has slightly more (controlled by pad) bits than given by s.

INPUT:

- s_real - a string that defines a real number (or something whose string representation defines a number)
- s_imag - a string that defines a real number (or something whose string representation defines a number)
- pad - an integer >= 0.
- min_prec - number will have at least this many bits of precision, no matter what.

EXAMPLES:

```
sage: ComplexNumber('2.3')
2.30000000000000
sage: ComplexNumber('2.3','1.1')
2.30000000000000 + 1.10000000000000*I
sage: ComplexNumber(10)
10.0000000000000
sage: ComplexNumber(10,10)
10.0000000000000 + 10.0000000000000*I
sage: ComplexNumber(1.00000000000000000000000000,2)
1.00000000000000000000000000 + 2.00000000000000000000000000*I
sage: ComplexNumber(1,2.00000000000000000000)
1.00000000000000000000 + 2.00000000000000000000*I


sage: sage.rings.complex_number.create_ComplexNumber(s_real=2,s_imag=1)
2.00000000000000 + 1.00000000000000*I
```

is_ComplexNumber()

Returns True if x is a complex number. In particular, if x is of the ComplexNumber type.

EXAMPLES:

```
sage: from sage.rings.complex_number import is_ComplexNumber
sage: a = ComplexNumber(1,2); a
1.00000000000000 + 2.00000000000000*I
sage: is_ComplexNumber(a)
True
sage: b = ComplexNumber(1); b
1.00000000000000
sage: is_ComplexNumber(b)
True
```

Note that the global element I is of type SymbolicConstant. However, elements of the class ComplexField_class are of type ComplexNumber:

```
sage: c = 1 + 2*I
sage: is_ComplexNumber(c)
False
sage: d = CC(1 + 2*I)
sage: is_ComplexNumber(d)
True
```

make_ComplexNumber0()

set_global_complex_round_mode()

25.6 Field of Arbitrary Precision Real Intervals

AUTHORS:

- Carl Witty (2007-01-21): based on `real_mpfr.pyx`; changed it to use mpfi rather than mpfr.
- William Stein (2007-01-24): modifications and clean up and docs, etc.

This is a straightforward binding to the MPFI library; it may be useful to refer to its documentation for more details.

An interval is represented as a pair of floating-point numbers a and b (where a <= b) and is printed as a standard floating-point number with a question mark (for instance, 3.1416?). The question mark indicates that the preceding digit may have an error of +/- 1. These floating-point numbers are implemented using MPFR (the same as the RealNumber elements of RealField).

There is also an alternate method of printing, where the interval prints as [a .. b] (for instance, [3.1415 .. 3.1416]).

The interval represents the set { x : a <= x <= b } (so if a == b, then the interval represents that particular floating-point number). The endpoints can include positive and negative infinity, with the obvious meaning. It is also possible to have a NaN (not-a-number) interval, which is represented by having either endpoint be NaN.

PRINTING:

There are two styles for printing intervals: 'brackets' style and 'question' style (the default).

In question style, we print the "known correct" part of the number, followed by a question mark. The question mark indicates that the preceding digit is possibly wrong by +/- 1.

```
sage: RIF(sqrt(2))
1.414213562373095?
```

However, if the interval is precise (its lower bound is equal to its upper bound) and equal to a not-too-large integer, then we just print that integer.

```
sage: RIF(0)
0
sage: RIF(654321)
654321
```

```
sage: RIF(123, 125)
124.?
sage: RIF(123, 126)
1.3?e2
```

As we see in the last example, question style can discard almost a whole digit's worth of precision. We can reduce this by allowing "error digits": an error following the question mark, that gives the maximum error of the digit(s) before the question mark. If the error is absent (which it always is in the default printing), then it is taken to be 1.

```
sage: RIF(123, 126).str(error_digits=1)
'125.?2'
sage: RIF(123, 127).str(error_digits=1)
'125.?2'
sage: v = RIF(-e, pi); v
0.?e1
sage: v.str(error_digits=1)
'1.?4'
sage: v.str(error_digits=5)
'0.2117?29300'
```

Error digits also sometimes let us indicate that the interval is actually equal to a single floating-point number.

```
sage: RIF(54321/256)
212.19140625000000?
sage: RIF(54321/256).str(error_digits=1)
'212.191406250000000?0'
```

In brackets style, intervals are printed with the left value rounded down and the right rounded up, which is conservative, but in some ways unsatisfying.

Consider a 3-bit interval containing exactly the floating-point number 1.25. In round-to-nearest or round-down, this prints as 1.2; in round-up, this prints as 1.3. The straightforward options, then, are to print this interval as [1.2 .. 1.2] (which does not even contain the true value, 1.25), or to print it as [1.2 .. 1.3] (which gives the impression that the upper and lower bounds are not equal, even though they really are). Neither of these is very satisfying, but we have chosen the latter.

```
sage: R = RealIntervalField(3)
sage: a = R(1.25)
sage: a.str(style='brackets')
'[1.2 .. 1.3]'
sage: a == 1.25
True
sage: a == 2
False
```

COMPARISONS:

Comparison operations (==,!=,<,<=,>,>=) return true if every value in the first interval has the given relation to every value in the second interval. The cmp(a, b) function works differently; it compares two intervals lexicographically. (However, the behavior is not specified if given a non-interval and an interval.)

This convention for comparison operators has good and bad points. The good:

- Expected transitivity properties hold (if a > b and b == c, then a > c; etc.)
- if a>b, then cmp(a, b) == 1; if a==b, then cmp(a,b) == 0; if a<b, then cmp(a, b) == -1
- a==0 is true if the interval contains only the floating-point number 0; similarly for a==1
- a>0 means something useful (that every value in the interval is greater than 0)

The bad:

- Trichotomy fails to hold: there are values (a,b) such that none of a<b, a==b, or a>b are true
- It is not the case that if cmp(a, b) == 0 then a==b, or that if cmp(a, b) == 1 then a>b, or that if cmp(a, b) == -1 then a<b
- There are values a,b such that a<=b but neither a<b nor a==b hold

Note that intervals a and b overlap iff not(a != b).

EXAMPLES:

```
sage: 0 < RIF(1, 2)
True
sage: 0 == RIF(0)
True
sage: not(0 == RIF(0, 1))
True
sage: not(0 != RIF(0, 1))
True
sage: 0 <= RIF(0, 1)
True
sage: not(0 < RIF(0, 1))
True
sage: cmp(RIF(0), RIF(0, 1))
-1
sage: cmp(RIF(0, 1), RIF(0))
1
sage: cmp(RIF(0, 1), RIF(1))
-1
sage: cmp(RIF(0, 1), RIF(0, 1))
0
```

RealInterval()

Return the real number defined by the string s as an element of `RealIntervalField(prec=n)`, where n potentially has slightly more (controlled by pad) bits than given by s.

INPUT:

- `s` - a string that defines a real number (or something whose string representation defines a number)
- `upper` - (default: None) - upper endpoint of interval if given, in which case s is the lower endpoint.
- `base` - an integer between 2 and 36
- `pad` - an integer = 0.
- `min_prec` - number will have at least this many bits of precision, no matter what.

EXAMPLES:

```
sage: RealInterval('2.3')
2.300000000000000?
sage: RealInterval(10)
10
sage: RealInterval('1.00000000000000000000000000000000')
1
sage: RealInterval('1.23456789012345678901234567890123450')
1.23456789012345678901234567890123450?
sage: RealInterval(29308290382930840239842390482, 3^20).str(style='brackets')
'[3.48678440100000000000000000000e9 .. 2.93082903829308402398423904820e28]'
```

RealIntervalField()
Construct a RealIntervalField_class, with caching.

INPUT:

- prec - (integer) precision; default = 53: The number of bits used to represent the mantissa of a floating-point number. The precision can be any integer between mpfr_prec_min() and mpfr_prec_max(). In the current implementation, mpfr_prec_min() is equal to 2.
- sci_not - (default: False) whether or not to display using scientific notation

EXAMPLES:

```
sage: RealIntervalField()
Real Interval Field with 53 bits of precision
sage: RealIntervalField(200, sci_not=True)
Real Interval Field with 200 bits of precision
sage: RealIntervalField(53) is RIF
True
sage: RealIntervalField(200) is RIF
False
sage: RealIntervalField(200) is RealIntervalField(200)
True
```

See the documentation for RealIntervalField_class for many more examples.

class **RealIntervalFieldElement**()
A real number interval.

absolute_diameter()
The diameter of this interval (for [a .. b], this is b-a), rounded upward, as a RealNumber.

EXAMPLES:

```
sage: RIF(1, pi).absolute_diameter()
2.14159265358979
```

alea()
RIF(a, b).alea() gives a floating-point number picked at random from the interval.

EXAMPLES:

```
sage: RIF(1, 2).alea() # random
1.34696133696137
```

algdep()
Returns a polynomial of degree at most n which is approximately satisfied by this number. Note that the returned polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n.

Pari needs to know the number of "known good bits" in the number; we automatically get that from the interval width.

ALGORITHM: Uses the PARI C-library algdep command.

EXAMPLE:

```
sage: r = sqrt(RIF(2)); r
1.414213562373095?
sage: r.algdep(5)
x^2 - 2
```

If we compute a wrong, but precise, interval, we get a wrong answer.

```
sage: r = sqrt(RealIntervalField(200)(2)) + (1/2)^40; r
1.41421356237400454350346165244761311763217187537694807331766802
sage: r.algdep(5)
7266488*x^5 + 22441629*x^4 - 90470501*x^3 + 23297703*x^2 + 45778664*x + 13681026
```

But if we compute an interval that includes the number we mean, we're much more likely to get the right answer, even if the interval is very imprecise.

```
sage: r = r.union(sqrt(2.0))
sage: r.algdep(5)
x^2 - 2
```

Even on this extremely imprecise interval we get an answer which is technically correct.

```
sage: RIF(-1, 1).algdep(5)
x
```

arccos()

Returns the inverse cosine of this number

EXAMPLES:

```
sage: q = RIF.pi()/3; q
1.047197551196598?
sage: i = q.cos(); i
0.500000000000000?
sage: q2 = i.arccos(); q2
1.047197551196598?
sage: q == q2
False
sage: q != q2
False
sage: q2.lower() == q.lower()
False
sage: q - q2
0.?e-15
sage: q in q2
True
```

arccosh()

Returns the hyperbolic inverse cosine of this number

EXAMPLES:

```
sage: q = RIF.pi()/2
sage: i = q.arccosh() ; i
1.023227478547551?
```

arccoth()

Returns the inverse hyperbolic cotangent of this number.

EXAMPLES:

```
sage: RealIntervalField(100)(2).arccoth()
0.54930614433405484569762261846?
sage: (2.0).arccoth()
0.549306144334055
```

arccsch()

Returns the inverse hyperbolic cosecant of this number.

EXAMPLES:

```
sage: RealIntervalField(100)(2).arccsch()
0.48121182505960344749775891342?
sage: (2.0).arccsch()
0.481211825059603
```

arcsech()

Returns the inverse hyperbolic secant of this number.

EXAMPLES:

```
sage: RealIntervalField(100)(0.5).arcsech()
1.3169578969248167086250463473?
sage: (0.5).arcsech()
1.31695789692482
```

arcsin()

Returns the inverse sine of this number

EXAMPLES:

```
sage: q = RIF.pi()/5; q
0.6283185307179587?
sage: i = q.sin(); i
0.587785252292474?
sage: q2 = i.arcsin(); q2
0.6283185307179593?
sage: q == q2
False
sage: q != q2
False
sage: q2.lower() == q.lower()
False
sage: q - q2
0.?e-15
sage: q in q2
True
```

arcsinh()

Returns the hyperbolic inverse sine of this number

EXAMPLES:

```
sage: q = RIF.pi()/7
sage: i = q.sinh() ; i
0.4640176304929913?
sage: i.arcsinh() - q
0.?e-15
```

arctan()

Returns the inverse tangent of this number

EXAMPLES:

```
sage: q = RIF.pi()/5; q
0.6283185307179587?
sage: i = q.tan(); i
0.7265425280053613?
sage: q2 = i.arctan(); q2
0.6283185307179593?
sage: q == q2
False
sage: q != q2
False
```

```
sage: q2.lower() == q.lower()
False
sage: q - q2
0.?e-15
sage: q in q2
True
```

arctanh()

Returns the hyperbolic inverse tangent of this number

EXAMPLES:

```
sage: q = RIF.pi()/7
sage: i = q.tanh() ; i
0.420911241048535?
sage: i.arctanh() - q
0.?e-15
```

ceil()

Returns the ceiling of this number

OUTPUT: integer

EXAMPLES:

```
sage: (2.99).ceil()
3
sage: (2.00).ceil()
2
sage: (2.01).ceil()
3
sage: R = RealIntervalField(30)
sage: a = R(-9.5, -11.3); a.str(style='brackets')
'[-11.300000012 .. -9.5000000000]'
sage: a.floor().str(style='brackets')
'[-12.000000000 .. -10.000000000]'
sage: a.ceil()
-10.?
sage: ceil(a).str(style='brackets')
'[-11.000000000 .. -9.0000000000]'
```

ceiling()

center()

RIF(a, b).center() == (a+b)/2

EXAMPLES:

```
sage: RIF(1, 2).center()
1.50000000000000
```

contains_zero()

Returns True if self is an interval containing zero.

EXAMPLES:

```
sage: RIF(0).contains_zero()
True
sage: RIF(1, 2).contains_zero()
False
sage: RIF(-1, 1).contains_zero()
True
sage: RIF(-1, 0).contains_zero()
True
```

cos()
Returns the cosine of this number.
EXAMPLES:

```
sage: t=RIF(pi)/2
sage: t.cos()
0.?e-15
sage: t.cos().str(style='brackets')
'[-1.6081226496766367e-16 .. 6.1232339957367661e-17]'
sage: t.cos().cos()
0.9999999999999999?
```

TESTS:
This looped forever with an earlier version of MPFI, but now it works.

```
sage: RIF(-1, 1).cos().str(style='brackets')
'[0.54030230586813965 .. 1.0000000000000000]'
```

cosh()
Returns the hyperbolic cosine of this number
EXAMPLES:

```
sage: q = RIF.pi()/12
sage: q.cosh()
1.034465640095511?
```

cot()
Returns the cotangent of this number.
EXAMPLES:

```
sage: RealIntervalField(100)(2).cot()
-0.45765755436028576375027741043 2?
```

coth()
Returns the hyperbolic cotangent of this number.
EXAMPLES:

```
sage: RealIntervalField(100)(2).coth()
1.0373147207275480958778097647 7?
```

csc()
Returns the cosecant of this number.
EXAMPLES:

```
sage: RealIntervalField(100)(2).csc()
1.0997501702946164667566973970 26?
```

csch()
Returns the hyperbolic cosecant of this number.
EXAMPLES:

```
sage: RealIntervalField(100)(2).csch()
0.27572056477178320775835148216 3?
```

diameter()
If (0 in self), returns self.absolute_diameter(), otherwise self.relative_diameter().
EXAMPLES:

```
sage: RIF(1, 2).diameter()
0.666666666666667
```

```
sage: RIF(1, 2).absolute_diameter()
1.00000000000000
sage: RIF(1, 2).relative_diameter()
0.666666666666667
sage: RIF(pi).diameter()
1.41357985842823e-16
sage: RIF(pi).absolute_diameter()
4.44089209850063e-16
sage: RIF(pi).relative_diameter()
1.41357985842823e-16
sage: (RIF(pi) - RIF(3, 22/7)).diameter()
0.142857142857144
sage: (RIF(pi) - RIF(3, 22/7)).absolute_diameter()
0.142857142857144
sage: (RIF(pi) - RIF(3, 22/7)).relative_diameter()
2.03604377705518
```

exp()

Returns $e^{\texttt{self}}$

EXAMPLES:

```
sage: r = RIF(0.0)
sage: r.exp()
1

sage: r = RIF(32.3)
sage: a = r.exp(); a
1.065888472748645?e14
sage: a.log()
32.300000000000000?

sage: r = RIF(-32.3)
sage: r.exp()
9.38184458849869?e-15
```

exp2()

Returns $2^{\texttt{self}}$

EXAMPLES:

```
sage: r = RIF(0.0)
sage: r.exp2()
1

sage: r = RIF(32.0)
sage: r.exp2()
4294967296

sage: r = RIF(-32.3)
sage: r.exp2()
1.891172482530207?e-10
```

floor()

Returns the floor of this number

EXAMPLES:

```
sage: R = RealIntervalField()
sage: (2.99).floor()
2
sage: (2.00).floor()
```

```
2
sage: floor(RR(-5/2))
-3
sage: R = RealIntervalField(100)
sage: a = R(9.5, 11.3); a.str(style='brackets')
'[9.5000000000000000000000000000000 .. 11.300000000000000710542735760101]'
sage: floor(a).str(style='brackets')
'[9.0000000000000000000000000000000 .. 11.000000000000000000000000000000]'
sage: a.floor()
10.?
sage: ceil(a)
11.?
sage: a.ceil().str(style='brackets')
'[10.000000000000000000000000000000 .. 12.000000000000000000000000000000]'
```

fp_rank_diameter()

Computes the diameter of this interval in terms of the "floating-point rank". That is, it gives the number of floating-point numbers (of the current precision) contained in the given interval, minus one. An `fp_rank_diameter` of 0 means that the interval is exact; an `fp_rank_diameter` of 1 means that the interval is as tight as possible, unless the number you're trying to represent is actually exactly representable as a floating-point number.

EXAMPLES:

```
sage: RIF(pi).fp_rank_diameter()
1
sage: RIF(12345).fp_rank_diameter()
0
sage: RIF(-sqrt(2)).fp_rank_diameter()
1
sage: RIF(5/8).fp_rank_diameter()
0
sage: RIF(5/7).fp_rank_diameter()
1
sage: a = RIF(pi)^12345; a
2.06622879260?e6137
sage: a.fp_rank_diameter()
30524
sage: (RIF(sqrt(2)) - RIF(sqrt(2))).fp_rank_diameter()
96714060885426721511178 26
```

Just because we have the best possible interval, doesn't mean the interval is actually small:

```
sage: a = RIF(pi)^1234567890; a
[2.0985787164673874e323228496 .. +infinity]
sage: a.fp_rank_diameter()
1
```

intersection()

Return the intersection of two intervals. If the intervals do not overlap, raises a ValueError.

EXAMPLES:

```
sage: RIF(1, 2).intersection(RIF(1.5, 3)).str(style='brackets')
'[1.5000000000000000 .. 2.0000000000000000]'
sage: RIF(1, 2).intersection(RIF(4/3, 5/3)).str(style='brackets')
'[1.3333333333333332 .. 1.6666666666666668]'
sage: RIF(1, 2).intersection(RIF(3, 4))
...
ValueError: intersection of non-overlapping intervals
```

is_NaN()

is_exact()

is_int()

OUTPUT:

- bool - True or False
- n - an integer

Checks to see whether this interval includes exactly one integer. If so, returns a tuple (True, n), where n is that integer; otherwise, returns (False, None).

EXAMPLES:

```
sage: a = RIF(0.8,1.5)
sage: a.is_int()
(True, 1)
sage: a = RIF(1.1,1.5)
sage: a.is_int()
(False, None)
sage: a = RIF(1,2)
sage: a.is_int()
(False, None)
sage: a = RIF(-1.1, -0.9)
sage: a.is_int()
(True, -1)
sage: a = RIF(0.1, 1.9)
sage: a.is_int()
(True, 1)
```

log()

EXAMPLES:

```
sage: R = RealIntervalField()
sage: R(2).log()
0.6931471805599453?
```

log10()

Returns log to the base 10 of self

EXAMPLES:

```
sage: r = RIF(16.0); r.log10()
1.204119982655925?
sage: r.log() / log(10.0)
1.204119982655925?

sage: r = RIF(39.9); r.log10()
1.600972895686749?

sage: r = RIF(0.0)
sage: r.log10()
[-infinity .. -infinity]

sage: r = RIF(-1.0)
sage: r.log10()
[.. NaN ..]
```

log2()

Returns log to the base 2 of self

EXAMPLES:

```
sage: r = RIF(16.0)
sage: r.log2()
4

sage: r = RIF(31.9); r.log2()
4.995484518877507?

sage: r = RIF(0.0, 2.0)
sage: r.log2()
[-infinity .. 1.0000000000000000]
```

lower()

Returns the lower bound of this interval

rnd - (string) the rounding mode

- RNDN - round to nearest
- RNDD - (default) round towards minus infinity
- RNDZ - round towards zero
- RNDU - round towards plus infinity

The rounding mode does not affect the value returned as a floating-point number, but it does control which variety of RealField the returned number is in, which affects printing and subsequent operations.

EXAMPLES:

```
sage: R = RealIntervalField(13)
sage: R.pi().lower().str(truncate=False)
'3.1411'

sage: x = R(1.2,1.3); x.str(style='brackets')
'[1.1999 .. 1.3001]'
sage: x.lower()
1.19
sage: x.lower('RNDU')
1.20
sage: x.lower('RNDN')
1.20
sage: x.lower('RNDZ')
1.19
sage: x.lower().parent()
Real Field with 13 bits of precision and rounding RNDD
sage: x.lower('RNDU').parent()
Real Field with 13 bits of precision and rounding RNDU
sage: x.lower() == x.lower('RNDU')
True
```

magnitude()

The largest absolute value of the elements of the interval.

EXAMPLES:

```
sage: RIF(-2, 1).magnitude()
2.00000000000000
sage: RIF(-1, 2).magnitude()
2.00000000000000
```

mignitude()

The smallest absolute value of the elements of the interval.

EXAMPLES:

```
sage: RIF(-2, 1).mignitude()
0.000000000000000
sage: RIF(-2, -1).mignitude()
1.00000000000000
sage: RIF(3, 4).mignitude()
3.00000000000000
```

multiplicative_order()

overlaps()

Returns true if self and other are intervals with at least one value in common. For intervals a and b, we have a.overlaps(b) iff not(a!=b).

EXAMPLES:

```
sage: RIF(0, 1).overlaps(RIF(1, 2))
True
sage: RIF(1, 2).overlaps(RIF(0, 1))
True
sage: RIF(0, 1).overlaps(RIF(2, 3))
False
sage: RIF(2, 3).overlaps(RIF(0, 1))
False
sage: RIF(0, 3).overlaps(RIF(1, 2))
True
sage: RIF(0, 2).overlaps(RIF(1, 3))
True
```

parent()

EXAMPLES:

```
sage: R = RealIntervalField()
sage: a = R('1.2456')
sage: a.parent()
Real Interval Field with 53 bits of precision
```

prec()

real()

Return the real part of self.

(Since self is a real number, this simply returns self.)

relative_diameter()

The relative diameter of this interval (for [a .. b], this is (b-a)/((a+b)/2)), rounded upward, as a RealNumber.

EXAMPLES:

```
sage: RIF(1, pi).relative_diameter()
1.03418797197910
```

sec()

Returns the secant of this number.

EXAMPLES:

```
sage: RealIntervalField(100)(2).sec()
-2.4029979617223809897546004014?
```

sech()

Returns the hyperbolic secant of this number.

EXAMPLES:

```
sage: RealIntervalField(100)(2).sech()
0.26580222883407969212086273982?
```

simplest_rational()

Returns the simplest rational in this interval. Given rationals a/b and c/d (both in lowest terms), the former is simpler if b<d or if b==d and $|a| < |c|$.

If optional parameters low_open or high_open are true, then treat this as an open interval on that end.

EXAMPLES:

```
sage: RealIntervalField(10)(pi).simplest_rational()
22/7
sage: RealIntervalField(20)(pi).simplest_rational()
355/113
sage: RIF(0.123, 0.567).simplest_rational()
1/2
sage: RIF(RR(1/3).nextabove(), RR(3/7)).simplest_rational()
2/5
sage: RIF(1234/567).simplest_rational()
1234/567
sage: RIF(-8765/432).simplest_rational()
-8765/432
sage: RIF(-1.234, 0.003).simplest_rational()
0
sage: RIF(RR(1/3)).simplest_rational()
6004799503160661/18014398509481984
sage: RIF(RR(1/3)).simplest_rational(high_open=True)
...
ValueError: simplest_rational() on open, empty interval
sage: RIF(1/3, 1/2).simplest_rational()
1/2
sage: RIF(1/3, 1/2).simplest_rational(high_open=True)
1/3
sage: phi = ((RealIntervalField(500)(5).sqrt() + 1)/2)
sage: phi.simplest_rational() == fibonacci(362)/fibonacci(361)
True
```

sin()

Returns the sine of this number

EXAMPLES:

```
sage: R = RealIntervalField(100)
sage: R(2).sin()
0.90929742682568169539601986591?
```

sinh()

Returns the hyperbolic sine of this number

EXAMPLES:

```
sage: q = RIF.pi()/12
sage: q.sinh()
0.2648002276022707?
```

sqrt()

Return a square root of self. Raises an error if self is nonpositive.

If you use self.square_root() then an interval will always be returned (though it will be NaN if self is nonpositive).

EXAMPLES:

```
sage: r = RIF(4.0)
sage: r.sqrt()
2
sage: r.sqrt()^2 == r
True

sage: r = RIF(4344)
sage: r.sqrt()
65.909028213136337
sage: r.sqrt()^2 == r
False
sage: r in r.sqrt()^2
True
sage: r.sqrt()^2 - r
0.?e-11
sage: (r.sqrt()^2 - r).str(style='brackets')
'[-9.0949470177292824e-13 .. 1.8189894035458565e-12]'

sage: r = RIF(-2.0)
sage: r.sqrt()
...
ValueError: self (=-2) is not >= 0

sage: r = RIF(-2, 2)
sage: r.sqrt()
...
ValueError: self (=0.?e1) is not >= 0
```

square()

Return the square of the interval. Note that squaring an interval is different than multiplying it by itself, because the square can never be negative.

EXAMPLES:

```
sage: RIF(1, 2).square().str(style='brackets')
'[1.0000000000000000 .. 4.0000000000000000]'
sage: RIF(-1, 1).square().str(style='brackets')
'[0.00000000000000000 .. 1.0000000000000000]'
sage: (RIF(-1, 1) * RIF(-1, 1)).str(style='brackets')
'[-1.0000000000000000 .. 1.0000000000000000]'
```

square_root()

Return a square root of self. An interval will always be returned (though it will be NaN if self is nonpositive).

EXAMPLES:

```
sage: r = RIF(-2.0)
sage: r.square_root()
[.. NaN ..]
sage: r.sqrt()
...
ValueError: self (=-2) is not >= 0
```

str()

INPUT:

- `base` - base for output
- `style` - The printing style; either 'brackets' or 'question' (or None, to use the current default).
- `no_sci` - if True do not print using scientific notation; if False print with scientific notation; if None (the default), print how the parent prints.

- `e` - symbol used in scientific notation
- `error_digits` - The number of digits of error to print, in 'question' style.

We support two different styles of printing; 'question' style and 'brackets' style. In question style (the default), we print the "known correct" part of the number, followed by a question mark:

```
sage: RIF(pi).str()
'3.1415926535897934?'
sage: RIF(pi, 22/7).str()
'3.142?'
sage: RIF(pi, 22/7).str(style='question')
'3.142?'
```

However, if the interval is precisely equal to some integer that's not too large, we just return that integer.

```
sage: RIF(-42).str()
'-42'
sage: RIF(0).str()
'0'
sage: RIF(12^5).str(base=3)
'110122100000'
```

Very large integers, however, revert to the normal question-style printing.

```
sage: RIF(3^7).str()
'2187'
sage: RIF(3^7 * 2^256).str()
'2.5323729916201052?e80'
```

In brackets style, we print the lower and upper bounds of the interval within brackets:

```
sage: RIF(237/16).str(style='brackets')
'[14.812500000000000 .. 14.812500000000000]'
```

Note that the lower bound is rounded down, and the upper bound is rounded up. So even if the lower and upper bounds are equal, they may print differently. (This is done so that the printed representation of the interval contains all the numbers in the internal binary interval.)

For instance, we find the best 10-bit floating point representation of 1/3:

```
sage: RR10 = RealField(10)
sage: RR(RR10(1/3))
0.333496093750000
```

And we see that the point interval containing only this floating-point number prints as a wider decimal interval, that does contain the number:

```
sage: RIF10 = RealIntervalField(10)
sage: RIF10(RR10(1/3)).str(style='brackets')
'[0.33349 .. 0.33350]'
```

We always use brackets style for NaN and infinities.

```
sage: RIF(pi, infinity)
[3.1415926535897931 .. +infinity]
sage: RIF(NaN)
[.. NaN ..]
```

Let's take a closer, formal look at the question style. In its full generality, a number printed in the question style looks like:

MANTISSA ?ERROR eEXPONENT

(without the spaces). The "eEXPONENT" part is optional; if it is missing, then the exponent is 0. (If the base is greater than 10, then the exponent separator is "@" instead of "e".)

The "ERROR" is optional; if it is missing, then the error is 1.

The mantissa is printed in base b, and always contains a decimal point (also known as a radix point, in bases other than 10). (The error and exponent are always printed in base 10.)

We define the "precision" of a floating-point printed representation to be the positional value of the last digit of the mantissa. For instance, in 2.7?e5, the precision is 10^4; in 8.?, the precision is 10^0; and in 9.35? the precision is 10^{-2}. This precision will always be 10^k for some k (or, for an arbitrary base b, b^k).

Then the interval is contained in the interval:

$$\text{mantissa} * b^{\text{exponent}} - \text{error} * b^k .. \text{mantissa} * b^{\text{exponent}} + \text{error} * b^k$$

To control the printing, we can specify a maximum number of error digits. The default is 0, which means that we do not print an error at all (so that the error is always the default, 1).

Now, consider the precisions needed to represent the endpoints (this is the precision that would be produced by `v.lower().str(no_sci=False, truncate=False)`). Our result is no more precise than the less precise endpoint, and is sufficiently imprecise that the error can be represented with the given number of decimal digits. Our result is the most precise possible result, given these restrictions. When there are two possible results of equal precision and with the same error width, then we pick the one which is farther from zero. (For instance, RIF(0, 123) with two error digits could print as 61.?62 or 62.?62. We prefer the latter because it makes it clear that the interval is known not to be negative.)

EXAMPLES:

```
sage: a = RIF(59/27); a
2.185185185185186?
sage: a.str()
'2.185185185185186?'
sage: a.str(style='brackets')
'[2.1851851851851851 .. 2.1851851851851856]'
sage: a.str(16)
'2.2f684bda12f69?'
sage: a.str(no_sci=False)
'2.185185185185186?e0'
sage: pi_appr = RIF(pi, 22/7)
sage: pi_appr.str(style='brackets')
'[3.1415926535897931 .. 3.1428571428571433]'
sage: pi_appr.str()
'3.142?'
sage: pi_appr.str(error_digits=1)
'3.1422?7'
sage: pi_appr.str(error_digits=2)
'3.14223?64'
sage: pi_appr.str(base=36)
'3.6?'
sage: RIF(NaN)
[.. NaN ..]
sage: RIF(pi, infinity)
[3.1415926535897931 .. +infinity]
sage: RIF(-infinity, pi)
[-infinity .. 3.1415926535897936]
sage: RealIntervalField(210)(3).sqrt()
1.732050807568877293527446341505872366942805253810380628055806980?
sage: RealIntervalField(210)(RIF(3).sqrt())
1.732050807568878?
sage: RIF(3).sqrt()
1.732050807568878?
sage: RIF(0, 3^-150)
1.?e-71
```

tan()

Returns the tangent of this number

EXAMPLES:

```
sage: q = RIF.pi()/3
sage: q.tan()
1.732050807568877?
sage: q = RIF.pi()/6
sage: q.tan()
0.577350269189626?
```

tanh()

Returns the hyperbolic tangent of this number

EXAMPLES:

```
sage: q = RIF.pi()/11
sage: q.tanh()
0.2780794292958503?
```

union()

Return the union of two intervals, or of an interval and a real number (more precisely, the convex hull).

EXAMPLES:

```
sage: RIF(1, 2).union(RIF(pi, 22/7)).str(style='brackets')
'[1.0000000000000000 .. 3.1428571428571433]'
sage: RIF(1, 2).union(pi).str(style='brackets')
'[1.0000000000000000 .. 3.1415926535897936]'
sage: RIF(1).union(RIF(0, 2)).str(style='brackets')
'[0.00000000000000000 .. 2.0000000000000000]'
sage: RIF(1).union(RIF(-1)).str(style='brackets')
'[-1.0000000000000000 .. 1.0000000000000000]'
```

upper()

Returns the upper bound of this interval

- RNDN - round to nearest
- RNDD - round towards minus infinity
- RNDZ - round towards zero
- RNDU - (default) round towards plus infinity

The rounding mode does not affect the value returned as a floating-point number, but it does control which variety of RealField the returned number is in, which affects printing and subsequent operations.

EXAMPLES:

```
sage: R = RealIntervalField(13)
sage: R.pi().upper().str(truncate=False)
'3.1417'

sage: R = RealIntervalField(13)
sage: x = R(1.2,1.3); x.str(style='brackets')
'[1.1999 .. 1.3001]'
sage: x.upper()
1.31
sage: x.upper('RNDU')
1.31
sage: x.upper('RNDN')
1.30
sage: x.upper('RNDD')
1.30
sage: x.upper('RNDZ')
1.30
sage: x.upper().parent()
```

```
Real Field with 13 bits of precision and rounding RNDU
sage: x.upper('RNDD').parent()
Real Field with 13 bits of precision and rounding RNDD
sage: x.upper() == x.upper('RNDD')
True
```

class RealIntervalField_class()

RealIntervalField(prec, sci_not, rnd):

INPUT:

- `prec` - (integer) precision; default = 53 prec is the number of bits used to represent the mantissa of a floating-point number. The precision can be any integer between mpfr_prec_min() and mpfr_prec_max(). In the current implementation, mpfr_prec_min() is equal to 2.
- `sci_not` - (default: False) whether or not to display using scientific notation

EXAMPLES:

```
sage: RealIntervalField(10)
Real Interval Field with 10 bits of precision
sage: RealIntervalField()
Real Interval Field with 53 bits of precision
sage: RealIntervalField(100000)
Real Interval Field with 100000 bits of precision
```

Note: The default precision is 53, since according to the GMP manual: 'mpfr should be able to exactly reproduce all computations with double-precision machine floating-point numbers (double type in C), except the default exponent range is much wider and subnormal numbers are not implemented.'

EXAMPLES: Creation of elements

First with default precision. First we coerce elements of various types, then we coerce intervals.

```
sage: RIF = RealIntervalField(); RIF
Real Interval Field with 53 bits of precision
sage: RIF(3)
3
sage: RIF(RIF(3))
3
sage: RIF(pi)
3.141592653589794?
sage: RIF(RealField(53)('1.5'))
1.5000000000000000?
sage: RIF(-2/19)
-0.1052631578947369?
sage: RIF(-3939)
-3939
sage: RIF(-3939r)
-3939
sage: RIF('1.5')
1.5000000000000000?
sage: R200 = RealField(200)
sage: RIF(R200.pi())
3.141592653589794?
```

The base must be explicitly specified as a named parameter:

```
sage: RIF('101101', base=2)
45
sage: RIF('+infinity')
[+infinity .. +infinity]
sage: RIF('[1..3]').str(style='brackets')
'[1.0000000000000000 .. 3.0000000000000000]'
```

Next we coerce some 2-tuples, which define intervals.

```
sage: RIF((-1.5, -1.3))
-1.4?
sage: RIF((RDF('-1.5'), RDF('-1.3')))
-1.4?
sage: RIF((1/3,2/3)).str(style='brackets')
'[0.33333333333333331 .. 0.66666666666666675]'
```

The extra paranthesis aren't needed.

```
sage: RIF(1/3,2/3).str(style='brackets')
'[0.33333333333333331 .. 0.66666666666666675]'
sage: RIF((1,2)).str(style='brackets')
'[1.0000000000000000 .. 2.0000000000000000]'
sage: RIF((1r,2r)).str(style='brackets')
'[1.0000000000000000 .. 2.0000000000000000]'
sage: RIF((pi, e)).str(style='brackets')
'[2.7182818284590455 .. 3.1415926535897932]'
```

Values which can be represented as an exact floating-point number (of the precision of this RealIntervalField) result in a precise interval, where the lower bound is equal to the upper bound (even if they print differently). Other values typically result in an interval where the lower and upper bounds are adjacent floating-point numbers.

```
sage: def check(x):
...       return (x, x.lower() == x.upper())
sage: check(RIF(pi))
(3.1415926535897932?, False)
sage: check(RIF(RR(pi)))
(3.1415926535897932?, True)
sage: check(RIF(1.5))
(1.5000000000000000?, True)
sage: check(RIF('1.5'))
(1.5000000000000000?, True)
sage: check(RIF(0.1))
(0.10000000000000001?, True)
sage: check(RIF(1/10))
(0.10000000000000000?, False)
sage: check(RIF('0.1'))
(0.10000000000000000?, False)
```

Similarly, when specifying both ends of an interval, the lower end is rounded down and the upper end is rounded up.

```
sage: outward = RIF(1/10, 7/10); outward.str(style='brackets')
'[0.09999999999999991 .. 0.70000000000000007]'
sage: nearest = RIF(RR(1/10), RR(7/10)); nearest.str(style='brackets')
'[0.10000000000000000 .. 0.69999999999999996]'
sage: nearest.lower() - outward.lower()
1.38777878078144e-17
```

```
sage: outward.upper() - nearest.upper()
1.11022302462516e-16
```

Some examples with a real interval field of higher precision:

```
sage: R = RealIntervalField(100)
sage: R(3)
3
sage: R(R(3))
3
sage: R(pi)
3.1415926535897932384626433832?
sage: R(-2/19)
-0.10526315789473684210526315789?
sage: R(e,pi).str(style='brackets')
'[2.71828182845904523536028747135?? .. 3.1415926535897932384626433832825]'
```

TESTS:

```
sage: RIF._lower_field() is RealField(53, rnd='RNDD')
True
sage: RIF._upper_field() is RealField(53, rnd='RNDU')
True
sage: RIF._middle_field() is RR
True
```

characteristic()

Returns 0, since the field of real numbers has characteristic 0.

EXAMPLES:

```
sage: RealIntervalField(10).characteristic()
0
```

complex_field()

Return complex field of the same precision.

EXAMPLES:

```
sage: RIF.complex_field()
Complex Interval Field with 53 bits of precision
```

construction()

Returns the functorial construction of self, namely, completion of the rational numbers with respect to the prime at ∞, and the note that this is an interval field.

Also preserves other information that makes this field unique (e.g. precision, print mode).

EXAMPLES:

```
sage: R = RealIntervalField(123)
sage: c, S = R.construction(); S
Rational Field
sage: R == c(S)
True
```

euler_constant()

Returns Euler's gamma constant to the precision of this field.

EXAMPLES:

```
sage: RealIntervalField(100).euler_constant()
0.57721566490153286060651209008?
```

gen()

gens()

is_atomic_repr()

Returns True, to signify that elements of this field print without sums, so parenthesis aren't required, e.g., in coefficients of polynomials.

EXAMPLES:

```
sage: RealIntervalField(10).is_atomic_repr()
True
```

is_exact()

is_finite()

Returns False, since the field of real numbers is not finite.

EXAMPLES:

```
sage: RealIntervalField(10).is_finite()
False
```

log2()

Returns log(2) to the precision of this field.

EXAMPLES:

```
sage: R=RealIntervalField(100)
sage: R.log2()
0.693147180559945309417232121458?
sage: R(2).log()
0.693147180559945309417232121458?
```

name()

ngens()

pi()

Returns pi to the precision of this field.

EXAMPLES:

```
sage: R = RealIntervalField(100)
sage: R.pi()
3.1415926535897932384626433832?
sage: R.pi().sqrt()/2
0.88622692545275801364908374167?
sage: R = RealIntervalField(150)
sage: R.pi().sqrt()/2
0.886226925452758013649083741670572591398774728?
```

prec()

Returns the precision of this field (in bits).

EXAMPLES:

```
sage: RIF.prec()
53
sage: RealIntervalField(200).prec()
200
```

precision()

random_element()

Returns a random element of self. Any arguments are passed onto the random element function in real field.

By default, this is uniformly distributed in $[-1, 1]$.

EXAMPLES:

```
sage: RIF.random_element()
-0.30607732607725314?
sage: RIF.random_element()
-0.075929193054320221?
sage: RIF.random_element(-100, 100)
-83.808125490023344?
```

scientific_notation()

Set or return the scientific notation printing flag. If this flag is True then real numbers with this space as parent print using scientific notation.

INPUT:

- `status` - (bool -) optional flag

to_prec()

Returns a real interval field to the given precision.

EXAMPLES:

```
sage: RIF.to_prec(200)
Real Interval Field with 200 bits of precision
sage: RIF.to_prec(20)
Real Interval Field with 20 bits of precision
sage: RIF.to_prec(53) is RIF
True
```

zeta()

Return an n-th root of unity in the real field, if one exists, or raise a ValueError otherwise.

EXAMPLES:

```
sage: R = RealIntervalField()
sage: R.zeta()
-1
sage: R.zeta(1)
1
sage: R.zeta(5)
...
ValueError: No 5th root of unity in self
```

is_RealIntervalField()

is_RealIntervalFieldElement()

CHAPTER

TWENTYSIX

ALGEBRAIC NUMBER FIELDS

26.1 Number Fields

AUTHORS:

- William Stein (2004, 2005): initial version
- Steven Sivek (2006-05-12): added support for relative extensions
- William Stein (2007-09-04): major rewrite and documentation
- Robert Bradshaw (2008-10): specified embeddings into ambient fields

Note: Unlike in PARI/GP, class group computations *in Sage* do *not* by default assume the Generalized Riemann Hypothesis. To do class groups computations not provably correctly you must often pass the flag proof=False to functions or call the function `proof.number_field(False)`. It can easily take 1000's of times longer to do computations with `proof=True` (the default).

This example follows one in the Magma reference manual:

```
sage: K.<y> = NumberField(x^4 - 420*x^2 + 40000)
sage: z = y^5/11; z
420/11*y^3 - 40000/11*y
sage: R.<y> = PolynomialRing(K)
sage: f = y^2 + y + 1
sage: L.<a> = K.extension(f); L
Number Field in a with defining polynomial y^2 + y + 1 over its base field
sage: KL.<b> = NumberField([x^4 - 420*x^2 + 40000, x^2 + x + 1]); KL
Number Field in b0 with defining polynomial x^4 - 420*x^2 + 40000 over its base field
```

We do some arithmetic in a tower of relative number fields:

```
sage: K.<cuberoot2> = NumberField(x^3 - 2)
sage: L.<cuberoot3> = K.extension(x^3 - 3)
sage: S.<sqrt2> = L.extension(x^2 - 2)
sage: S
Number Field in sqrt2 with defining polynomial x^2 - 2 over its base field
sage: sqrt2 * cuberoot3
cuberoot3*sqrt2
sage: (sqrt2 + cuberoot3)^5
(20*cuberoot3^2 + 15*cuberoot3 + 4)*sqrt2 + 3*cuberoot3^2 + 20*cuberoot3 + 60
sage: cuberoot2 + cuberoot3
cuberoot3 + cuberoot2
```

```
sage: cuberoot2 + cuberoot3 + sqrt2
sqrt2 + cuberoot3 + cuberoot2
sage: (cuberoot2 + cuberoot3 + sqrt2)^2
(2*cuberoot3 + 2*cuberoot2)*sqrt2 + cuberoot3^2 + 2*cuberoot2*cuberoot3 + cuberoot2^2 + 2
sage: cuberoot2 + sqrt2
sqrt2 + cuberoot2
sage: a = S(cuberoot2); a
cuberoot2
sage: a.parent()
Number Field in sqrt2 with defining polynomial x^2 - 2 over its base field
```

Warning: Doing arithmetic in towers of relative fields that depends on canonical coercions is currently VERY SLOW. It is much better to explicitly coerce all elements into a common field, then do arithmetic with them there (which is quite fast).

TESTS:

```
sage: y = polygen(QQ,'y'); K.<beta> = NumberField([y^3 - 3, y^2 - 2])
sage: K(y^10)
27*beta0
sage: beta^10
27*beta0
```

CyclotomicField(*n, names=None, embedding=True*)

Return the n-th cyclotomic field, where n is a positive integer.

INPUT:

- `n` - a positive integer
- `names` - name of generator (optional - defaults to zetan).
- `embedding` - bool or n-th root of unity in an ambient field (default True)

EXAMPLES: We create the 7th cyclotomic field $\mathbf{Q}(\zeta_7)$ with the default generator name.

```
sage: k = CyclotomicField(7); k
Cyclotomic Field of order 7 and degree 6
sage: k.gen()
zeta7
```

The default embedding sends the generator to the complex primative n^{th} root of unitiy of least argument.

```
sage: CC(k.gen())
0.623489801858734 + 0.781831482468030*I
```

Cyclotomic fields are of a special type.

```
sage: type(k)
<class 'sage.rings.number_field.number_field.NumberField_cyclotomic'>
```

We can specify a different generator name as follows.

```
sage: k.<z7> = CyclotomicField(7); k
Cyclotomic Field of order 7 and degree 6
sage: k.gen()
z7
```

The n must be an integer.

```
sage: CyclotomicField(3/2)
...
TypeError: no conversion of this rational to integer
```

The degree must be positive.

```
sage: CyclotomicField(0)
...
ValueError: n (=0) must be a positive integer
```

The special case $n = 1$ does *not* return the rational numbers:

```
sage: CyclotomicField(1)
Cyclotomic Field of order 1 and degree 1
```

Due to their default embedding into $\mathbf{C}$, cyclotomic number fields are all compatable.

```
sage: cf30 = CyclotomicField(30)
sage: cf5 = CyclotomicField(5)
sage: cf3 = CyclotomicField(3)
sage: cf30.gen() + cf5.gen() + cf3.gen()
zeta30^6 + zeta30^5 + zeta30 - 1
sage: cf6 = CyclotomicField(6) ; z6 = cf6.0
sage: cf3 = CyclotomicField(3) ; z3 = cf3.0
sage: cf3(z6)
zeta3 + 1
sage: cf6(z3)
zeta6 - 1
sage: cf9 = CyclotomicField(9) ; z9 = cf9.0
sage: cf18 = CyclotomicField(18) ; z18 = cf18.0
sage: cf18(z9)
zeta18^2
sage: cf9(z18)
-zeta9^5
sage: cf18(z3)
zeta18^3 - 1
sage: cf18(z6)
zeta18^3
sage: cf18(z6)**2
zeta18^3 - 1
sage: cf9(z3)
zeta9^3
```

NumberField(*polynomial, name=None, check=True, names=None, cache=True, embedding=None*)

Return *the* number field defined by the given irreducible polynomial and with variable with the given name. If check is True (the default), also verify that the defining polynomial is irreducible and over Q.

INPUT:

- `polynomial` - a polynomial over QQ or a number field, or a list of polynomials.
- `name` - a string (default: 'a'), the name of the generator
- `check` - bool (default: True); do type checking and irreducibility checking.
- `embedding` - image of the generator in an ambient field (default: None)

EXAMPLES:

```
sage: z = QQ['z'].0
sage: K = NumberField(z^2 - 2,'s'); K
Number Field in s with defining polynomial z^2 - 2
sage: s = K.0; s
s
sage: s*s
2
sage: s^2
2
```

Constructing a relative number field

```
sage: K.<a> = NumberField(x^2 - 2)
sage: R.<t> = K[]
sage: L.<b> = K.extension(t^3+t+a); L
Number Field in b with defining polynomial t^3 + t + a over its base field
sage: L.absolute_field('c')
Number Field in c with defining polynomial x^6 + 2*x^4 + x^2 - 2
sage: a*b
a*b
sage: L(a)
a
sage: L.lift_to_base(b^3 + b)
-a
```

Constructing another number field:

```
sage: k.<i> = NumberField(x^2 + 1)
sage: R.<z> = k[]
sage: m.<j> = NumberField(z^3 + i*z + 3)
sage: m
Number Field in j with defining polynomial z^3 + i*z + 3 over its base field
```

Number fields are globally unique.

```
sage: K.<a>= NumberField(x^3-5)
sage: a^3
5
sage: L.<a>= NumberField(x^3-5)
sage: K is L
True
```

Having different defining polynomials makes the fields different:

```
sage: x = polygen(QQ, 'x'); y = polygen(QQ, 'y')
sage: k.<a> = NumberField(x^2 + 3)
sage: m.<a> = NumberField(y^2 + 3)
sage: k
Number Field in a with defining polynomial x^2 + 3
sage: m
Number Field in a with defining polynomial y^2 + 3
```

One can also define number fields with specified embeddings, may be used for arithmetic and deduce relations with other number fields which would not be valid for an abstract number field:

```
sage: K.<a> = NumberField(x^3-2, embedding=1.2)
sage: RR.coerce_map_from(K)
Composite map:
```

```
  From: Number Field in a with defining polynomial x^3 - 2
  To:   Real Field with 53 bits of precision
  Defn:   Generic morphism:
          From: Number Field in a with defining polynomial x^3 - 2
          To:   Real Lazy Field
          Defn: a -> 1.259921049894873?
        then
          Conversion via _mpfr_ method map:
          From: Real Lazy Field
          To:   Real Field with 53 bits of precision
sage: RR(a)
1.25992104989487
sage: 1.1 + a
2.35992104989487
sage: b = 1/(a+1); b
1/3*a^2 - 1/3*a + 1/3
sage: RR(b)
0.442493334024442
sage: L.<b> = NumberField(x^6-2, embedding=1.1)
sage: L(a)
b^2
sage: a + b
b^2 + b
```

Note that the image only needs to be specified to enough precision to distinguish roots, and is exactly computed to any needed precision:

```
sage: RealField(200)(a)
1.2599210498948731647672106072782283505702514647015079800820
```

One can embed into any other field:

```
sage: K.<a> = NumberField(x^3-2, embedding=CC.gen()-0.6)
sage: CC(a)
-0.629960524947436 + 1.09112363597172*I
sage: L = Qp(5)
sage: f = polygen(L)^3 - 2
sage: K.<a> = NumberField(x^3-2, embedding=f.roots()[0][0])
sage: a + L(1)
4 + 2*5^2 + 2*5^3 + 3*5^4 + 5^5 + 4*5^6 + 2*5^8 + 3*5^9 + 4*5^12 + 4*5^14 + 4*5^15 + 3*5^16 + 5^
sage: L.<b> = NumberField(x^6-x^2+1/10, embedding=1)
sage: K.<a> = NumberField(x^3-x+1/10, embedding=b^2)
sage: a+b
b^2 + b
sage: CC(a) == CC(b)^2
True
sage: K.coerce_embedding()
Generic morphism:
  From: Number Field in a with defining polynomial x^3 - x + 1/10
  To:   Number Field in b with defining polynomial x^6 - x^2 + 1/10
  Defn: a -> b^2
```

The `QuadraticField` and `CyclotomicField` constructors create an embedding by default unless otherwise specified.

```
sage: K.<zeta> = CyclotomicField(15)
sage: CC(zeta)
0.913545457642601 + 0.406736643075800*I
```

```
sage: L.<sqrtn3> = QuadraticField(-3)
sage: K(sqrtn3)
2*zeta^5 + 1
sage: sqrtn3 + zeta
2*zeta^5 + zeta + 1
```

An example involving a variable name that defines a function in PARI:

```
sage: theta = polygen(QQ, 'theta')
sage: M.<z> = NumberField([theta^3 + 4, theta^2 + 3]); M
Number Field in z0 with defining polynomial theta^3 + 4 over its base field
```

TESTS:

```
sage: x = QQ['x'].gen()
sage: y = ZZ['y'].gen()
sage: K = NumberField(x^3 + x + 3, 'a'); K
Number Field in a with defining polynomial x^3 + x + 3
sage: K.defining_polynomial().parent()
Univariate Polynomial Ring in x over Rational Field
```

```
sage: L = NumberField(y^3 + y + 3, 'a'); L
Number Field in a with defining polynomial y^3 + y + 3
sage: L.defining_polynomial().parent()
Univariate Polynomial Ring in y over Rational Field
```

NumberFieldTower(*v, names, check=True, embeddings=None*)

Return the tower of number fields defined by the polynomials or number fields in the list v.

This is the field constructed first from v[0], then over that field from v[1], etc. If all is False, then each v[i] must be irreducible over the previous fields. Otherwise a list of all possible fields defined by all those polynomials is output.

If names defines a variable name a, say, then the generators of the intermediate number fields are a0, a1, a2, ...

INPUT:

- `v` - a list of polynomials or number fields
- `names` - variable names
- `check` - bool (default: True) only relevant if all is False. Then check irreducibility of each input polynomial.

OUTPUT: a single number field or a list of number fields

EXAMPLES:

```
sage: k.<a,b,c> = NumberField([x^2 + 1, x^2 + 3, x^2 + 5]); k
Number Field in a with defining polynomial x^2 + 1 over its base field
sage: a^2
-1
sage: b^2
-3
sage: c^2
-5
sage: (a+b+c)^2
(2*b + 2*c)*a + 2*c*b - 9
```

The Galois group is a product of 3 groups of order 2:

```
sage: k.galois_group(type="pari")
Galois group PARI group [8, 1, 3, "E(8)=2[x]2[x]2"] of degree 8 of the Number Field in a with de
```

Repeatedly calling base_field allows us to descend the internally constructed tower of fields:

```
sage: k.base_field()
Number Field in b with defining polynomial x^2 + 3 over its base field
sage: k.base_field().base_field()
Number Field in c with defining polynomial x^2 + 5
sage: k.base_field().base_field().base_field()
Rational Field
```

In the following example the second polynomial is reducible over the first, so we get an error:

```
sage: v = NumberField([x^3 - 2, x^3 - 2], names='a')
...
ValueError: defining polynomial (x^3 - 2) must be irreducible
```

We mix polynomial parent rings:

```
sage: k.<y> = QQ[]
sage: m = NumberField([y^3 - 3, x^2 + x + 1, y^3 + 2], 'beta')
sage: m
Number Field in beta0 with defining polynomial y^3 - 3 over its base field
sage: m.base_field ()
Number Field in beta1 with defining polynomial x^2 + x + 1 over its base field
```

A tower of quadratic fields:

```
sage: K.<a> = NumberField([x^2 + 3, x^2 + 2, x^2 + 1])
sage: K
Number Field in a0 with defining polynomial x^2 + 3 over its base field
sage: K.base_field()
Number Field in a1 with defining polynomial x^2 + 2 over its base field
sage: K.base_field().base_field()
Number Field in a2 with defining polynomial x^2 + 1
```

A bigger tower of quadratic fields.

```
sage: K.<a2,a3,a5,a7> = NumberField([x^2 + p for p in [2,3,5,7]]); K
Number Field in a2 with defining polynomial x^2 + 2 over its base field
sage: a2^2
-2
sage: a3^2
-3
sage: (a2+a3+a5+a7)^3
((6*a5 + 6*a7)*a3 + 6*a7*a5 - 47)*a2 + (6*a7*a5 - 45)*a3 - 41*a5 - 37*a7
```

class **NumberField_absolute**(*polynomial, name, latex_name=None, check=True, embedding=None*)

Minkowski_embedding(*B=None, prec=None*)

Return an nxn matrix over RDF whose columns are the images of the basis $\{1, \alpha, \ldots, \alpha^{n-1}\}$ of self over $\mathbf{Q}$ (as vector spaces), where here α is the generator of self over $\mathbf{Q}$, i.e. self.gen(0). If B is not None, return the images of the vectors in B as the columns instead. If prec is not None, use RealField(prec) instead of RDF.

This embedding is the so-called "Minkowski embedding" of a number field in $\mathbf{R}^n$: given the n embeddings $\sigma_1, \ldots, \sigma_n$ of self in $\mathbf{C}$, write $\sigma_1, \ldots, \sigma_r$ for the real embeddings, and $\sigma_{r+1}, \ldots, \sigma_{r+s}$ for choices of one

of each pair of complex conjugate embeddings (in our case, we simply choose the one where the image of α has positive real part). Here (r, s) is the signature of self. Then the Minkowski embedding is given by

$$x \mapsto (\sigma_1(x), \ldots, \sigma_r(x), \sqrt{2}\Re(\sigma_{r+1}(x)), \sqrt{2}\Im(\sigma_{r+1}(x)), \ldots, \sqrt{2}\Re(\sigma_{r+s}(x)), \sqrt{2}\Im(\sigma_{r+s}(x)))$$

Equivalently, this is an embedding of self in $\mathbf{R}^n$ so that the usual norm on $\mathbf{R}^n$ coincides with $|x| = \sum_i |\sigma_i(x)|^2$ on self.

TODO: This could be much improved by implementing homomorphisms over VectorSpaces.

EXAMPLES:

```
sage: F.<alpha> = NumberField(x^3+2)
sage: F.Minkowski_embedding()
[ 1.00000000000000 -1.25992104989487  1.58740105196820]
[ 1.41421356237... 0.8908987181... -1.12246204830...]
[0.000000000000000  1.54308184421...  1.94416129723...]
sage: F.Minkowski_embedding([1, alpha+2, alpha^2-alpha])
[ 1.00000000000000 0.740078950105127  2.84732210186307]
[ 1.41421356237...  3.7193258428... -2.01336076644...]
[0.000000000000000  1.54308184421... 0.40107945302...]
sage: F.Minkowski_embedding() * (alpha + 2).vector().transpose()
[0.740078950105127]
[ 3.7193258428...]
[ 1.54308184421...]
```

absolute_degree()

A synonym for degree.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.absolute_degree()
2
```

absolute_different()

A synonym for different.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.absolute_different()
Fractional ideal (2)
```

absolute_discriminant()

A synonym for discriminant.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.absolute_discriminant()
-4
```

absolute_polynomial()

A synonym for polynomial.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.absolute_polynomial()
x^2 + 1
```

absolute_vector_space()

A synonym for vector_space.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.absolute_vector_space()
(Vector space of dimension 2 over Rational Field,
 Isomorphism map:
  From: Vector space of dimension 2 over Rational Field
  To:   Number Field in i with defining polynomial x^2 + 1,
 Isomorphism map:
  From: Number Field in i with defining polynomial x^2 + 1
  To:   Vector space of dimension 2 over Rational Field)
```

automorphisms()

Compute all Galois automorphisms of self.

This uses PARI's code{nfgaloisconj} and is much faster than root finding for many fields.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 10000)
sage: K.automorphisms()
[
Ring endomorphism of Number Field in a with defining polynomial x^2 + 10000
  Defn: a |--> a,
Ring endomorphism of Number Field in a with defining polynomial x^2 + 10000
  Defn: a |--> -a
]
```

Here's a larger example, that would take some time if we found roots instead of using PARI's specialized machinery:

```
sage: K = NumberField(x^6 - x^4 - 2*x^2 + 1, 'a')
sage: len(K.automorphisms())
2
```

L is the Galois closure of K:

```
sage: L = NumberField(x^24 - 84*x^22 + 2814*x^20 - 15880*x^18 - 409563*x^16 - 8543892*x^14 +
sage: len(L.automorphisms())
24
```

base_field()

Returns the base field of self, which is always QQ

EXAMPLES:

```
sage: K = CyclotomicField(5)
sage: K.base_field()
Rational Field
```

change_names(*names*)

Return number field isomorphic to self but with the given generator name.

INPUT:

- `names` - should be exactly one variable name.

Also, `K.structure()` returns from_K and to_K, where from_K is an isomorphism from K to self and to_K is an isomorphism from self to K.

EXAMPLES:

```
sage: K.<z> = NumberField(x^2 + 3); K
Number Field in z with defining polynomial x^2 + 3
sage: L.<ww> = K.change_names()
sage: L
Number Field in ww with defining polynomial x^2 + 3
sage: L.structure()[0]
```

```
Isomorphism given by variable name change map:
  From: Number Field in ww with defining polynomial x^2 + 3
  To:   Number Field in z with defining polynomial x^2 + 3
sage: L.structure()[0](ww + 5/3)
z + 5/3
```

embeddings(*K*)

Compute all field embeddings of self into the field K (which need not even be a number field, e.g., it could be the complex numbers). This will return an identical result when given K as input again.

If possible, the most natural embedding of self into K is put first in the list.

INPUT:

- K - a number field

EXAMPLES:

```
sage: K.<a> = NumberField(x^3 - 2)
sage: L.<a1> = K.galois_closure(); L
Number Field in a1 with defining polynomial x^6 + 40*x^3 + 1372
sage: K.embeddings(L)[0]
Ring morphism:
  From: Number Field in a with defining polynomial x^3 - 2
  To:   Number Field in a1 with defining polynomial x^6 + 40*x^3 + 1372
  Defn: a |--> 1/84*a1^4 + 13/42*a1
sage: K.embeddings(L)  is K.embeddings(L)
True
```

We embed a quadratic field into a cyclotomic field:

```
sage: L.<a> = QuadraticField(-7)
sage: K = CyclotomicField(7)
sage: L.embeddings(K)
[
Ring morphism:
  From: Number Field in a with defining polynomial x^2 + 7
  To:   Cyclotomic Field of order 7 and degree 6
  Defn: a |--> 2*zeta7^4 + 2*zeta7^2 + 2*zeta7 + 1,
Ring morphism:
  From: Number Field in a with defining polynomial x^2 + 7
  To:   Cyclotomic Field of order 7 and degree 6
  Defn: a |--> -2*zeta7^4 - 2*zeta7^2 - 2*zeta7 - 1
]
```

We embed a cubic field in the complex numbers:

```
sage: K.<a> = NumberField(x^3 - 2)
sage: K.embeddings(CC)
[
Ring morphism:
  From: Number Field in a with defining polynomial x^3 - 2
  To:   Complex Field with 53 bits of precision
  Defn: a |--> -0.62996052494743... - 1.09112363597172*I,
Ring morphism:
  From: Number Field in a with defining polynomial x^3 - 2
  To:   Complex Field with 53 bits of precision
  Defn: a |--> -0.62996052494743... + 1.09112363597172*I,
Ring morphism:
  From: Number Field in a with defining polynomial x^3 - 2
  To:   Complex Field with 53 bits of precision
  Defn: a |--> 1.25992104989487
]
```

galois_closure(*names=None, map=False*)

Return number field K that is the Galois closure of self, i.e., is generated by all roots of the defining polynomial of self, and possibly an embedding of self into K.

INPUT:

- `names` - variable name for Galois closure
- `map` - (default: False) also return an embedding of self into K

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 - 2)
sage: M = K.galois_closure('b'); M
Number Field in b with defining polynomial x^8 + 28*x^4 + 2500
sage: L.<a2> = K.galois_closure(); L
Number Field in a2 with defining polynomial x^8 + 28*x^4 + 2500
sage: K.galois_group(names=("a3")).order()
8

sage: phi = K.embeddings(L)[0]
sage: phi(K.0)
1/120*a2^5 + 19/60*a2
sage: phi(K.0).minpoly()
x^4 - 2

sage: L, phi = K.galois_closure('b', map=True)
sage: L
Number Field in b with defining polynomial x^8 + 28*x^4 + 2500
sage: phi
Ring morphism:
  From: Number Field in a with defining polynomial x^4 - 2
  To:   Number Field in b with defining polynomial x^8 + 28*x^4 + 2500
  Defn: a |--> 1/240*b^5 - 41/120*b
```

TESTS:

Let's make sure we're renaming correctly:

```
sage: L, phi = K.galois_closure('cc', map=True)
sage: L
Number Field in cc with defining polynomial x^8 + 28*x^4 + 2500
sage: phi
Ring morphism:
  From: Number Field in a with defining polynomial x^4 - 2
  To:   Number Field in cc with defining polynomial x^8 + 28*x^4 + 2500
  Defn: a |--> 1/240*cc^5 - 41/120*cc
```

is_absolute()

Returns True since self is an absolute field.

EXAMPLES:

```
sage: K = CyclotomicField(5)
sage: K.is_absolute()
True
```

maximal_order(*v=None*)

Return the maximal order, i.e., the ring of integers, associated to this number field.

INPUT:

- `v` - (default: None) None, a prime, or a list of primes.
  - –if v is None, return the maximal order.
  - –if v is a prime, return an order that is p-maximal.

–if v is a list, return an order that is maximal at each prime in the list v.

EXAMPLES: In this example, the maximal order cannot be generated by a single element.

```
sage: k.<a> = NumberField(x^3 + x^2 - 2*x+8)
sage: o = k.maximal_order()
sage: o
Maximal Order in Number Field in a with defining polynomial x^3 + x^2 - 2*x + 8
```

We compute p-maximal orders for several p. Note that computing a p-maximal order is much faster in general than computing the maximal order:

```
sage: p = next_prime(10^22); q = next_prime(10^23)
sage: K.<a> = NumberField(x^3 - p*q)
sage: K.maximal_order([3]).basis()
[1/3*a^2 + 1/3*a + 1/3, a, a^2]
sage: K.maximal_order([2]).basis()
[1, a, a^2]
sage: K.maximal_order([p]).basis()
[1, a, a^2]
sage: K.maximal_order([q]).basis()
[1, a, a^2]
sage: K.maximal_order([p,3]).basis()
[1/3*a^2 + 1/3*a + 1/3, a, a^2]
```

An example with bigger discriminant:

```
sage: p = next_prime(10^97); q = next_prime(10^99)
sage: K.<a> = NumberField(x^3 - p*q)
sage: K.maximal_order(prime_range(10000)).basis()
[1, a, a^2]
```

optimized_representation(*names=None, both_maps=True*)

Return a field isomorphic to self with a better defining polynomial if possible, along with field isomorphisms from the new field to self and from self to the new field.

EXAMPLES: We construct a compositum of 3 quadratic fields, then find an optimized representation and transform elements back and forth.

```
sage: K = NumberField([x^2 + p for p in [5, 3, 2]],'a').absolute_field('b'); K
Number Field in b with defining polynomial x^8 + 40*x^6 + 352*x^4 + 960*x^2 + 576
sage: L, from_L, to_L = K.optimized_representation()
sage: L    # your answer may different, since algorithm is random
Number Field in a14 with defining polynomial x^8 + 4*x^6 + 7*x^4 + 36*x^2 + 81
sage: to_L(K.0)   # random
4/189*a14^7 - 1/63*a14^6 + 1/27*a14^5 + 2/9*a14^4 - 5/27*a14^3 + 8/9*a14^2 + 3/7*a14 + 3/7
sage: from_L(L.0)   # random
1/1152*a1^7 + 1/192*a1^6 + 23/576*a1^5 + 17/96*a1^4 + 37/72*a1^3 + 5/6*a1^2 + 55/24*a1 + 3/4
```

The transformation maps are mutually inverse isomorphisms.

```
sage: from_L(to_L(K.0))
b
sage: to_L(from_L(L.0))     # random
a14
```

optimized_subfields(*degree=0, name=None, both_maps=True*)

Return optimized representations of many (but *not* necessarily all!) subfields of self of degree 0, or of all possible degrees if degree is 0.

EXAMPLES:

```
sage: K = NumberField([x^2 + p for p in [5, 3, 2]],'a').absolute_field('b'); K
Number Field in b with defining polynomial x^8 + 40*x^6 + 352*x^4 + 960*x^2 + 576
```

```
sage: L = K.optimized_subfields(name='b')
sage: L[0][0]
Number Field in b0 with defining polynomial x - 1
sage: L[1][0]
Number Field in b1 with defining polynomial x^2 - x + 1
sage: [z[0] for z in L]          # random -- since algorithm is random
[Number Field in b0 with defining polynomial x - 1,
 Number Field in b1 with defining polynomial x^2 - x + 1,
 Number Field in b2 with defining polynomial x^4 - 5*x^2 + 25,
 Number Field in b3 with defining polynomial x^4 - 2*x^2 + 4,
 Number Field in b4 with defining polynomial x^8 + 4*x^6 + 7*x^4 + 36*x^2 + 81]
```

We examine one of the optimized subfields in more detail:

```
sage: M, from_M, to_M = L[2]
sage: M                              # random
Number Field in b2 with defining polynomial x^4 - 5*x^2 + 25
sage: from_M     # may be slightly random
Ring morphism:
  From: Number Field in b2 with defining polynomial x^4 - 5*x^2 + 25
  To:   Number Field in a1 with defining polynomial x^8 + 40*x^6 + 352*x^4 + 960*x^2 + 576
  Defn: b2 |--> -5/1152*a1^7 + 1/96*a1^6 - 97/576*a1^5 + 17/48*a1^4 - 95/72*a1^3 + 17/12*a1^
```

The to_M map is None, since there is no map from K to M:

```
sage: to_M
```

We apply the from_M map to the generator of M, which gives a rather large element of K:

```
sage: from_M(M.0)           # random
-5/1152*a1^7 + 1/96*a1^6 - 97/576*a1^5 + 17/48*a1^4 - 95/72*a1^3 + 17/12*a1^2 - 53/24*a1 - 1
```

Nevertheless, that large-ish element lies in a degree 4 subfield:

```
sage: from_M(M.0).minpoly()    # random
x^4 - 5*x^2 + 25
```

order(**gens, **kwds*)

Return the order with given ring generators in the maximal order of this number field.

INPUT:

- `gens` - list of elements of self; if no generators are given, just returns the cardinality of this number field (oo) for consistency.
- `check_is_integral` - bool (default: True), whether to check that each generator is integral.
- `check_rank` - bool (default: True), whether to check that the ring generated by gens is of full rank.
- `allow_subfield` - bool (default: False), if True and the generators do not generate an order, i.e., they generate a subring of smaller rank, instead of raising an error, return an order in a smaller number field.

EXAMPLES:

```
sage: k.<i> = NumberField(x^2 + 1)
sage: k.order(2*i)
Order in Number Field in i with defining polynomial x^2 + 1
sage: k.order(10*i)
Order in Number Field in i with defining polynomial x^2 + 1
sage: k.order(3)
...
ValueError: the rank of the span of gens is wrong
sage: k.order(i/2)
...
ValueError: each generator must be integral
```

Alternatively, an order can be constructed by adjoining elements to $\mathbf{Z}$:

places (*all_complex=False, prec=None*)

Return the collection of all places of self. By default, this returns the set of real places as homomorphisms into RIF first, followed by a choice of one of each pair of complex conjugate homomorphisms into CIF.

On the other hand, if prec is not None, we simply return places into RealField(prec) and ComplexField(prec) (or RDF, CDF if prec=53).

There is an optional flag all_complex, which defaults to False. If all_complex is True, then the real embeddings are returned as embeddings into CIF instead of RIF.

EXAMPLES:

```
sage: F.<alpha> = NumberField(x^3-100*x+1) ; F.places()
[Ring morphism:
From: Number Field in alpha with defining polynomial x^3 - 100*x + 1
To:   Real Field with 106 bits of precision
Defn: alpha |--> -10.004996254991811845733672192 80,
Ring morphism:
From: Number Field in alpha with defining polynomial x^3 - 100*x + 1
To:   Real Field with 106 bits of precision
Defn: alpha |--> 0.010000010000030000120000550002 73,
Ring morphism:
From: Number Field in alpha with defining polynomial x^3 - 100*x + 1
To:   Real Field with 106 bits of precision
Defn: alpha |--> 9.994996244991781845613530439509]

sage: F.<alpha> = NumberField(x^3+7) ; F.places()
[Ring morphism:
From: Number Field in alpha with defining polynomial x^3 + 7
To:   Real Field with 106 bits of precision
Defn: alpha |--> -1.91293118277238910119911683 9549,
Ring morphism:
From: Number Field in alpha with defining polynomial x^3 + 7
To:   Complex Field with 53 bits of precision
Defn: alpha |--> 0.956465591386195 + 1.65664699997230*I]

sage: F.<alpha> = NumberField(x^3+7) ; F.places(all_complex=True)
[Ring morphism:
From: Number Field in alpha with defining polynomial x^3 + 7
To:   Complex Field with 53 bits of precision
Defn: alpha |--> -1.91293118277239,
Ring morphism:
From: Number Field in alpha with defining polynomial x^3 + 7
To:   Complex Field with 53 bits of precision
Defn: alpha |--> 0.956465591386195 + 1.65664699997230*I]
sage: F.places(prec=10)
[Ring morphism:
From: Number Field in alpha with defining polynomial x^3 + 7
To:   Real Field with 10 bits of precision
Defn: alpha |--> -1.9,
Ring morphism:
From: Number Field in alpha with defining polynomial x^3 + 7
To:   Complex Field with 10 bits of precision
Defn: alpha |--> 0.96 + 1.7*I]
```

real_places (*prec=None*)

Return all real places of self as homomorphisms into RIF.

EXAMPLES:

```
sage: F.<alpha> = NumberField(x^4-7) ; F.real_places()
[Ring morphism:
From: Number Field in alpha with defining polynomial x^4 - 7
To:   Real Field with 106 bits of precision
Defn: alpha |--> -1.6265765616977857432112323454 94,
Ring morphism:
From: Number Field in alpha with defining polynomial x^4 - 7
To:   Real Field with 106 bits of precision
Defn: alpha |--> 1.6265765616977857432112323454 94]
```

relative_degree()

A synonym for degree.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.relative_degree()
2
```

relative_different()

A synonym for different.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.relative_different()
Fractional ideal (2)
```

relative_discriminant()

A synonym for discriminant.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.relative_discriminant()
-4
```

relative_polynomial()

A synonym for polynomial.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.relative_polynomial()
x^2 + 1
```

relative_vector_space()

A synonym for vector_space.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.relative_vector_space()
(Vector space of dimension 2 over Rational Field,
 Isomorphism map:
  From: Vector space of dimension 2 over Rational Field
  To:   Number Field in i with defining polynomial x^2 + 1,
 Isomorphism map:
  From: Number Field in i with defining polynomial x^2 + 1
  To:   Vector space of dimension 2 over Rational Field)
```

relativize(*alpha, names*)

Given an element in self or an embedding of a subfield into self, return a relative number field K isomorphic to self that is relative over the absolute field $\mathbf{Q}(\alpha)$ or the domain of $alpha$, along with isomorphisms from K to self and from self to K.

INPUT:

- `alpha` - an element of self or an embedding of a subfield into self
- `names` - 2-tuple of names of generator for output field K and the subfield QQ(alpha) names[0] generators K and names[1] QQ(alpha).

OUTPUT:

K – relative number field

Also, code{K.structure()} returns from_K and to_K, where from_K is an isomorphism from K to self and to_K is an isomorphism from self to K.

EXAMPLES:

```
sage: K.<a> = NumberField(x^10 - 2)
sage: L.<c,d> = K.relativize(a^4 + a^2 + 2); L
Number Field in c with defining polynomial x^2 - 1/5*d^4 + 8/5*d^3 - 23/5*d^2 + 7*d - 18/5 o
sage: c.absolute_minpoly()
x^10 - 2
sage: d.absolute_minpoly()
x^5 - 10*x^4 + 40*x^3 - 90*x^2 + 110*x - 58
sage: (a^4 + a^2 + 2).minpoly()
x^5 - 10*x^4 + 40*x^3 - 90*x^2 + 110*x - 58
sage: from_L, to_L = L.structure()
sage: to_L(a)
c
sage: to_L(a^4 + a^2 + 2)
d
sage: from_L(to_L(a^4 + a^2 + 2))
a^4 + a^2 + 2
```

The following demonstrates distinct embeddings of a subfield into a larger field:

```
sage: K.<a> = NumberField(x^4 + 2*x^2 + 2)
sage: K0 = K.subfields(2)[0][0]; K0
Number Field in a0 with defining polynomial x^2 - 2*x + 2
sage: rho, tau = K0.embeddings(K)
sage: L0 = K.relativize(rho(K0.gen()), 'b'); L0
Number Field in b0 with defining polynomial x^2 - b1 + 2 over its base field
sage: L1 = K.relativize(rho, 'b'); L1
Number Field in b0 with defining polynomial x^2 - a0 + 2 over its base field
sage: L2 = K.relativize(tau, 'b'); L2
Number Field in b0 with defining polynomial x^2 + a0 over its base field
sage: L0.base_field() is K0
False
sage: L1.base_field() is K0
True
sage: L2.base_field() is K0
True
```

Here we see that with the different embeddings, the relative norms are different:

```
sage: a0 = K0.gen()
sage: L1_into_K, K_into_L1 = L1.structure()
sage: L2_into_K, K_into_L2 = L2.structure()
sage: len(K.factor(41))
4
sage: w1 = -a^2 + a + 1; P = K.ideal([w1])
sage: Pp = L1.ideal(K_into_L1(w1)).ideal_below(); Pp == K0.ideal([4*a0 + 1])
True
sage: Pp == w1.norm(rho)
True
```

```
sage: w2 = a^2 + a - 1; Q = K.ideal([w2])
sage: Qq = L2.ideal(K_into_L2(w2)).ideal_below(); Qq == K0.ideal([-4*a0 + 9])
True
sage: Qq == w2.norm(tau)
True

sage: Pp == Qq
False
```

TESTS:

We can relativize over the whole field:

```
sage: K.<a> = NumberField(x^4 + 2*x^2 + 2)
sage: K.relativize(K.gen(), 'a')
Number Field in a0 with defining polynomial x - a1 over its base field
sage: K.relativize(2*K.gen(), 'a')
Number Field in a0 with defining polynomial x - 1/2*a1 over its base field
```

We can relativize over the prime field:

```
sage: L = K.relativize(K(1), 'a'); L
Number Field in a0 with defining polynomial x^4 + 2*x^2 + 2 over its base field
sage: L.base_field()
Number Field in a1 with defining polynomial x - 1
sage: L.base_field().base_field()
Rational Field

sage: L = K.relativize(K(2), 'a'); L
Number Field in a0 with defining polynomial x^4 + 2*x^2 + 2 over its base field
sage: L.base_field()
Number Field in a1 with defining polynomial x - 2
sage: L.base_field().base_field()
Rational Field
```

We can relativize over a relative field:

```
sage: K.<z> = CyclotomicField(16)
sage: L, L_into_K, _ = K.subfields(4)[0]; L
Number Field in z0 with defining polynomial x^4 + 16
sage: F, F_into_L, _ = L.subfields(2)[0]; F
Number Field in z00 with defining polynomial x^2 + 64

sage: L_over_F = L.relativize(F_into_L, 'c'); L_over_F
Number Field in c0 with defining polynomial x^2 - 1/2*z00 over its base field
sage: L_over_F_into_L, _ = L_over_F.structure()

sage: K_over_rel = K.relativize(L_into_K * L_over_F_into_L, 'a'); K_over_rel
Number Field in a0 with defining polynomial x^2 - 1/2*c0 over its base field
sage: K_over_rel.base_field() is L_over_F
True
sage: K_over_rel.structure()
(Relative number field morphism:
  From: Number Field in a0 with defining polynomial x^2 - 1/2*c0 over its base field
  To:   Cyclotomic Field of order 16 and degree 8
  Defn: a0 |--> z
        c0 |--> 2*z^2
        z00 |--> 8*z^4, Ring morphism:
  From: Cyclotomic Field of order 16 and degree 8
  To:   Number Field in a0 with defining polynomial x^2 - 1/2*c0 over its base field
  Defn: z |--> a0)
```

We can relativize over a really large field. This requires great care to not factor or do any operation that would trigger a pari nfinit() internally:

```
sage: K.<a> = CyclotomicField(3^3*2^3)
sage: R = K.relativize(a^(3^2), 't'); R
Number Field in t0 with defining polynomial x^9 - t1 over its base field
sage: R.structure()
(Relative number field morphism:
  From: Number Field in t0 with defining polynomial x^9 - t1 over its base field
  To:   Cyclotomic Field of order 216 and degree 72
  Defn: t0 |--> a
        t1 |--> a^9,
 Ring morphism:
  From: Cyclotomic Field of order 216 and degree 72
  To:   Number Field in t0 with defining polynomial x^9 - t1 over its base field
  Defn: a |--> t0)
```

subfields(*degree=0, name=None*)

EXAMPLES:

```
sage: K.<a> = NumberField( [x^3 - 2, x^2 + x + 1] )
sage: K = K.absolute_field('b')
sage: S = K.subfields()
sage: len(S)
6
sage: [k[0].polynomial() for k in S]
[x - 3,
 x^2 - 3*x + 9,
 x^3 - 3*x^2 + 3*x + 1,
 x^3 - 3*x^2 + 3*x + 1,
 x^3 - 3*x^2 + 3*x - 17,
 x^6 - 3*x^5 + 6*x^4 - 11*x^3 + 12*x^2 + 3*x + 1]
 sage: R.<t> = QQ[]
 sage: L = NumberField(t^3 - 3*t + 1, 'c')
 sage: [k[1] for k in L.subfields()]
 [Ring morphism:
   From: Number Field in c0 with defining polynomial t
   To:   Number Field in c with defining polynomial t^3 - 3*t + 1
   Defn: 0 |--> 0,
  Ring morphism:
   From: Number Field in c1 with defining polynomial t^3 - 3*t + 1
   To:   Number Field in c with defining polynomial t^3 - 3*t + 1
   Defn: c1 |--> c]
```

vector_space()

Return a vector space V and isomorphisms self –> V and V –> self.

OUTPUT:

- V - a vector space over the rational numbers
- from_V - an isomorphism from V to self
- to_V - an isomorphism from self to V

EXAMPLES:

```
sage: k.<a> = NumberField(x^3 + 2)
sage: V, from_V, to_V  = k.vector_space()
sage: from_V(V([1,2,3]))
3*a^2 + 2*a + 1
sage: to_V(1 + 2*a + 3*a^2)
(1, 2, 3)
```

```
sage: V
Vector space of dimension 3 over Rational Field
sage: to_V
Isomorphism map:
  From: Number Field in a with defining polynomial x^3 + 2
  To:   Vector space of dimension 3 over Rational Field
sage: from_V(to_V(2/3*a - 5/8))
2/3*a - 5/8
sage: to_V(from_V(V([0,-1/7,0])))
(0, -1/7, 0)
```

NumberField_absolute_v1(*poly, name, latex_name, canonical_embedding=None*)

This is used in pickling generic number fields.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field import NumberField_generic_v1
sage: R.<x> = QQ[]
sage: NumberField_generic_v1(x^2 + 1, 'i', 'i')
Number Field in i with defining polynomial x^2 + 1
```

class **NumberField_cyclotomic**(*n, names, embedding=None*)

Create a cyclotomic extension of the rational field.

The command CyclotomicField(n) creates the n-th cyclotomic field, obtained by adjoining an n-th root of unity to the rational field.

EXAMPLES:

```
sage: CyclotomicField(3)
Cyclotomic Field of order 3 and degree 2
sage: CyclotomicField(18)
Cyclotomic Field of order 18 and degree 6
sage: z = CyclotomicField(6).gen(); z
zeta6
sage: z^3
-1
sage: (1+z)^3
6*zeta6 - 3
```

```
sage: K = CyclotomicField(197)
sage: loads(K.dumps()) == K
True
sage: loads((z^2).dumps()) == z^2
True
```

```
sage: cf12 = CyclotomicField( 12 )
sage: z12 = cf12.0
sage: cf6 = CyclotomicField( 6 )
sage: z6 = cf6.0
sage: FF = Frac( cf12['x'] )
sage: x = FF.0
sage: z6*x^3/(z6 + x)
zeta12^2*x^3/(x + zeta12^2)
```

```
sage: cf6 = CyclotomicField(6) ; z6 = cf6.gen(0)
sage: cf3 = CyclotomicField(3) ; z3 = cf3.gen(0)
sage: cf3(z6)
```

```
zeta3 + 1
sage: cf6(z3)
zeta6 - 1
sage: type(cf6(z3))
<type 'sage.rings.number_field.number_field_element_quadratic.NumberFieldElement_quadratic'>
sage: cf1 = CyclotomicField(1) ; z1 = cf1.0
sage: cf3(z1)
1
sage: type(cf3(z1))
<type 'sage.rings.number_field.number_field_element_quadratic.NumberFieldElement_quadratic'>
```

complex_embedding(*prec=53*)

Return the embedding of this cyclotomic field into the approximate complex field with precision prec obtained by sending the generator ζ of self to exp(2*pi*i/n), where n is the multiplicative order of ζ.

If prec is 53 (the default), then the complex double field is used; otherwise the arbitrary precision (but slow) complex field is used. EXAMPLES:

```
sage: C = CyclotomicField(4)
sage: C.complex_embedding()
Ring morphism:
  From: Cyclotomic Field of order 4 and degree 2
  To:   Complex Double Field
  Defn: zeta4 |--> 6.12323399574e-17 + 1.0*I
```

Note in the example above that the way zeta is computed (using sin and cosine in MPFR) means that only the prec bits of the number after the decimal point are valid.

```
sage: K = CyclotomicField(3)
sage: phi = K.complex_embedding(10)
sage: phi(K.0)
-0.50 + 0.87*I
sage: phi(K.0^3)
1.0
sage: phi(K.0^3 - 1)
0
sage: phi(K.0^3 + 7)
8.0
```

complex_embeddings(*prec=53*)

Return all embeddings of this cyclotomic field into the approximate complex field with precision prec.

If prec is 53 (the default), then the complex double field is used; otherwise the arbitrary precision (but slow) complex field is used. If you want 53-bit arbitrary precision then do `self.embeddings(ComplexField(53))`.

EXAMPLES:

```
sage: CyclotomicField(5).complex_embeddings()
[
Ring morphism:
  From: Cyclotomic Field of order 5 and degree 4
  To:   Complex Double Field
  Defn: zeta5 |--> 0.309016994375 + 0.951056516295*I,
Ring morphism:
  From: Cyclotomic Field of order 5 and degree 4
  To:   Complex Double Field
  Defn: zeta5 |--> -0.809016994375 + 0.587785252292*I,
Ring morphism:
  From: Cyclotomic Field of order 5 and degree 4
  To:   Complex Double Field
```

```
  Defn: zeta5 |--> -0.809016994375 - 0.587785252292*I,
Ring morphism:
  From: Cyclotomic Field of order 5 and degree 4
  To:   Complex Double Field
  Defn: zeta5 |--> 0.309016994375 - 0.951056516295*I
]
```

different()

Returns the different ideal of the cyclotomic field self.

EXAMPLES:

```
sage: C20 = CyclotomicField(20)
sage: C20.different()
Fractional ideal (-4*zeta20^7 + 8*zeta20^5 - 12*zeta20^3 + 6*zeta20)
sage: C18 = CyclotomicField(18)
sage: D = C18.different().norm()
sage: D == C18.discriminant().abs()
True
```

discriminant(*v=None*)

Returns the discriminant of the ring of integers of the cyclotomic field self, or if v is specified, the determinant of the trace pairing on the elements of the list v.

Uses the formula for the discriminant of a prime power cyclotomic field and Hilbert Theorem 88 on the discriminant of composita.

INPUT:

- `v (optional)` - list of element of this number field

OUTPUT: Integer if v is omitted, and Rational otherwise.

EXAMPLES:

```
sage: CyclotomicField(20).discriminant()
4000000
sage: CyclotomicField(18).discriminant()
-19683
```

integral_basis(*v=None*)

Return a list of elements of this number field that are a basis for the full ring of integers.

This field is cyclomotic, so this is a trivial computation, since the power basis on the generator is an integral basis. Thus the v parameter is ignored.

EXAMPLES:

```
sage: CyclotomicField(5).integral_basis()
[1, zeta5, zeta5^2, zeta5^3]
```

is_galois()

Return True since all cyclotomic fields are automatically Galois.

EXAMPLES:

```
sage: CyclotomicField(29).is_galois()
True
```

is_isomorphic(*other*)

Return True if the cyclotomic field self is isomorphic as a number field to other.

EXAMPLES:

```
sage: CyclotomicField(11).is_isomorphic(CyclotomicField(22))
True
sage: CyclotomicField(11).is_isomorphic(CyclotomicField(23))
```

```
False
sage: CyclotomicField(3).is_isomorphic(NumberField(x^2 + x +1, 'a'))
True
```

next_split_prime(*p=2*)

Return the next prime integer p that splits completely in this cyclotomic field (and does not ramify).

EXAMPLES:

```
sage: K.<z> = CyclotomicField(3)
sage: K.next_split_prime(7)
13
```

number_of_roots_of_unity()

Return number of roots of unity in this cyclotomic field.

EXAMPLES:

```
sage: K.<a> = CyclotomicField(21)
sage: K.number_of_roots_of_unity()
42
```

primitive_root_of_unity()

Return a generator of the roots of unity in this field.

EXAMPLES:

```
sage: K.<a> = CyclotomicField(3)
sage: z = K.primitive_root_of_unity(); z
-a
sage: z.multiplicative_order()
6

sage: K.<a> = CyclotomicField(10)
sage: z = K.primitive_root_of_unity(); z
a
sage: z.multiplicative_order()
10
```

real_embeddings(*prec=53*)

Return all embeddings of this cyclotomic field into the approximate real field with precision prec.

If prec is 53 (the default), then the real double field is used; otherwise the arbitrary precision (but slow) real field is used.

Mostly, of course, there are no such embeddings.

EXAMPLES:

```
sage: CyclotomicField(4).real_embeddings()
[]
sage: CyclotomicField(2).real_embeddings()
[
Ring morphism:
  From: Cyclotomic Field of order 2 and degree 1
  To:   Real Double Field
  Defn: -1 |--> -1.0
]
```

roots_of_unity()

Return all the roots of unity in this cyclotomic field, primitive or not.

EXAMPLES:

```
sage: K.<a> = CyclotomicField(3)
sage: zs = K.roots_of_unity(); zs
[1, a, -a - 1, -1, -a, a + 1]
sage: [ z**K.number_of_roots_of_unity() for z in zs ]
[1, 1, 1, 1, 1, 1]
```

signature()

Return (r1, r2), where r1 and r2 are the number of real embeddings and pairs of complex embeddings of this cyclotomic field, respectively.

Trivial since, apart from QQ, cyclotomic fields are totally complex.

EXAMPLES:

```
sage: CyclotomicField(5).signature()
(0, 2)
sage: CyclotomicField(2).signature()
(1, 0)
```

zeta(*n=None, all=False*)

Returns an element of multiplicative order n in this this cyclotomic field, if there is one. Raises a ValueError if there is not.

INPUT:

- `n` - integer (default: None, returns element of maximal order)
- `all` - bool (default: False) - whether to return a list of all n-th roots.

OUTPUT: root of unity or list

EXAMPLES:

```
sage: k = CyclotomicField(7)
sage: k.zeta()
zeta7
sage: k.zeta().multiplicative_order()
7
sage: k = CyclotomicField(49)
sage: k.zeta().multiplicative_order()
49
sage: k.zeta(7).multiplicative_order()
7
sage: k.zeta()
zeta49
sage: k.zeta(7)
zeta49^7

sage: K.<a> = CyclotomicField(7)
sage: K.zeta(14, all=True)
[-a^4, -a^5, a^5 + a^4 + a^3 + a^2 + a + 1, -a, -a^2, -a^3]
sage: K.<a> = CyclotomicField(10)
sage: K.zeta(20, all=True)
...
ValueError: n (=20) does not divide order of generator

sage: K.<a> = CyclotomicField(5)
sage: K.zeta(4)
...
ValueError: n (=4) does not divide order of generator
sage: v = K.zeta(5, all=True); v
[a, a^2, a^3, -a^3 - a^2 - a - 1]
sage: [b^5 for b in v]
[1, 1, 1, 1]
```

zeta_order()
Return the order of the maximal root of unity contained in this cyclotomic field.
EXAMPLES:

```
sage: CyclotomicField(1).zeta_order()
2
sage: CyclotomicField(4).zeta_order()
4
sage: CyclotomicField(5).zeta_order()
10
sage: CyclotomicField(5)._n()
5
sage: CyclotomicField(389).zeta_order()
778
```

NumberField_cyclotomic_v1(*zeta_order, name, canonical_embedding=None*)
This is used in pickling cyclotomic fields.
EXAMPLES:

```
sage: from sage.rings.number_field.number_field import NumberField_cyclotomic_v1
sage: NumberField_cyclotomic_v1(5,'a')
Cyclotomic Field of order 5 and degree 4
sage: NumberField_cyclotomic_v1(5,'a').variable_name()
'a'
```

class **NumberField_generic**(*polynomial, name, latex_name=None, check=True, embedding=None*)
EXAMPLES:

```
sage: K.<a> = NumberField(x^3 - 2); K
Number Field in a with defining polynomial x^3 - 2
sage: loads(K.dumps()) == K
True
```

absolute_degree()
Return the degree of self over $\mathbf{Q}$.
EXAMPLES:

```
sage: NumberField(x^3 + x^2 + 997*x + 1, 'a').absolute_degree()
3
sage: NumberField(x + 1, 'a').absolute_degree()
1
sage: NumberField(x^997 + 17*x + 3, 'a', check=False).absolute_degree()
997
```

absolute_field(*names*)
Returns self as an absolute extension over QQ.
OUTPUT:

- K - this number field (since it is already absolute)

Also, `K.structure()` returns from_K and to_K, where from_K is an isomorphism from K to self and to_K is an isomorphism from self to K.
EXAMPLES:

```
sage: K = CyclotomicField(5)
sage: K.absolute_field('a')
Number Field in a with defining polynomial x^4 + x^3 + x^2 + x + 1
```

absolute_polynomial_ntl()

category()

Return the category of number fields.

EXAMPLES:

```
sage: NumberField(x^2 + 3, 'a').category()
Category of number fields
sage: category(NumberField(x^2 + 3, 'a'))
Category of number fields
```

The special types of number fields, e.g., quadratic fields, don't have their own category:

```
sage: QuadraticField(2,'d').category()
Category of number fields
```

change_generator(*alpha, name=None, names=None*)

Given the number field self, construct another isomorphic number field K generated by the element alpha of self, along with isomorphisms from K to self and from self to K.

EXAMPLES:

```
sage: L.<i> = NumberField(x^2 + 1); L
Number Field in i with defining polynomial x^2 + 1
sage: K, from_K, to_K = L.change_generator(i/2 + 3)
sage: K
Number Field in i0 with defining polynomial x^2 - 6*x + 37/4
sage: from_K
Ring morphism:
  From: Number Field in i0 with defining polynomial x^2 - 6*x + 37/4
  To:   Number Field in i with defining polynomial x^2 + 1
  Defn: i0 |--> 1/2*i + 3
sage: to_K
Ring morphism:
  From: Number Field in i with defining polynomial x^2 + 1
  To:   Number Field in i0 with defining polynomial x^2 - 6*x + 37/4
  Defn: i |--> 2*i0 - 6
```

We can also do

```
sage: K.<c>, from_K, to_K = L.change_generator(i/2 + 3); K
Number Field in c with defining polynomial x^2 - 6*x + 37/4
```

We compute the image of the generator $\sqrt{-1}$ of L.

```
sage: to_K(i)
2*c - 6
```

Note that the image is indeed a square root of -1.

```
sage: to_K(i)^2
-1
sage: from_K(to_K(i))
i
sage: to_K(from_K(c))
c
```

characteristic()

Return the characteristic of this number field, which is of course 0.

EXAMPLES:

```
sage: k.<a> = NumberField(x^99 + 2); k
Number Field in a with defining polynomial x^99 + 2
sage: k.characteristic()
0
```

class_group(*proof=None, names='c'*)

Return the class group of the ring of integers of this number field.

INPUT:

- `proof` - if True then compute the classgroup provably correctly. Default is True. Call number_field_proof to change this default globally.
- `names` - names of the generators of this class group.

OUTPUT: The class group of this number field.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 23)
sage: G = K.class_group(); G
Class group of order 3 with structure C3 of Number Field in a with defining polynomial x^2 +
sage: G.0
Fractional ideal class (2, 1/2*a - 1/2)
sage: G.gens()
[Fractional ideal class (2, 1/2*a - 1/2)]

sage: G.number_field()
Number Field in a with defining polynomial x^2 + 23
sage: G is K.class_group()
True
sage: G is K.class_group(proof=False)
False
sage: G.gens()
[Fractional ideal class (2, 1/2*a - 1/2)]
```

There can be multiple generators:

```
sage: k.<a> = NumberField(x^2 + 20072)
sage: G = k.class_group(); G
Class group of order 76 with structure C38 x C2 of Number Field in a with defining polynomia
sage: G.gens()
[Fractional ideal class (41, a + 10), Fractional ideal class (2, -1/2*a)]
sage: G.0
Fractional ideal class (41, a + 10)
sage: G.0^20
Fractional ideal class (43, a + 3)
sage: G.0^38
Trivial principal fractional ideal class
sage: G.1
Fractional ideal class (2, -1/2*a)
sage: G.1^2
Trivial principal fractional ideal class
```

Class groups of Hecke polynomials tend to be very small:

```
sage: f = ModularForms(97, 2).T(2).charpoly()
sage: f.factor()
(x - 3) * (x^3 + 4*x^2 + 3*x - 1) * (x^4 - 3*x^3 - x^2 + 6*x - 1)
sage: for g,_ in f.factor(): print NumberField(g,'a').class_group().order()
...
1
1
1
```

class_number(*proof=None*)

Return the class number of this number field, as an integer.

INPUT:

- `proof` - bool (default: True unless you called number_field_proof)

EXAMPLES:

```
sage: NumberField(x^2 + 23, 'a').class_number()
3
sage: NumberField(x^2 + 163, 'a').class_number()
1
sage: NumberField(x^3 + x^2 + 997*x + 1, 'a').class_number(proof=False)
1539
```

completion (*p, prec, extras={}*)

Returns the completion of self at p to the specified precision. Only implemented at archimedean places, and then only if an embedding has been fixed.

EXAMPLES:

```
sage: K.<a> = QuadraticField(2)
sage: K.completion(infinity, 100)
Real Field with 100 bits of precision
sage: K.<zeta> = CyclotomicField(12)
sage: K.completion(infinity, 53, extras={'type': 'RDF'})
Complex Double Field
sage: zeta + 1.5                              # implicit test
2.36602540378444 + 0.500000000000000*I
```

complex_embeddings (*prec=53*)

Return all homomorphisms of this number field into the approximate complex field with precision prec.

If prec is 53 (the default), then the complex double field is used; otherwise the arbitrary precision (but slow) complex field is used. If you want 53-bit arbitrary precision then do `self.embeddings(ComplexField(53))`.

EXAMPLES:

```
sage: k.<a> = NumberField(x^5 + x + 17)
sage: v = k.complex_embeddings()
sage: ls = [phi(k.0^2) for phi in v] ; ls # random order
[2.97572074038...,
 -2.40889943716 + 1.90254105304*I,
 -2.40889943716 - 1.90254105304*I,
 0.921039066973 + 3.07553311885*I,
 0.921039066973 - 3.07553311885*I]
sage: K.<a> = NumberField(x^3 + 2)
sage: ls = K.complex_embeddings() ; ls # random order
[
Ring morphism:
  From: Number Field in a with defining polynomial x^3 + 2
  To:   Complex Double Field
  Defn: a |--> -1.25992104989...,
Ring morphism:
  From: Number Field in a with defining polynomial x^3 + 2
  To:   Complex Double Field
  Defn: a |--> 0.629960524947 - 1.09112363597*I,
Ring morphism:
  From: Number Field in a with defining polynomial x^3 + 2
  To:   Complex Double Field
  Defn: a |--> 0.629960524947 + 1.09112363597*I
]
```

composite_fields (*other, names=None, both_maps=False, preserve_embedding=True*)

List of all possible composite number fields formed from self and other, as well as possibly embeddings into the compositum. See the documentation for both_maps below.

If preserve_embedding is code{True} and if self and other em{both} have embeddings into the same

ambient field, only compositums respecting both embeddings are returned. If one (or both) of self or other does not have an embedding, or they do not have embeddings into the same ambient field, or preserve_embedding is not code{True} all possible composite number fields are returned.

INPUT:

- `other` - a number field
- `names` - generator name for composite fields
- `both_maps` - (default: False) if True, return quadruples (F, self_into_F, other_into_F, k) such that self_into_F maps self into F, other_into_F maps other into F, and k is an integer such that other_into_F(other.gen()) + k*self_into_F(self.gen()) == F.gen()
- `preserve_embedding` - (default: True) return only compositums with compatible ambient embeddings.

OUTPUT:

- `list` - list of the composite fields, possibly with maps.

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 - 2)
sage: K.composite_fields(K)
[Number Field in a0 with defining polynomial x^4 - 162,
 Number Field in a1 with defining polynomial x^4 - 2,
 Number Field in a2 with defining polynomial x^8 + 28*x^4 + 2500]
sage: k.<a> = NumberField(x^3 + 2)
sage: m.<b> = NumberField(x^3 + 2)
sage: k.composite_fields(m, 'c')
[Number Field in c0 with defining polynomial x^3 - 2,
 Number Field in c1 with defining polynomial x^6 - 40*x^3 + 1372]
```

Let's get the maps as well:

```
sage: Q1.<a> = NumberField(x^2 + 2, 'b').extension(x^2 + 3, 'c').absolute_field()
sage: Q2.<b> = NumberField(x^2 + 3, 'a').extension(x^2 + 5, 'c').absolute_field()
sage: Q1.composite_fields(Q2)
[Number Field in ab0 with defining polynomial x^8 + 64*x^6 + 904*x^4 + 3840*x^2 + 3600,
 Number Field in ab1 with defining polynomial x^8 + 160*x^6 + 6472*x^4 + 74880*x^2 + 1296]

sage: F, Q1_into_F, Q2_into_F, k = Q1.composite_fields(Q2, both_maps=True)[0]
sage: F
Number Field in ab0 with defining polynomial x^8 + 64*x^6 + 904*x^4 + 3840*x^2 + 3600
sage: Q1_into_F
Ring morphism:
  From: Number Field in a with defining polynomial x^4 + 10*x^2 + 1
  To:   Number Field in ab0 with defining polynomial x^8 + 64*x^6 + 904*x^4 + 3840*x^2 + 360
  Defn: a |--> 19/14400*ab0^7 + 137/1800*ab0^5 + 2599/3600*ab0^3 + 8/15*ab0
sage: Q2_into_F
Ring morphism:
  From: Number Field in b with defining polynomial x^4 + 16*x^2 + 4
  To:   Number Field in ab0 with defining polynomial x^8 + 64*x^6 + 904*x^4 + 3840*x^2 + 360
  Defn: b |--> 19/7200*ab0^7 + 137/900*ab0^5 + 2599/1800*ab0^3 + 31/15*ab0

sage: Q1_into_F.domain() is Q1
True
sage: Q2_into_F(b) + k*Q1_into_F(a) == F.gen()
True
```

Let's check something about the "other" composite field:

```
sage: F, Q1_into_F, Q2_into_F, k = Q1.composite_fields(Q2, both_maps=True)[1]
sage: Q1_into_F.domain() is Q1
```

```
True
sage: Q2_into_F(b) + k*Q1_into_F(a) == F.gen()
True
```

TESTS:

Let's check that embeddings are being respected:

```
sage: x = ZZ['x'].0
sage: K0.<b> = CyclotomicField(7, 'a').subfields(3)[0][0].change_names()
sage: K1.<a1> = K0.extension(x^2 - 2*b^2, 'a1').absolute_field()
sage: K2.<a2> = K0.extension(x^2 - 3*b^2, 'a2').absolute_field()
sage: K1
Number Field in a1 with defining polynomial x^6 - 10*x^4 + 24*x^2 - 8
sage: K2
Number Field in a2 with defining polynomial x^6 - 15*x^4 + 54*x^2 - 27
sage: K1.is_isomorphic(K2)
False
```

We need embeddings, so we redefine:

```
sage: L1.<a1> = NumberField(K1.polynomial(), 'a1', embedding=CC.0)
sage: CDF(a1)
-0.629384245426
sage: L2.<a2> = NumberField(K2.polynomial(), 'a2', embedding=CC.0)
sage: CDF(a2)
-0.77083512672

sage: F, L1_into_F, L2_into_F, k = L1.composite_fields(L2, both_maps=True)[0]
sage: CDF(F.gen())
-0.141450881294
```

Both subfield generators have correct embeddings:

```
sage: CDF(L1_into_F(L1.gen())), CDF(L1.gen())
(-0.629384245426, -0.629384245426)
sage: CDF(L2_into_F(L2.gen())), CDF(L2.gen())
(-0.77083512672, -0.77083512672)
```

On the other hand, without embeddings, there are more composite fields:

```
sage: M1.<a1> = NumberField(L1.polynomial(), 'a1')
sage: M2.<a2> = NumberField(L2.polynomial(), 'a2')
sage: M1.composite_fields(M2)
[Number Field in a1a20 with defining polynomial x^12 - 50*x^10 + 613*x^8 - 1270*x^6 + 526*x^
 Number Field in a1a21 with defining polynomial x^12 - 50*x^10 + 865*x^8 - 6730*x^6 + 24970*
 Number Field in a1a22 with defining polynomial x^12 - 50*x^10 + 865*x^8 - 6310*x^6 + 18670*
```

Here's another example:

```
sage: Q1.<a> = NumberField(x^4 + 10*x^2 + 1, embedding=CC.0); Q1, CDF(a)
(Number Field in a with defining polynomial x^4 + 10*x^2 + 1, 0.317837245196*I)
sage: Q2.<b> = NumberField(x^4 + 16*x^2 + 4, embedding=CC.0); Q2, CDF(b)
(Number Field in b with defining polynomial x^4 + 16*x^2 + 4, 0.504017169931*I)

sage: len(Q1.composite_fields(Q2))
1
sage: F, Q1_into_F, Q2_into_F, k2 = Q1.composite_fields(Q2, both_maps=True)[0]
sage: F, CDF(F.gen())
(Number Field in ab1 with defining polynomial x^8 + 160*x^6 + 6472*x^4 + 74880*x^2 + 1296,
 -0.131657320461*I)

sage: t = Q2_into_F(Q2.gen()) + k2*Q1_into_F(Q1.gen()); t, CDF(t)
```

```
(ab1, -0.131657320461*I)
sage: abs(t.minpoly()(CDF(t))) < 1e-8
True
```

Let's check that the preserve_embedding flag is respected:

```
sage: len(Q1.composite_fields(Q2))
1
sage: len(Q1.composite_fields(Q2, preserve_embedding=False))
2
```

Let's check that if only one field has an embedding, the resulting fields do not have an embedding:

```
sage: Q2.<b> = NumberField(x^4 + 16*x^2 + 4)
sage: Q2.coerce_embedding() is None
True

sage: Q1.composite_fields(Q2)
[Number Field in ab0 with defining polynomial x^8 + 64*x^6 + 904*x^4 + 3840*x^2 + 3600,
 Number Field in ab1 with defining polynomial x^8 + 160*x^6 + 6472*x^4 + 74880*x^2 + 1296]
sage: Q1.composite_fields(Q2)[0].coerce_embedding() is None
True
```

defining_polynomial()

Return the defining polynomial of this number field.

This is exactly the same as code{self.polynomal()}.

EXAMPLES:

```
sage: k5.<z> = CyclotomicField(5)
sage: k5.defining_polynomial()
x^4 + x^3 + x^2 + x + 1
sage: y = polygen(QQ,'y')
sage: k.<a> = NumberField(y^9 - 3*y + 5); k
Number Field in a with defining polynomial y^9 - 3*y + 5
sage: k.defining_polynomial()
y^9 - 3*y + 5
```

degree()

Return the degree of this number field.

EXAMPLES:

```
sage: NumberField(x^3 + x^2 + 997*x + 1, 'a').degree()
3
sage: NumberField(x + 1, 'a').degree()
1
sage: NumberField(x^997 + 17*x + 3, 'a', check=False).degree()
997
```

different()

Compute the different fractional ideal of this number field.

The codifferent is the fractional ideal of all x in K such that the the trace of xy is an integer for all $y \in O_K$.

The different is the integral ideal which is the inverse of the codifferent.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 23)
sage: d = k.different()
sage: d        # random sign in output
Fractional ideal (-a)
sage: d.norm()
```

```
23
sage: k.disc()
-23
```

The different is cached:

```
sage: d is k.different()
True
```

Another example:

```
sage: k.<b> = NumberField(x^2 - 123)
sage: d = k.different(); d
Fractional ideal (2*b)
sage: d.norm()
492
sage: k.disc()
492
```

disc(*v=None*)

Shortcut for self.discriminant.

EXAMPLES:

```
sage: k.<b> = NumberField(x^2 - 123)
sage: k.disc()
492
```

discriminant(*v=None*)

Returns the discriminant of the ring of integers of the number field, or if v is specified, the determinant of the trace pairing on the elements of the list v.

INPUT:

- `v (optional)` - list of element of this number field

OUTPUT: Integer if v is omitted, and Rational otherwise.

EXAMPLES:

```
sage: K.<t> = NumberField(x^3 + x^2 - 2*x + 8)
sage: K.disc()
-503
sage: K.disc([1, t, t^2])
-2012
sage: K.disc([1/7, (1/5)*t, (1/3)*t^2])
-2012/11025
sage: (5*7*3)^2
11025
```

elements_of_norm(*n, proof=None*)

Return a list of solutions modulo units of positive norm to $Norm(a) = n$, where a can be any integer in this number field.

INPUT:

- `proof` - default: True, unless you called number_field_proof and set it otherwise.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2+1)
sage: K.elements_of_norm(3)
[]
sage: K.elements_of_norm(50)
[7*a - 1, -5*a + 5, a - 7]           # 32-bit
[7*a - 1, -5*a + 5, -7*a - 1]        # 64-bit
```

extension(*poly, name=None, names=None, check=True, embedding=None*)

Return the relative extension of this field by a given polynomial.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3 - 2)
sage: R.<t> = K[]
sage: L.<b> = K.extension(t^2 + a); L
Number Field in b with defining polynomial t^2 + a over its base field
```

We create another extension.

```
sage: k.<a> = NumberField(x^2 + 1); k
Number Field in a with defining polynomial x^2 + 1
sage: y = var('y')
sage: m.<b> = k.extension(y^2 + 2); m
Number Field in b with defining polynomial y^2 + 2 over its base field
```

Note that b is a root of $y^2 + 2$:

```
sage: b.minpoly()
x^2 + 2
sage: b.minpoly('z')
z^2 + 2
```

A relative extension of a relative extension.

```
sage: k.<a> = NumberField([x^2 + 1, x^3 + x + 1])
sage: R.<z> = k[]
sage: L.<b> = NumberField(z^3 + 3 + a); L
Number Field in b with defining polynomial z^3 + a0 + 3 over its base field
```

factor(*n*)

Ideal factorization of the principal ideal of the ring of integers generated by n.

EXAMPLE: Here we show how to factor gaussian integers. First we form a number field defined by x^2+1:

```
sage: K.<I> = NumberField(x^2 + 1); K
Number Field in I with defining polynomial x^2 + 1
```

Here are the factors:

```
sage: fi, fj = K.factor(13); fi,fj
((Fractional ideal (-3*I - 2), 1), (Fractional ideal (3*I - 2), 1))
```

Now we extract the reduced form of the generators:

```
sage: zi = fi[0].gens_reduced()[0]; zi
-3*I - 2
sage: zj = fj[0].gens_reduced()[0]; zj
3*I - 2
```

We recover the integer that was factored in $\mathbf{Z}[i]$

```
sage: zi*zj
13
```

One can also factor elements of the number field:

```
sage: K.<a> = NumberField(x^2 + 1)
sage: K.factor(1/3)
Fractional ideal (3)^-1
sage: K.factor(1+a)
Fractional ideal (a + 1)
sage: K.factor(1+a/5)
(Fractional ideal (-3*a - 2)) * (Fractional ideal (a + 1)) * (Fractional ideal (-a - 2))^-1
```

AUTHORS:

- Alex Clemesha (2006-05-20): examples

fractional_ideal(**gens, **kwds*)

Return the ideal in $\mathcal{O}_K$ generated by gens. This overrides the `sage.rings.ring.Field` method to use the `sage.rings.ring.Ring` one instead, since we're not really concerned with ideals in a field but in its ring of integers.

INPUT:

- `gens` - a list of generators, or a number field ideal.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-2)
sage: K.fractional_ideal([1/a])
Fractional ideal (1/2*a^2)
```

One can also input in a number field ideal itself.

```
sage: K.fractional_ideal(K.ideal(a))
Fractional ideal (a)
```

The zero ideal is not a fractional ideal!

```
sage: K.fractional_ideal(0)
...
ValueError: gens must have a nonzero element (zero ideal is not a fractional ideal)
```

galois_group(*type=None, algorithm='pari', names=None*)

Return the Galois group of the Galois closure of this number field.

INPUT:

- `type` - `none`, `gap`, or `pari`. If None (the default), return an explicit group of automorphisms of self as a `GaloisGroup_v2` object. Otherwise, return a `GaloisGroup_v1` wrapper object based on a Pari or Gap transitive group object, which is quicker to compute, but rather less useful (in particular, it can't be made to act on self). If type = 'gap', the database_gap package should be installed.
- `algorithm` - 'pari', 'kash', 'magma'. (default: 'pari', except when the degree is >= 12 when 'kash' is tried.)
- `name` - a string giving a name for the generator of the Galois closure of self, when self is not Galois. This is ignored if type is not None.

Note that computing Galois groups as abstract groups is often much faster than computing them as explicit automorphism groups (but of course you get less information out!) For more (important!) documentation, so the documentation for Galois groups of polynomials over $\mathbf{Q}$, e.g., by typing `K.polynomial().galois_group?`, where K is a number field.

To obtain actual field homomorphisms from the number field to its splitting field, use type=None.

EXAMPLES:

With type `None`:

```
sage: k.<b> = NumberField(x^2 - 14) # a Galois extension
sage: G = k.galois_group(); G
Galois group of Number Field in b with defining polynomial x^2 - 14
sage: G.gen(0)
(1,2)
sage: G.gen(0)(b)
-b
sage: G.artin_symbol(k.primes_above(3)[0])
(1,2)

sage: k.<b> = NumberField(x^3 - x + 1) # not Galois
```

```
sage: G = k.galois_group(names='c'); G
Galois group of Galois closure in c of Number Field in b with defining polynomial x^3 - x +
sage: G.gen(0)
(1,2)(3,5)(4,6)
```

With type 'pari':

```
sage: NumberField(x^3-2, 'a').galois_group(type="pari")
Galois group PARI group [6, -1, 2, "S3"] of degree 3 of the Number Field in a with defining

sage: NumberField(x-1, 'a').galois_group(type="gap")    # optional - database_gap
Galois group Transitive group number 1 of degree 1 of the Number Field in a with defining po
sage: NumberField(x^2+2, 'a').galois_group(type="gap")  # optional - database_gap
Galois group Transitive group number 1 of degree 2 of the Number Field in a with defining po
sage: NumberField(x^3-2, 'a').galois_group(type="gap")  # optional - database_gap
Galois group Transitive group number 2 of degree 3 of the Number Field in a with defining po

sage: x = polygen(QQ)
sage: NumberField(x^3 + 2*x + 1, 'a').galois_group(pari_group=False)    # opticnal - databas
Galois group Transitive group number 2 of degree 3 of the Number Field in a with defining po
sage: NumberField(x^3 + 2*x + 1, 'a').galois_group(algorithm='magma')   # opticnal - magma,
Galois group Transitive group number 2 of degree 3 of the Number Field in a with defining po
```

EXPLICIT GALOIS GROUP: We compute the Galois group as an explicit group of automorphisms of the Galois closure of a field.

```
sage: K.<a> = NumberField(x^3 - 2)
sage: L.<b1> = K.galois_closure(); L
Number Field in b1 with defining polynomial x^6 + 40*x^3 + 1372
sage: G = End(L); G
Automorphism group of Number Field in b1 with defining polynomial x^6 + 40*x^3 + 1372
sage: G.list()
[
Ring endomorphism of Number Field in b1 with defining polynomial x^6 + 40*x^3 + 1372
  Defn: b1 |--> b1,
...
Ring endomorphism of Number Field in b1 with defining polynomial x^6 + 40*x^3 + 1372
  Defn: b1 |--> -2/63*b1^4 - 31/63*b1
]
sage: G[1](b1)
1/36*b1^4 + 1/18*b1
```

gen(*n=0*)

Return the generator for this number field.

INPUT:

- n - must be 0 (the default), or an exception is raised.

EXAMPLES:

```
sage: k.<theta> = NumberField(x^14 + 2); k
Number Field in theta with defining polynomial x^14 + 2
sage: k.gen()
theta
sage: k.gen(1)
...
IndexError: Only one generator.
```

gen_embedding()

If an embedding has been specified, return the image of the generator under that embedding. Otherwise return None.

EXAMPLES:

```
sage: QuadraticField(-7, 'a').gen_embedding()
2.645751311064591?*I
sage: NumberField(x^2+7, 'a').gen_embedding() # None
```

ideal(*gens, **kwds*)

K.ideal() returns a fractional ideal of the field, except for the zero ideal which is not a fractional ideal.

EXAMPLES:

```
sage: K.<i>=NumberField(x^2+1)
sage: K.ideal(2)
Fractional ideal (2)
sage: K.ideal(2+i)
Fractional ideal (i + 2)
sage: K.ideal(0)
Ideal (0) of Number Field in i with defining polynomial x^2 + 1
```

ideals_of_bdd_norm(*bound*)

All integral ideals of bounded norm.

INPUT:

- bound - a positive integer

OUTPUT: A dict of all integral ideals I such that Norm(I) <= bound, keyed by norm.

EXAMPLE:

```
sage: K.<a> = NumberField(x^2 + 23)
sage: d = K.ideals_of_bdd_norm(10)
sage: for n in d:
...       print n
...       for I in d[n]:
...           print I
1
Fractional ideal (1)
2
Fractional ideal (2, 1/2*a - 1/2)
Fractional ideal (2, 1/2*a + 1/2)
3
Fractional ideal (3, -1/2*a + 1/2)
Fractional ideal (3, -1/2*a - 1/2)
4
Fractional ideal (4, 1/2*a + 3/2)
Fractional ideal (2)
Fractional ideal (4, 1/2*a + 5/2)
5
6
Fractional ideal (-1/2*a + 1/2)
Fractional ideal (6, 1/2*a + 5/2)
Fractional ideal (6, 1/2*a + 7/2)
Fractional ideal (1/2*a + 1/2)
7
8
Fractional ideal (-1/2*a - 3/2)
Fractional ideal (4, a - 1)
Fractional ideal (4, a + 1)
Fractional ideal (1/2*a - 3/2)
9
Fractional ideal (9, 1/2*a + 11/2)
Fractional ideal (3)
```

```
Fractional ideal (9, 1/2*a + 7/2)
10
```

integral_basis(*v=None*)

Returns a list containing a ZZ-basis for the full ring of integers of this number field.

INPUT:

- v - None, a prime, or a list of primes. See the documentation for self.maximal_order.

EXAMPLES:

```
sage: K.<a> = NumberField(x^5 + 10*x + 1)
sage: K.integral_basis()
[1, a, a^2, a^3, a^4]
```

Next we compute the ring of integers of a cubic field in which 2 is an "essential discriminant divisor", so the ring of integers is not generated by a single element.

```
sage: K.<a> = NumberField(x^3 + x^2 - 2*x + 8)
sage: K.integral_basis()
[1, 1/2*a^2 + 1/2*a, a^2]
```

ALGORITHM: Uses the pari library.

is_absolute()

Returns True if self is an absolute field.

This function will be implemented in the derived classes.

EXAMPLES:

```
sage: K = CyclotomicField(5)
sage: K.is_absolute()
True
```

is_field()

Return True since a number field is a field.

EXAMPLES:

```
sage: NumberField(x^5 + x + 3, 'c').is_field()
True
```

is_galois()

Return True if this number field is a Galois extension of $\mathbf{Q}$.

EXAMPLES:

```
sage: NumberField(x^2 + 1, 'i').is_galois()
True
sage: NumberField(x^3 + 2, 'a').is_galois()
False
```

is_isomorphic(*other*)

Return True if self is isomorphic as a number field to other.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 1)
sage: m.<b> = NumberField(x^2 + 4)
sage: k.is_isomorphic(m)
True
sage: m.<b> = NumberField(x^2 + 5)
sage: k.is_isomorphic (m)
False
```

```
sage: k = NumberField(x^3 + 2, 'a')
sage: k.is_isomorphic(NumberField((x+1/3)^3 + 2, 'b'))
True
sage: k.is_isomorphic(NumberField(x^3 + 4, 'b'))
True
sage: k.is_isomorphic(NumberField(x^3 + 5, 'b'))
False
```

is_relative()

EXAMPLES:

```
sage: K.<a> = NumberField(x^10 - 2)
sage: K.is_absolute()
True
sage: K.is_relative()
False
```

is_totally_imaginary()

Return True if self is totally imaginary, and False otherwise.

Totally imaginary means that no isomorphic embedding of self into the complex numbers has image contained in the real numbers.

EXAMPLES:

```
sage: NumberField(x^2+2, 'alpha').is_totally_imaginary()
True
sage: NumberField(x^2-2, 'alpha').is_totally_imaginary()
False
sage: NumberField(x^4-2, 'alpha').is_totally_imaginary()
False
```

is_totally_real()

Return True if self is totally real, and False otherwise.

Totally real means that every isomorphic embedding of self into the complex numbers has image contained in the real numbers.

EXAMPLES:

```
sage: NumberField(x^2+2, 'alpha').is_totally_real()
False
sage: NumberField(x^2-2, 'alpha').is_totally_real()
True
sage: NumberField(x^4-2, 'alpha').is_totally_real()
False
```

latex_variable_name(*name=None*)

Return the latex representation of the variable name for this number field.

EXAMPLES:

```
sage: NumberField(x^2 + 3, 'a').latex_variable_name()
'a'
sage: NumberField(x^3 + 3, 'theta3').latex_variable_name()
'\theta_{3}'
sage: CyclotomicField(5).latex_variable_name()
'\zeta_{5}'
```

narrow_class_group(*proof=None*)

Return the narrow class group of this field.

INPUT:

- `proof` - default: None (use the global proof setting, which defaults to True).

EXAMPLES:

```
sage: NumberField(x^3+x+9, 'a').narrow_class_group()
Multiplicative Abelian Group isomorphic to C2
```

ngens()

Return the number of generators of this number field (always 1).

OUTPUT: the python integer 1.

EXAMPLES:

```
sage: NumberField(x^2 + 17,'a').ngens()
1
sage: NumberField(x + 3,'a').ngens()
1
sage: k.<a> = NumberField(x + 3)
sage: k.ngens()
1
sage: k.0
-3
```

number_of_roots_of_unity()

Return the number of roots of unity in this field.

Note: We do not create the full unit group since that can be expensive, but we do use it if it is already known.

EXAMPLES:

```
sage: F.<alpha> = NumberField(x**22+3)
sage: F.zeta_order()
6
sage: F.<alpha> = NumberField(x**2-7)
sage: F.zeta_order()
2
```

order()

Return the order of this number field (always +infinity).

OUTPUT: always positive infinity

EXAMPLES:

```
sage: NumberField(x^2 + 19,'a').order()
+Infinity
```

pari_bnf(*certify=False, units=True*)

PARI big number field corresponding to this field.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 1); k
Number Field in a with defining polynomial x^2 + 1
sage: len(k.pari_bnf())
10
sage: k.pari_bnf()[:4]
[[;], matrix(0,7), [;], ...]
sage: len(k.pari_nf())
9
```

pari_bnf_certify()

Run the PARI bnfcertify function to ensure the correctness of answers.

If this function returns True (and doesn't raise a ValueError), then certification succeeded, and results that use the PARI bnf structure with this field are supposed to be correct.

Warning: I wouldn't trust this to mean that everything computed involving this number field is actually correct.

EXAMPLES:

```
sage: k.<a> = NumberField(x^7 + 7); k
Number Field in a with defining polynomial x^7 + 7
sage: k.pari_bnf_certify()
True
```

pari_nf()

PARI number field corresponding to this field.

This is the number field constructed using nfinit. This is the same as the number field got by doing pari(self) or gp(self).

EXAMPLES:

```
sage: k.<a> = NumberField(x^4 - 3*x + 7); k
Number Field in a with defining polynomial x^4 - 3*x + 7
sage: k.pari_nf()[:4]
[x^4 - 3*x + 7, [0, 2], 85621, 1]
sage: pari(k)[:4]
[x^4 - 3*x + 7, [0, 2], 85621, 1]

sage: k.<a> = NumberField(x^4 - 3/2*x + 5/3); k
Number Field in a with defining polynomial x^4 - 3/2*x + 5/3
sage: k.pari_nf()
...
TypeError: Unable to coerce number field defined by non-integral polynomial to PARI.
sage: pari(k)
...
TypeError: Unable to coerce number field defined by non-integral polynomial to PARI.
sage: gp(k)
...
TypeError: Unable to coerce number field defined by non-integral polynomial to PARI.
```

pari_polynomial(*name='x'*)

PARI polynomial corresponding to polynomial that defines this field. By default, this is a polynomial in the variable "x".

EXAMPLES:

```
sage: y = polygen(QQ)
sage: k.<a> = NumberField(y^2 - 3/2*y + 5/3)
sage: k.pari_polynomial()
x^2 - 3/2*x + 5/3
sage: k.pari_polynomial('a')
a^2 - 3/2*a + 5/3
```

polynomial()

Return the defining polynomial of this number field.

This is exactly the same as `self.defining_polynomal()`.

EXAMPLES:

```
sage: NumberField(x^2 + (2/3)*x - 9/17,'a').polynomial()
x^2 + 2/3*x - 9/17
```

polynomial_ntl()

Return defining polynomial of this number field as a pair, an ntl polynomial and a denominator.

This is used mainly to implement some internal arithmetic.

EXAMPLES:

```
sage: NumberField(x^2 + (2/3)*x - 9/17,'a').polynomial_ntl()
([-27 34 51], 51)
```

polynomial_quotient_ring()

Return the polynomial quotient ring isomorphic to this number field.

EXAMPLES:

```
sage: K = NumberField(x^3 + 2*x - 5, 'alpha')
sage: K.polynomial_quotient_ring()
Univariate Quotient Polynomial Ring in alpha over Rational Field with modulus x^3 + 2*x - 5
```

polynomial_ring()

Return the polynomial ring that we view this number field as being a quotient of (by a principal ideal).

EXAMPLES: An example with an absolute field:

```
sage: k.<a> = NumberField(x^2 + 3)
sage: y = polygen(QQ, 'y')
sage: k.<a> = NumberField(y^2 + 3)
sage: k.polynomial_ring()
Univariate Polynomial Ring in y over Rational Field
```

An example with a relative field:

```
sage: y = polygen(QQ, 'y')
sage: M.<a> = NumberField([y^3 + 97, y^2 + 1]); M
Number Field in a0 with defining polynomial y^3 + 97 over its base field
sage: M.polynomial_ring()
Univariate Polynomial Ring in y over Number Field in a1 with defining polynomial y^2 + 1
```

power_basis()

Return a power basis for this number field over its base field.

If this number field is represented as $k[t]/f(t)$, then the basis returned is $1, t, t^2, \ldots, t^{d-1}$ where d is the degree of this number field over its base field.

EXAMPLES:

```
sage: K.<a> = NumberField(x^5 + 10*x + 1)
sage: K.power_basis()
[1, a, a^2, a^3, a^4]

sage: L.<b> = K.extension(x^2 - 2)
sage: L.power_basis()
[1, b]
sage: L.absolute_field('c').power_basis()
[1, c, c^2, c^3, c^4, c^5, c^6, c^7, c^8, c^9]

sage: M = CyclotomicField(15)
sage: M.power_basis()
[1, zeta15, zeta15^2, zeta15^3, zeta15^4, zeta15^5, zeta15^6, zeta15^7]
```

prime_above(*x*, *degree=None*)

Return a prime ideal of self lying over x.

INPUT:

- `x`: usually an element or ideal of self. It should be such that self.ideal(x) is sensible. This excludes x=0.
- `degree` (default: None): None or an integer. If one, find a prime above x of any degree. If an integer, find a prime above x such that the resulting residue field has exactly this degree.

OUTPUT: A prime ideal of self lying over x. If degree is specified and no such ideal exists, raises a ValueError.

Warning: At this time we factor the ideal x, which may not be supported for relative number fields.

EXAMPLES:

```
sage: x = ZZ['x'].gen()
sage: F.<t> = NumberField(x^3 - 2)

sage: P2 = F.prime_above(2)
sage: P2 # random
Fractional ideal (-t)
sage: 2 in P2
True
sage: P2.is_prime()
True
sage: P2.norm()
2

sage: P3 = F.prime_above(3)
sage: P3 # random
Fractional ideal (t + 1)
sage: 3 in P3
True
sage: P3.is_prime()
True
sage: P3.norm()
3
```

The ideal (3) is totally ramified in F, so there is no degree 2 prime above 3:

```
sage: F.prime_above(3, degree=2)
...
ValueError: No prime of degree 2 above Fractional ideal (3)
sage: [ id.residue_class_degree() for id, _ in F.ideal(3).factor() ]
[1]
```

Asking for a specific degree works:

```
sage: P5_1 = F.prime_above(5, degree=1)
sage: P5_1 # random
Fractional ideal (-t^2 - 1)
sage: P5_1.residue_class_degree()
1

sage: P5_2 = F.prime_above(5, degree=2)
sage: P5_2 # random
Fractional ideal (t^2 - 2*t - 1)
sage: P5_2.residue_class_degree()
2
```

Relative number fields are ok:

```
sage: G = F.extension(x^2 - 11, 'b')
sage: G.prime_above(7)
Fractional ideal (7, (3*t^2 + 2*t + 9)*b - t^2 - 31*t - 10)
```

TESTS:

It doesn't make sense to factor the ideal (0):

```
sage: F.prime_above(0)
...
AttributeError: 'NumberFieldIdeal' object has no attribute 'factor'
```

primes_above (*x, degree=None*)

Return prime ideals of self lying over x.

INPUT:

- x: usually an element or ideal of self. It should be such that self.ideal(x) is sensible. This excludes x=0.
- degree (default: None): None or an integer. If None, find all primes above x of any degree. If an integer, find all primes above x such that the resulting residue field has exactly this degree.

OUTPUT: A list of prime ideals of self lying over x. If degree is specified and no such ideal exists, returns the empty list.

EXAMPLES:

```
sage: x = ZZ['x'].gen()
sage: F.<t> = NumberField(x^3 - 2)

sage: P2s = F.primes_above(2)
sage: P2s # random
[Fractional ideal (-t)]
sage: all(2 in P2 for P2 in P2s)
True
sage: all(P2.is_prime() for P2 in P2s)
True
sage: [ P2.norm() for P2 in P2s ]
[2]

sage: P3s = F.primes_above(3)
sage: P3s # random
[Fractional ideal (t + 1)]
sage: all(3 in P3 for P3 in P3s)
True
sage: all(P3.is_prime() for P3 in P3s)
True
sage: [ P3.norm() for P3 in P3s ]
[3]
```

The ideal (3) is totally ramified in F, so there is no degree 2 prime above 3:

```
sage: F.primes_above(3, degree=2)
[]
sage: [ id.residue_class_degree() for id, _ in F.ideal(3).factor() ]
[1]
```

Asking for a specific degree works:

```
sage: P5_1s = F.primes_above(5, degree=1)
sage: P5_1s # random
[Fractional ideal (-t^2 - 1)]
sage: P5_1 = P5_1s[0]; P5_1.residue_class_degree()
1

sage: P5_2s = F.primes_above(5, degree=2)
sage: P5_2s # random
[Fractional ideal (t^2 - 2*t - 1)]
sage: P5_2 = P5_2s[0]; P5_2.residue_class_degree()
2
```

Works in relative extensions too:

```
sage: PQ.<X> = QQ[]
sage: F.<a, b> = NumberField([X^2 - 2, X^2 - 3])
sage: PF.<Y> = F[]
sage: K.<c> = F.extension(Y^2 - (1 + a)*(a + b)*a*b)
sage: I = F.ideal(a + 2*b)
sage: K.primes_above(I)
[Fractional ideal (2, ((-13*b - 45/2)*a - 37/2*b - 63/2)*c + 1),
 Fractional ideal (5, (5/2*b + 7/2)*a + 2*b + 10)]
sage: K.primes_above(I, degree=1)
[Fractional ideal (2, ((-13*b - 45/2)*a - 37/2*b - 63/2)*c + 1)]
sage: K.primes_above(I, degree=4)
[Fractional ideal (5, (5/2*b + 7/2)*a + 2*b + 10)]
```

TESTS:

It doesn't make sense to factor the ideal (0):

```
sage: F.primes_above(0)
...
AttributeError: 'NumberFieldIdeal' object has no attribute 'factor'
```

primes_of_degree_one_iter(*num_integer_primes=10000, max_iterations=100*)

Return an iterator yielding prime ideals of absolute degree one and small norm.

> **Warning:** It is possible that there are no primes of K of absolute degree one of small prime norm, and it possible that this algorithm will not find any primes of small norm.
> See module `sage.rings.number_field.small_primes_of_degree_one` for details.

INPUT:

- `num_integer_primes (default: 10000)` - an integer. We try to find primes of absolute norm no greater than the num_integer_primes-th prime number. For example, if num_integer_primes is 2, the largest norm found will be 3, since the second prime is 3.
- `max_iterations (default: 100)` - an integer. We test max_iterations integers to find small primes before raising StopIteration.

EXAMPLES:

```
sage: K.<z> = CyclotomicField(10)
sage: it = K.primes_of_degree_one_iter()
sage: Ps = [ it.next() for i in range(3) ]
sage: Ps # random
[Fractional ideal (z^3 + z + 1), Fractional ideal (3*z^3 - z^2 + z - 1), Fractional ideal (2
sage: [ P.norm() for P in Ps ] # random
[11, 31, 41]
sage: [ P.residue_class_degree() for P in Ps ]
[1, 1, 1]
```

primes_of_degree_one_list(*n, num_integer_primes=10000, max_iterations=100*)

Return a list of n prime ideals of absolute degree one and small norm.

> **Warning:** It is possible that there are no primes of K of absolute degree one of small prime norm, and it possible that this algorithm will not find any primes of small norm.
> See module `sage.rings.number_field.small_primes_of_degree_one` for details.

INPUT:

- `num_integer_primes (default: 10000)` - an integer. We try to find primes of absolute norm no greater than the num_integer_primes-th prime number. For example, if num_integer_primes is 2, the largest norm found will be 3, since the second prime is 3.

- max_iterations (default: 100) - an integer. We test max_iterations integers to find small primes before raising StopIteration.

EXAMPLES:

```
sage: K.<z> = CyclotomicField(10)
sage: Ps = K.primes_of_degree_one_list(3)
sage: Ps
[Fractional ideal (z^3 + z + 1), Fractional ideal (3*z^3 - z^2 + z - 1), Fractional ideal (2
sage: [ P.norm() for P in Ps ]
[11, 31, 41]
sage: [ P.residue_class_degree() for P in Ps ]
[1, 1, 1]
```

primitive_element()

Return a primitive element for this field, i.e., an element that generates it over $\mathbf{Q}$.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3 + 2)
sage: K.primitive_element()
a
sage: K.<a,b,c> = NumberField([x^2-2,x^2-3,x^2-5])
sage: K.primitive_element()
a - b + c
sage: alpha = K.primitive_element(); alpha
a - b + c
sage: alpha.minpoly()
x^2 + (2*b - 2*c)*x - 2*c*b + 6
sage: alpha.absolute_minpoly()
x^8 - 40*x^6 + 352*x^4 - 960*x^2 + 576
```

primitive_root_of_unity()

Return a generator of the roots of unity in this field.

Note: We do not create the full unit group since that can be expensive, but we do use it if it is already known.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2+1)
sage: z = K.primitive_root_of_unity(); z
i
sage: z.multiplicative_order()
4

sage: K.<a> = NumberField(x^2+x+1)
sage: z = K.primitive_root_of_unity(); z
-a
sage: z.multiplicative_order()
6
```

random_element()

Return a random element of this number field.

EXAMPLES:

```
sage: K.<j> = NumberField(x^8+1)
sage: K.random_element()
1/2*j^7 - j^6 - 12*j^5 + 1/2*j^4 - 1/95*j^3 - 1/2*j^2 - 4

sage: K.<a,b,c> = NumberField([x^2-2,x^2-3,x^2-5])
sage: K.random_element()
((-c + 1)*b - c + 2/3)*a - 5/2*b + 2/3*c - 1/4
```

real_embeddings(*prec=53*)

Return all homomorphisms of this number field into the approximate real field with precision prec.

If prec is 53 (the default), then the real double field is used; otherwise the arbitrary precision (but slow) real field is used.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3 + 2)
sage: K.real_embeddings()
[
Ring morphism:
  From: Number Field in a with defining polynomial x^3 + 2
  To:   Real Double Field
  Defn: a |--> -1.25992104989
]
 sage: K.real_embeddings(16)
 [
 Ring morphism:
   From: Number Field in a with defining polynomial x^3 + 2
   To:   Real Field with 16 bits of precision
   Defn: a |--> -1.260
 ]
sage: K.real_embeddings(100)
[
Ring morphism:
  From: Number Field in a with defining polynomial x^3 + 2
  To:   Real Field with 100 bits of precision
  Defn: a |--> -1.2599210498948731647672106073
]
```

reduced_basis(*prec=None*)

This function returns an LLL-reduced basis for the Minkowski-embedding of the maximal order of a number field.

INPUT:

- `self` - number field, the base field
- `prec (default: None)` - the precision with which to compute the Minkowski embedding. (See NOTE below.)

OUTPUT:

An LLL-reduced basis for the Minkowski-embedding of the maximal order of a number field, given by a sequence of (integral) elements from the field.

Note: In the non-totally-real case, the LLL routine we call is currently Pari's qflll(), which works with floating point approximations, and so the result is only as good as the precision promised by Pari. The matrix returned will always be integral; however, it may only be only "almost" LLL-reduced when the precision is not sufficiently high.

If the following run-time error occurs: "PariError: not a definite matrix in lllgram (42)" try increasing the prec parameter,

EXAMPLES:

```
sage: F.<t> = NumberField(x^6-7*x^4-x^3+11*x^2+x-1)
sage: F.maximal_order().basis()
[1/2*t^5 + 1/2*t^4 + 1/2*t^2 + 1/2, t, t^2, t^3, t^4, t^5]
sage: F.reduced_basis()
[-1, -1/2*t^5 + 1/2*t^4 + 3*t^3 - 3/2*t^2 - 4*t - 1/2, t, 1/2*t^5 + 1/2*t^4 - 4*t^3 - 5/2*t^

sage: F.<alpha> = NumberField(x^4+x^2+712312*x+131001238)
sage: F.integral_basis()
[1, alpha, 1/2*alpha^3 + 1/2*alpha^2, alpha^3]
```

```
sage: F.reduced_basis(prec=64)
[1, alpha, alpha^3 - 2*alpha^2 + 15*alpha, 16*alpha^3 - 31*alpha^2 + 469*alpha + 267109]
...                                                    # 64-bit
PariError: not a definite matrix in lllgram (42)   # 64-bit
sage: F.reduced_basis(prec=96)
[1, alpha, alpha^3 - 2*alpha^2 + 15*alpha, 16*alpha^3 - 31*alpha^2 + 469*alpha + 267109]
```

reduced_gram_matrix(*prec=None*)

This function returns the Gram matrix of an LLL-reduced basis for the Minkowski embedding of the maximal order of a number field.

INPUT:

- `self` - number field, the base field
- `prec (default: None)` - the precision with which to calculate the Minkowski embedding. (See NOTE below.)

OUTPUT: The Gram matrix $[< x_i, x_j >]$ of an LLL reduced basis for the maximal order of self, where the integral basis for self is given by $\{x_0, \ldots, x_{n-1}\}$. Here < , > is the usual inner product on $\mathbf{R}^n$, and self is embedded in $\mathbf{R}^n$ by the Minkowski embedding. See the docstring for `Minkowski_embedding()` for more information.

Note: In the non-totally-real case, the LLL routine we call is currently Pari's qflll(), which works with floating point approximations, and so the result is only as good as the precision promised by Pari. In particular, in this case, the returned matrix will *not* be integral, and may not have enough precision to recover the correct gram matrix (which is known to be integral for theoretical reasons). Thus the need for the prec flag above.

If the following run-time error occurs: "PariError: not a definite matrix in lllgram (42)" try increasing the prec parameter,

EXAMPLES:

```
sage: F.<t> = NumberField(x^6-7*x^4-x^3+11*x^2+x-1)
sage: F.reduced_gram_matrix()
[ 6  3  0  2  0  1]
[ 3  9  0  1  0 -2]
[ 0  0 14  6 -2  3]
[ 2  1  6 16 -3  3]
[ 0  0 -2 -3 16  6]
[ 1 -2  3  3  6 19]
sage: Matrix(6, [(x*y).trace() for x in F.integral_basis() for y in F.integral_basis()])
[2550  133  259  664 1368 3421]
[ 133   14    3   54   30  233]
[ 259    3   54   30  233  217]
[ 664   54   30  233  217 1078]
[1368   30  233  217 1078 1371]
[3421  233  217 1078 1371 5224]

sage: var('x')
x
sage: F.<alpha> = NumberField(x^4+x^2+712312*x+131001238)
sage: F.reduced_gram_matrix(prec=128)
[   4.0000000000000000000000000000000000000   0.0000000000000000000000000000000000000 -2.13
[  0.0000000000000000000000000000000000000    46721.539331563218381658483353092335550 -2.24
[-2.1369320000000000000000000000000000000e6 -2.2467769057394530109094755223395819322e7 7.070
[-3.3122478000000000000000000000000000000e7 -3.4807276041138450473611629088647496430e8 1.125
```

regulator(*proof=None*)

Return the regulator of this number field.

Note that PARI computes the regulator to higher precision than the Sage default.

INPUT:

- `proof` - default: True, unless you set it otherwise.

EXAMPLES:

```
sage: NumberField(x^2-2, 'a').regulator()
0.881373587019543
sage: NumberField(x^4+x^3+x^2+x+1, 'a').regulator()
0.962423650119207
```

residue_field(*prime, names=None, check=True*)

Return the residue field of this number field at a given prime, ie O_K/pO_K.

INPUT:

- `prime` - a prime ideal of the maximal order in this number field, or an element of the field which generates a principal prime ideal.
- `names` - the name of the variable in the residue field
- `check` - whether or not to check the primality of prime.

OUTPUT: The residue field at this prime.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^4+3*x^2-17)
sage: P = K.ideal(61).factor()[0][0]
sage: K.residue_field(P)
Residue field in abar of Fractional ideal (-2*a^2 + 1)

sage: K.<i> = NumberField(x^2 + 1)
sage: K.residue_field(1+i)
Residue field of Fractional ideal (i + 1)
```

TESTS:

```
sage: L.<b> = NumberField(x^2 + 5)
sage: L.residue_field(P)
...
ValueError: Fractional ideal (-2*a^2 + 1) is not an ideal of Number Field in b with defining
sage: L.residue_field(2)
...
ValueError: Fractional ideal (2) is not a prime ideal

sage: L.residue_field(L.prime_above(5)^2)
...
ValueError: Fractional ideal (5) is not a prime ideal
```

roots_of_unity()

Return all the roots of unity in this field, primitive or not.

EXAMPLES:

```
sage: K.<b> = NumberField(x^2+1)
sage: zs = K.roots_of_unity(); zs
[b, -1, -b, 1]
sage: [ z**K.number_of_roots_of_unity() for z in zs ]
[1, 1, 1, 1]
```

signature()

Return (r1, r2), where r1 and r2 are the number of real embeddings and pairs of complex embeddings of this field, respectively.

EXAMPLES:

```
sage: NumberField(x^2+1, 'a').signature()
(0, 1)
sage: NumberField(x^3-2, 'a').signature()
(1, 1)
```

specified_complex_embedding()

Returns the embedding of this field into the complex numbers has been specified.

Fields created with the QuadraticField or CyclotomicField constructors come with an implicit embedding. To get one of these fields without the embedding, use the generic NumberField constructor.

EXAMPLES:

```
sage: QuadraticField(-1, 'I').specified_complex_embedding()
Generic morphism:
  From: Number Field in I with defining polynomial x^2 + 1
  To:   Complex Lazy Field
  Defn: I -> 1*I

sage: QuadraticField(3, 'a').specified_complex_embedding()
Generic morphism:
  From: Number Field in a with defining polynomial x^2 - 3
  To:   Real Lazy Field
  Defn: a -> 1.732050807568878?

sage: CyclotomicField(13).specified_complex_embedding()
Generic morphism:
  From: Cyclotomic Field of order 13 and degree 12
  To:   Complex Lazy Field
  Defn: zeta13 -> 0.885456025653210? + 0.464723172043769?*I
```

Most fields don't implicitly have embeddings unless explicitly specified:

```
sage: NumberField(x^2-2, 'a').specified_complex_embedding() is None
True
sage: NumberField(x^3-x+5, 'a').specified_complex_embedding() is None
True
sage: NumberField(x^3-x+5, 'a', embedding=2).specified_complex_embedding()
Generic morphism:
  From: Number Field in a with defining polynomial x^3 - x + 5
  To:   Real Lazy Field
  Defn: a -> -1.904160859134921?
sage: NumberField(x^3-x+5, 'a', embedding=CDF.0).specified_complex_embedding()
Generic morphism:
  From: Number Field in a with defining polynomial x^3 - x + 5
  To:   Complex Lazy Field
  Defn: a -> 0.952080429567461? + 1.311248044077123?*I
```

This function only returns complex embeddings:

```
sage: K.<a> = NumberField(x^2-2, embedding=Qp(7)(2).sqrt())
sage: K.specified_complex_embedding() is None
True
sage: K.gen_embedding()
3 + 7 + 2*7^2 + 6*7^3 + 7^4 + 2*7^5 + 7^6 + 2*7^7 + 4*7^8 + 6*7^9 + 6*7^10 + 2*7^11 + 7^12 +
sage: K.coerce_embedding()
Generic morphism:
  From: Number Field in a with defining polynomial x^2 - 2
  To:   7-adic Field with capped relative precision 20
  Defn: a -> 3 + 7 + 2*7^2 + 6*7^3 + 7^4 + 2*7^5 + 7^6 + 2*7^7 + 4*7^8 + 6*7^9 + 6*7^10 + 2*
```

structure()

Return fixed isomorphism or embedding structure on self.

This is used to record various isomorphisms or embeddings that arise naturally in other constructions.

EXAMPLES:

```
sage: K.<z> = NumberField(x^2 + 3)
sage: L.<a> = K.absolute_field(); L
Number Field in a with defining polynomial x^2 + 3
sage: L.structure()
(Isomorphism given by variable name change map:
  From: Number Field in a with defining polynomial x^2 + 3
  To:   Number Field in z with defining polynomial x^2 + 3,
 Isomorphism given by variable name change map:
  From: Number Field in z with defining polynomial x^2 + 3
  To:   Number Field in a with defining polynomial x^2 + 3)
```

subfield(*alpha, name=None, names=None*)

Return an absolute number field K isomorphic to QQ(alpha) and a map from K to self that sends the generator of K to alpha.

INPUT:

- `alpha` - an element of self, or something that coerces to an element of self.

OUTPUT:

- `K` - a number field
- `from_K` - a homomorphism from K to self that sends the generator of K to alpha.

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 - 3); K
Number Field in a with defining polynomial x^4 - 3
sage: H.<b>, from_H = K.subfield(a^2)
sage: H
Number Field in b with defining polynomial x^2 - 3
sage: from_H(b)
a^2
sage: from_H
Ring morphism:
  From: Number Field in b with defining polynomial x^2 - 3
  To:   Number Field in a with defining polynomial x^4 - 3
  Defn: b |--> a^2
```

```
sage: K.<z> = CyclotomicField(5)
sage: K.subfield(z-z^2-z^3+z^4)
(Number Field in z0 with defining polynomial x^2 - 5,
Ring morphism:
From: Number Field in z0 with defining polynomial x^2 - 5
To:   Cyclotomic Field of order 5 and degree 4
Defn: z0 |--> -2*z^3 - 2*z^2 - 1)
```

You can also view a number field as having a different generator by just choosing the input to generate the whole field; for that it is better to use `self.change_generator`, which gives isomorphisms in both directions.

trace_pairing(*v*)

Return the matrix of the trace pairing on the elements of the list v.

EXAMPLES:

```
sage: K.<zeta3> = NumberField(x^2 + 3)
sage: K.trace_pairing([1,zeta3])
[ 2  0]
[ 0 -6]
```

uniformizer(*P, others='positive'*)

Returns an element of self with valuation 1 at the prime ideal P.

INPUT:

- `self` - a number field
- `P` - a prime ideal of self
- `others` - either "positive" (default), in which case the element will have non-negative valuation at all other primes of self, or "negative", in which case the element will have non-positive valuation at all other primes of self.

Note: When P is principal (e.g. always when self has class number one) the result may or may not be a generator of P!

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 5); K
Number Field in a with defining polynomial x^2 + 5
sage: P,Q = K.ideal(3).prime_factors()
sage: P
Fractional ideal (3, a + 1)
sage: pi=K.uniformizer(P); pi
a + 1
sage: K.ideal(pi).factor()
(Fractional ideal (2, a + 1)) * (Fractional ideal (3, a + 1))
sage: pi=K.uniformizer(P,'negative'); pi
1/2*a + 1/2
sage: K.ideal(pi).factor()
(Fractional ideal (2, a + 1))^-1 * (Fractional ideal (3, a + 1))

sage: K = CyclotomicField(9)
sage: Plist=K.ideal(17).prime_factors()
sage: pilist = [K.uniformizer(P) for P in Plist]
sage: [pi.is_integral() for pi in pilist]
[True, True, True]
sage: [pi.valuation(P) for pi,P in zip(pilist,Plist)]
[1, 1, 1]
sage: [ pilist[i] in Plist[i] for i in range(len(Plist)) ]
[True, True, True]
```

unit_group(*proof=None*)

Return the unit group (including torsion) of this number field.

ALGORITHM: Uses PARI's bnfunit command.

INPUTS: proof – default: True

NOTE: the group is cached.

EXAMPLES:

```
sage: x = QQ['x'].0
sage: A = x^4 - 10*x^3 + 20*5*x^2 - 15*5^2*x + 11*5^3
sage: K = NumberField(A, 'a')
sage: U = K.unit_group(); U
Unit group with structure C10 x Z of Number Field in a with defining polynomial x^4 - 10*x^3
sage: U.gens()
[-1/275*a^3 + 7/55*a^2 - 6/11*a + 4, 8/275*a^3 - 12/55*a^2 + 15/11*a - 2]
sage: U.invariants()
[10, 0]
sage: [u.multiplicative_order() for u in U.gens()]
[10, +Infinity]
```

Sage might not be able to provably compute the unit group:

```
sage: K = NumberField(x^17 + 3, 'a')
sage: K.unit_group(proof=True) # default
...
PariError: not enough precomputed primes, need primelimit ~  (35)
```

In this case, one can ask for the conjectural unit group (correct if the Generalized Riemann Hypothesis is true):

```
sage: U = K.unit_group(proof=False); U; U.gens()
Unit group with structure C2 x Z x Z x Z x Z x Z x Z x Z x Z of Number Field in a with defin
[-1,
a^9 + a - 1,
a^16 - a^15 + a^14 - a^12 + a^11 - a^10 - a^8 + a^7 - 2*a^6 + a^4 - 3*a^3 + 2*a^2 - 2*a + 1,
2*a^16 - a^14 - a^13 + 3*a^12 - 2*a^10 + a^9 + 3*a^8 - 3*a^6 + 3*a^5 + 3*a^4 - 2*a^3 - 2*a^2
2*a^16 - 3*a^15 + 3*a^14 - 3*a^13 + 3*a^12 - a^11 + a^9 - 3*a^8 + 4*a^7 - 5*a^6 + 6*a^5 - 4*
a^15 - a^12 + a^10 - a^9 - 2*a^8 + 3*a^7 + a^6 - 3*a^5 + a^4 + 4*a^3 - 3*a^2 - 2*a + 2,
a^16 - a^15 - 3*a^14 - 4*a^13 - 4*a^12 - 3*a^11 - a^10 + 2*a^9 + 4*a^8 + 5*a^7 + 4*a^6 + 2*a
a^15 + a^14 + 2*a^11 + a^10 - a^9 + a^8 + 2*a^7 - a^5 + 2*a^3 - a^2 - 3*a + 1,
3*a^16 + 3*a^15 + 3*a^14 + 3*a^13 + 3*a^12 + 2*a^11 + 2*a^10 + 2*a^9 + a^8 - a^7 - 2*a^6 - 3
```

The provable and the conjectural results are cached separately (this fixes trac #2504):

```
sage: K.units(proof=True)
...
PariError: not enough precomputed primes, need primelimit ~  (35)
```

units (*proof=None*)

Return generators for the unit group modulo torsion.

ALGORITHM: Uses PARI's bnfunit command.

INPUTS: proof - default: True

NOTE: For more functionality see the unit_group() function.

EXAMPLES:

```
sage: x = QQ['x'].0
sage: A = x^4 - 10*x^3 + 20*5*x^2 - 15*5^2*x + 11*5^3
sage: K = NumberField(A, 'a')
sage: K.units()
[8/275*a^3 - 12/55*a^2 + 15/11*a - 2]
```

Sage might not be able to provably compute the unit group:

```
sage: K = NumberField(x^17 + 3, 'a')
sage: K.units(proof=True) # default
...
PariError: not enough precomputed primes, need primelimit ~  (35)
```

In this case, one can ask for the conjectural unit group (correct if the Generalized Riemann Hypothesis is true):

```
sage: K.units(proof=False)
[a^9 + a - 1,
a^16 - a^15 + a^14 - a^12 + a^11 - a^10 - a^8 + a^7 - 2*a^6 + a^4 - 3*a^3 + 2*a^2 - 2*a + 1,
2*a^16 - a^14 - a^13 + 3*a^12 - 2*a^10 + a^9 + 3*a^8 - 3*a^6 + 3*a^5 + 3*a^4 - 2*a^3 - 2*a^2
2*a^16 - 3*a^15 + 3*a^14 - 3*a^13 + 3*a^12 - a^11 + a^9 - 3*a^8 + 4*a^7 - 5*a^6 + 6*a^5 - 4*
a^15 - a^12 + a^10 - a^9 - 2*a^8 + 3*a^7 + a^6 - 3*a^5 + a^4 + 4*a^3 - 3*a^2 - 2*a + 2,
a^16 - a^15 - 3*a^14 - 4*a^13 - 4*a^12 - 3*a^11 - a^10 + 2*a^9 + 4*a^8 + 5*a^7 + 4*a^6 + 2*a
a^15 + a^14 + 2*a^11 + a^10 - a^9 + a^8 + 2*a^7 - a^5 + 2*a^3 - a^2 - 3*a + 1,
3*a^16 + 3*a^15 + 3*a^14 + 3*a^13 + 3*a^12 + 2*a^11 + 2*a^10 + 2*a^9 + a^8 - a^7 - 2*a^6 - 3
```

The provable and the conjectural results are cached separately (this fixes trac #2504):

```
sage: K.units(proof=True)
...
PariError: not enough precomputed primes, need primelimit ~  (35)
```

zeta(*n=2, all=False*)

Return one, or a list of all, primitive n-th root of unity in this field.

INPUT:

- n - positive integer
- all - bool. If False (default), return a primitive n-th root of unity in this field, or raise an ArithmeticError exception if there are none. If True, return a list of all primitive n-th roots of unity in this field (possibly empty).

Note: To obtain the maximal order of a root of unity in this field, use self.number_of_roots_of_unity().

Note: We do not create the full unit group since that can be expensive, but we do use it if it is already known.

EXAMPLES:

```
sage: K.<z> = NumberField(x^2 + 3)
sage: K.zeta(1)
1
sage: K.zeta(2)
-1
sage: K.zeta(2, all=True)
[-1]
sage: K.zeta(3)
1/2*z - 1/2
sage: K.zeta(3, all=True)
[1/2*z - 1/2, -1/2*z - 1/2]
sage: K.zeta(4)
...
ValueError: n (=4) does not divide order of generator

sage: r.<x> = QQ[]
sage: K.<b> = NumberField(x^2+1)
sage: K.zeta(4)
b
sage: K.zeta(4,all=True)
[b, -b]
sage: K.zeta(3)
...
ValueError: n (=3) does not divide order of generator
sage: K.zeta(3,all=True)
[]
```

zeta_coefficients(*n*)

Compute the first n coefficients of the Dedekind zeta function of this field as a Dirichlet series.

EXAMPLE:

```
sage: x = QQ['x'].0
sage: NumberField(x^2+1, 'a').zeta_coefficients(10)
[1, 1, 0, 1, 2, 0, 0, 1, 1, 2]
```

zeta_function(*prec=53, max_imaginary_part=0, max_asymp_coeffs=40*)

Return the Zeta function of this number field.

This actually returns an interface to Tim Dokchitser's program for computing with the Dedekind zeta function zeta_F(s) of the number field F.

INPUT:

- `prec` - integer (bits precision)
- `max_imaginary_part` - real number
- `max_asymp_coeffs` - integer

OUTPUT: The zeta function of this number field.

EXAMPLES:

```
sage: K.<a> = NumberField(ZZ['x'].0^2+ZZ['x'].0-1)
sage: Z = K.zeta_function()
sage: Z
Zeta function associated to Number Field in a with defining polynomial x^2 + x - 1
sage: Z(-1)
0.0333333333333333
```

zeta_order()

Return the number of roots of unity in this field.

Note: We do not create the full unit group since that can be expensive, but we do use it if it is already known.

EXAMPLES:

```
sage: F.<alpha> = NumberField(x**22+3)
sage: F.zeta_order()
6
sage: F.<alpha> = NumberField(x**2-7)
sage: F.zeta_order()
2
```

NumberField_generic_v1(*poly, name, latex_name, canonical_embedding=None*)

This is used in pickling generic number fields.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field import NumberField_generic_v1
sage: R.<x> = QQ[]
sage: NumberField_generic_v1(x^2 + 1, 'i', 'i')
Number Field in i with defining polynomial x^2 + 1
```

class **NumberField_quadratic**(*polynomial, name=None, check=True, embedding=None*)

Create a quadratic extension of the rational field.

The command QuadraticField(a) creates the field Q(sqrt(a)).

EXAMPLES:

```
sage: QuadraticField(3, 'a')
Number Field in a with defining polynomial x^2 - 3
sage: QuadraticField(-4, 'b')
Number Field in b with defining polynomial x^2 + 4
```

class_number(*proof=None*)

Return the size of the class group of self.

If proof = False (*not* the default!) and the discriminant of the field is negative, then the following warning from the PARI manual applies:

> IMPORTANT WARNING: For $D < 0$, this function may give incorrect results when the class group has a low exponent (has many cyclic factors), because implementing Shank's method in full generality slows it down immensely.

EXAMPLES:

```
sage: QuadraticField(-23,'a').class_number()
3
```

These are all the primes so that the class number of $\mathbf{Q}(\sqrt{-p})$ is 1:

```
sage: [d for d in prime_range(2,300) if not is_square(d) and QuadraticField(-d,'a').class_nu
[2, 3, 7, 11, 19, 43, 67, 163]
```

It is an open problem to *prove* that there are infinity many positive square-free d such that $\mathbf{Q}(\sqrt{d})$ has class number 1:n

```
sage: len([d for d in range(2,200) if not is_square(d) and QuadraticField(d,'a').class_numbe
121
```

TESTS:

```
sage: type(QuadraticField(-23,'a').class_number())
<type 'sage.rings.integer.Integer'>
sage: type(NumberField(x^3 + 23, 'a').class_number())
<type 'sage.rings.integer.Integer'>
sage: type(NumberField(x^3 + 23, 'a').extension(x^2 + 5, 'b').class_number())
<type 'sage.rings.integer.Integer'>
sage: type(CyclotomicField(10).class_number())
<type 'sage.rings.integer.Integer'>
```

discriminant(*v=None*)

Returns the discriminant of the ring of integers of the number field, or if v is specified, the determinant of the trace pairing on the elements of the list v.

INPUT:

- `v (optional)` - list of element of this number field

OUTPUT: Integer if v is omitted, and Rational otherwise.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2+1)
sage: K.discriminant()
-4
sage: K.<a> = NumberField(x^2+5)
sage: K.discriminant()
-20
sage: K.<a> = NumberField(x^2-5)
sage: K.discriminant()
5
```

hilbert_class_field(*names*)

Returns the Hilbert class field of this quadratic field as a relative extension of this field.

Note: For the polynomial that defines this field as a relative extension, see the `hilbert_class_field_defining_polynomial` command, which is vastly faster than this command, since it doesn't construct a relative extension.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 23)
sage: L = K.hilbert_class_field('b'); L
Number Field in b with defining polynomial x^3 + x^2 - 1 over its base field
sage: L.absolute_field('c')
Number Field in c with defining polynomial x^6 + 2*x^5 + 70*x^4 + 90*x^3 + 1631*x^2 + 1196*x
sage: K.hilbert_class_field_defining_polynomial()
x^3 + x^2 - 1
```

hilbert_class_field_defining_polynomial(*name='x'*)

Returns a polynomial over $\mathbf{Q}$ whose roots generate Hilbert class field of this quadratic field as an extension of this quadratic field.

Note: Computed using PARI via Schertz's method. This implementation is quite fast.

EXAMPLES:

```
sage: K.<b> = QuadraticField(-23)
sage: K.hilbert_class_field_defining_polynomial()
x^3 + x^2 - 1
```

Note that this polynomial is not the actual Hilbert class polynomial.

```
sage: magma(K.discriminant()).HilbertClassPolynomial()       # optional - magma
$.1^3 + 3491750*$.1^2 - 5151296875*$.1 + 12771880859375

sage: K.<a> = QuadraticField(-431)
sage: K.class_number()
21
sage: K.hilbert_class_field_defining_polynomial(name='z')
z^21 + z^20 - 13*z^19 - 50*z^18 + 592*z^17 - 2403*z^16 + 5969*z^15 - 10327*z^14 + 13253*z^13
```

hilbert_class_polynomial(*name='x'*)

Compute the Hilbert class polynomial of this quadratic field.

Right now, this is only implemented for imaginary quadratic fields.

EXAMPLES:

```
sage: K.<a> = QuadraticField(-3)
sage: K.hilbert_class_polynomial()
x

sage: K.<a> = QuadraticField(-31)
sage: K.hilbert_class_polynomial(name='z')
z^3 + 39491307*z^2 - 58682638134*z + 1566028350940383
```

is_galois()

Return True since all quadratic fields are automatically Galois.

EXAMPLES:

```
sage: QuadraticField(1234,'d').is_galois()
True
```

NumberField_quadratic_v1(*poly, name, canonical_embedding=None*)

This is used in pickling quadratic fields.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field import NumberField_quadratic_v1
sage: R.<x> = QQ[]
sage: NumberField_quadratic_v1(x^2 - 2, 'd')
Number Field in d with defining polynomial x^2 - 2
```

QuadraticField(*D, names, check=True, embedding=True*)

Return a quadratic field obtained by adjoining a square root of D to the rational numbers, where D is not a perfect square.

INPUT:

- `D` - a rational number
- `names` - variable name
- `check` - bool (default: True)

- `embedding` - bool or square root of D in an ambient field (default: True)

OUTPUT: A number field defined by a quadratic polynomial. Unless otherwise specified, it has an embedding into $\mathbf{R}$ or $\mathbf{C}$ by sending the generator to the positive root.

EXAMPLES:

```
sage: QuadraticField(3, 'a')
Number Field in a with defining polynomial x^2 - 3
sage: K.<theta> = QuadraticField(3); K
Number Field in theta with defining polynomial x^2 - 3
sage: RR(theta)
1.73205080756888
sage: QuadraticField(9, 'a')
...
ValueError: D must not be a perfect square.
sage: QuadraticField(9, 'a', check=False)
Number Field in a with defining polynomial x^2 - 9
```

Quadratic number fields derive from general number fields.

```
sage: from sage.rings.number_field.number_field import is_NumberField
sage: type(K)
<class 'sage.rings.number_field.number_field.NumberField_quadratic'>
sage: is_NumberField(K)
True
```

Quadratic number fields are cached:

```
sage: QuadraticField(-11, 'a') is QuadraticField(-11, 'a')
True
```

gp()

Return the unique copy of the gp (PARI) interpreter used for number field computations.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field import gp
sage: gp()
GP/PARI interpreter
```

is_AbsoluteNumberField(*x*)

Return True if x is an absolute number field.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field import is_AbsoluteNumberField
sage: is_AbsoluteNumberField(NumberField(x^2+1,'a'))
True
sage: is_AbsoluteNumberField(NumberField([x^3 + 17, x^2+1],'a'))
False
```

The rationals are a number field, but they're not of the absolute number field class.

```
sage: is_AbsoluteNumberField(QQ)
False
```

is_CyclotomicField(*x*)

Return True if x is a cyclotomic field, i.e., of the special cyclotomic field class. This function does not return True for a number field that just happens to be isomorphic to a cyclotomic field.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field import is_CyclotomicField
sage: is_CyclotomicField(NumberField(x^2 + 1,'zeta4'))
False
sage: is_CyclotomicField(CyclotomicField(4))
True
sage: is_CyclotomicField(CyclotomicField(1))
True
sage: is_CyclotomicField(QQ)
False
sage: is_CyclotomicField(7)
False
```

is_QuadraticField(*x*)

Return True if x is of the quadratic *number* field type.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field import is_QuadraticField
sage: is_QuadraticField(QuadraticField(5,'a'))
True
sage: is_QuadraticField(NumberField(x^2 - 5, 'b'))
True
sage: is_QuadraticField(NumberField(x^3 - 5, 'b'))
False
```

A quadratic field specially refers to a number field, not a finite field:

```
sage: is_QuadraticField(GF(9,'a'))
False
```

is_fundamental_discriminant(*D*)

Return True if the integer D is a fundamental discriminant, i.e., if $D \cong 0, 1 \pmod 4$, and $D \neq 0, 1$ and either (1) D is square free or (2) we have $D \cong 0 \pmod 4$ with $D/4 \cong 2, 3 \pmod 4$ and $D/4$ square free. These are exactly the discriminants of quadratic fields.

EXAMPLES:

```
sage: [D for D in range(-15,15) if is_fundamental_discriminant(D)]
[-15, -11, -8, -7, -4, -3, 5, 8, 12, 13]
sage: [D for D in range(-15,15) if not is_square(D) and QuadraticField(D,'a').disc() == D]
[-15, -11, -8, -7, -4, -3, 5, 8, 12, 13]
```

proof_flag(*t*)

Used for easily determining the correct proof flag to use.

Returns t if t is not None, otherwise returns the system-wide proof-flag for number fields (default: True).

EXAMPLES:

```
sage: from sage.rings.number_field.number_field import proof_flag
sage: proof_flag(True)
True
sage: proof_flag(False)
False
sage: proof_flag(None)
```

```
True
sage: proof_flag("banana")
'banana'
```

put_natural_embedding_first(*v*)

Helper function for embeddings() functions for number fields.

INPUT: a list of embeddings of a number field

OUTPUT: None. The list is altered in-place, so that, if possible, the first embedding has been switched with one of the others, so that if there is an embedding which preserves the generator names then it appears first.

EXAMPLES:

```
sage: K.<a> = CyclotomicField(7)
sage: embs = K.embeddings(K)
sage: [e(a) for e in embs] # random - there is no natural sort order
[a, a^2, a^3, a^4, a^5, -a^5 - a^4 - a^3 - a^2 - a - 1]
sage: id = [ e for e in embs if e(a) == a ][0]; id
Ring endomorphism of Cyclotomic Field of order 7 and degree 6
  Defn: a |--> a
sage: permuted_embs = list(embs); permuted_embs.remove(id); permuted_embs.append(id)
sage: [e(a) for e in permuted_embs] # random - but natural map is not first
[a^2, a^3, a^4, a^5, -a^5 - a^4 - a^3 - a^2 - a - 1, a]
sage: permuted_embs[0] != a
True
sage: from sage.rings.number_field.number_field import put_natural_embedding_first
sage: put_natural_embedding_first(permuted_embs)
sage: [e(a) for e in permuted_embs] # random - but natural map is first
[a, a^3, a^4, a^5, -a^5 - a^4 - a^3 - a^2 - a - 1, a^2]
sage: permuted_embs[0] == id
True
```

refine_embedding(*e, prec=None*)

Given an embedding e: K->RR or CC, returns an equivalent embedding with higher precision.

INPUT:

- `e` - an embedding of a number field into either RR or CC (with some precision)
- **`prec` - (default None) the desired precision; if None,** current precision is doubled; if Infinity, the equivalent embedding into either `QQbar` or `AA` is returned.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field import refine_embedding
sage: K = CyclotomicField(3)
sage: e10 = K.complex_embedding(10)
sage: e10.codomain().precision()
10
sage: e25 = refine_embedding(e10, prec=25)
sage: e25.codomain().precision()
25
```

An example where we extend a real enmbedding into `AA`:

```
sage: K.<a> = NumberField(x^3-2)
sage: K.signature()
(1, 1)
sage: e = K.embeddings(RR)[0]; e
Ring morphism:
```

```
From: Number Field in a with defining polynomial x^3 - 2
To:   Real Field with 53 bits of precision
Defn: a |--> 1.25992104989487
sage: e = refine_embedding(e,Infinity); e
Ring morphism:
From: Number Field in a with defining polynomial x^3 - 2
To:   Algebraic Real Field
Defn: a |--> 1.259921049894873?
```

Now we can obtain abitrary precision values with no trouble:

```
sage: RealField(150)(e(a))
1.2599210498948731647672106072782283505702515
sage: _^3
2.0000000000000000000000000000000000000000000000
sage: RealField(200)(e(a^2-3*a+7))
4.807637902283579980450073817437623208680738933795329069562 4
```

Complex enmbeddings can be extended into `QQbar`:

```
sage: e = K.embeddings(CC)[0]; e
Ring morphism:
From: Number Field in a with defining polynomial x^3 - 2
To:   Complex Field with 53 bits of precision
Defn: a |--> -0.629960524947436 - 1.09112363597172*I
sage: e = refine_embedding(e,Infinity); e
Ring morphism:
From: Number Field in a with defining polynomial x^3 - 2
To:   Algebraic Field
Defn: a |--> -0.6299605249474365? - 1.091123635971722?*I
sage: ComplexField(200)(e(a))
-0.62996052494743658238360530363911417528512573235075399004099 - 1.0911236359717214035600726141 8
sage: e(a)^3
2
```

26.2 Number Field Elements

AUTHORS:

- William Stein: version before it got Cython'd
- Joel B. Mohler (2007-03-09): First reimplementation in Cython
- William Stein (2007-09-04): add doctests
- Robert Bradshaw (2007-09-15): specialized classes for relative and absolute elements

class **CoordinateFunction**()

alpha()

class **NumberFieldElement**()

An element of a number field.

EXAMPLES:

```
sage: k.<a> = NumberField(x^3 + x + 1)
sage: a^3
-a - 1
```

abs()

Return the absolute value of this element with respect to the ith complex embedding of parent, to the given precision.

If prec is 53 (the default), then the complex double field is used; otherwise the arbitrary precision (but slow) complex field is used.

INPUT:

- `prec` - (default: 53) integer bits of precision
- `i` - (default:) integer, which embedding to use

EXAMPLES:

```
sage: z = CyclotomicField(7).gen()
sage: abs(z)
1.0
sage: abs(z^2 + 17*z - 3)
16.06044268
sage: K.<a> = NumberField(x^3+17)
sage: abs(a)
2.57128159066
sage: a.abs(prec=100)
2.5712815906582353554531872087
sage: a.abs(prec=100,i=1)
2.5712815906582353554531872087
sage: a.abs(100, 2)
2.5712815906582353554531872087
```

Here's one where the absolute value depends on the embedding.

```
sage: K.<b> = NumberField(x^2-2)
sage: a = 1 + b
sage: a.abs(i=0)
0.414213562373
sage: a.abs(i=1)
2.41421356237
```

additive_order()

Return the additive order of this element (i.e. infinity if self != 0, 1 if self == 0)

EXAMPLES:

```
sage: K.<u> = NumberField(x^4 - 3*x^2 + 3)
sage: u.additive_order()
+Infinity
sage: K(0).additive_order()
1
sage: K.ring_of_integers().characteristic() # implicit doctest
0
```

charpoly()

complex_embedding()

Return the i-th embedding of self in the complex numbers, to the given precision.

EXAMPLES:

```
sage: k.<a> = NumberField(x^3 - 2)
sage: a.complex_embedding()
```

```
-0.629960524947 - 1.09112363597*I
sage: a.complex_embedding(10)
-0.63 - 1.1*I
sage: a.complex_embedding(100)
-0.62996052494743658238360530364 - 1.0911236359717214035600726142*I
sage: a.complex_embedding(20, 1)
-0.62996 + 1.0911*I
sage: a.complex_embedding(20, 2)
1.2599
```

complex_embeddings()

Return the images of this element in the floating point complex numbers, to the given bits of precision.

INPUT:

- prec - integer (default: 53) bits of precision

EXAMPLES:

```
sage: k.<a> = NumberField(x^3 - 2)
sage: a.complex_embeddings()
[-0.629960524947 - 1.09112363597*I, -0.629960524947 + 1.09112363597*I, 1.25992104989]
sage: a.complex_embeddings(10)
[-0.63 - 1.1*I, -0.63 + 1.1*I, 1.3]
sage: a.complex_embeddings(100)
[-0.62996052494743658238360530364 - 1.0911236359717214035600726142*I, -0.629960524947436582
```

conjugate()

Return the complex conjugate of the number field element. Currently, this is implemented for cyclotomic fields and quadratic extensions of Q. It seems likely that there are other number fields for which the idea of a conjugate would be easy to compute.

EXAMPLES:

```
sage: k.<I> = QuadraticField(-1)
sage: I.conjugate()
-I
sage: (I/(1+I)).conjugate()
-1/2*I + 1/2
sage: z6=CyclotomicField(6).gen(0)
sage: (2*z6).conjugate()
-2*zeta6 + 2

sage: K.<b> = NumberField(x^3 - 2)
sage: b.conjugate()
...
NotImplementedError: complex conjugation is not implemented (or doesn't make sense).
```

coordinates_in_terms_of_powers()

Let α be self. Return a Python function that takes any element of the parent of self in $\mathbf{Q}(\alpha)$ and writes it in terms of the powers of α: $1, \alpha, \alpha^2, ...$

(NOT CACHED).

EXAMPLES:

This function allows us to write elements of a number field in terms of a different generator without having to construct a whole separate number field.

```
sage: y = polygen(QQ,'y'); K.<beta> = NumberField(y^3 - 2); K
Number Field in beta with defining polynomial y^3 - 2
sage: alpha = beta^2 + beta + 1
sage: c = alpha.coordinates_in_terms_of_powers(); c
Coordinate function that writes elements in terms of the powers of beta^2 + beta + 1
```

```
sage: c(beta)
[-2, -3, 1]
sage: c(alpha)
[0, 1, 0]
sage: c((1+beta)^5)
[3, 3, 3]
sage: c((1+beta)^10)
[54, 162, 189]
```

This function works even if self only generates a subfield of this number field.

```
sage: k.<a> = NumberField(x^6 - 5)
sage: alpha = a^3
sage: c = alpha.coordinates_in_terms_of_powers()
sage: c((2/3)*a^3 - 5/3)
[-5/3, 2/3]
sage: c
Coordinate function that writes elements in terms of the powers of a^3
sage: c(a)
...
ArithmeticError: vector is not in free module
```

denominator()

Return the denominator of this element, which is by definition the denominator of the corresponding polynomial representation. I.e., elements of number fields are represented as a polynomial (in reduced form) modulo the modulus of the number field, and the denominator is the denominator of this polynomial.

EXAMPLES:

```
sage: K.<z> = CyclotomicField(3)
sage: a = 1/3 + (1/5)*z
sage: print a.denominator()
15
```

galois_conjugates()

Return all Gal(Qbar/Q)-conjugates of this number field element in the field K.

EXAMPLES:

In the first example the conjugates are obvious:

```
sage: K.<a> = NumberField(x^2 - 2)
sage: a.galois_conjugates(K)
[a, -a]
sage: K(3).galois_conjugates(K)
[3]
```

In this example the field is not Galois, so we have to pass to an extension to obtain the Galois conjugates.

```
sage: K.<a> = NumberField(x^3 - 2)
sage: c = a.galois_conjugates(K); c
[a]
sage: K.<a> = NumberField(x^3 - 2)
sage: c = a.galois_conjugates(K.galois_closure('a1')); c
[1/84*a1^4 + 13/42*a1, -1/252*a1^4 - 55/126*a1, -1/126*a1^4 + 8/63*a1]
sage: c[0]^3
2
sage: parent(c[0])
Number Field in a1 with defining polynomial x^6 + 40*x^3 + 1372
sage: parent(c[0]).is_galois()
True
```

There is only one Galois conjugate of $\sqrt[3]{2}$ in $\mathbf{Q}(\sqrt[3]{2})$.

```
sage: a.galois_conjugates(K)
[a]
```

Galois conjugates of $\sqrt[3]{2}$ in the field $\mathbf{Q}(\zeta_3, \sqrt[3]{2})$:

```
sage: L.<a> = CyclotomicField(3).extension(x^3 - 2)
sage: a.galois_conjugates(L)
[a, (-zeta3 - 1)*a, zeta3*a]
```

is_integral()

Determine if a number is in the ring of integers of this number field.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 23)
sage: a.is_integral()
True
sage: t = (1+a)/2
sage: t.is_integral()
True
sage: t.minpoly()
x^2 - x + 6
sage: t = a/2
sage: t.is_integral()
False
sage: t.minpoly()
x^2 + 23/4
```

An example in a relative extension:

```
sage: K.<a,b> = NumberField([x^2+1, x^2+3])
sage: (a+b).is_integral()
True
sage: ((a-b)/2).is_integral()
False
```

is_square()

Return True if self is a square in its parent number field and otherwise return False.

INPUT:

- `root` - if True, also return a square root (or None if self is not a perfect square)

EXAMPLES:

```
sage: m.<b> = NumberField(x^4 - 1789)
sage: b.is_square()
False
sage: c = (2/3*b + 5)^2; c
4/9*b^2 + 20/3*b + 25
sage: c.is_square()
True
sage: c.is_square(True)
(True, 2/3*b + 5)
```

We also test the functional notation.

```
sage: is_square(c, True)
(True, 2/3*b + 5)
sage: is_square(c)
True
sage: is_square(c+1)
False
```

is_totally_positive()

Returns True if self is positive for all real embeddings of its parent number field. We do nothing at complex places, so e.g. any element of a totally complex number field will return True.

EXAMPLES:

```
sage: F.<b> = NumberField(x^3-3*x-1)
sage: b.is_totally_positive()
False
sage: (b^2).is_totally_positive()
True
```

list()

Return the list of coefficients of self written in terms of a power basis.

matrix()

If base is None, return the matrix of right multiplication by the element on the power basis $1, x, x^2, \ldots, x^{d-1}$ for the number field. Thus the *rows* of this matrix give the images of each of the x^i.

If base is not None, then base must be either a field that embeds in the parent of self or a morphism to the parent of self, in which case this function returns the matrix of multiplication by self on the power basis, where we view the parent field as a field over base.

INPUT:

- `base` - field or morphism

EXAMPLES:

Regular number field:

```
sage: K.<a> = NumberField(QQ['x'].0^3 - 5)
sage: M = a.matrix(); M
[0 1 0]
[0 0 1]
[5 0 0]
sage: M.base_ring() is QQ
True
```

Relative number field:

```
sage: L.<b> = K.extension(K['x'].0^2 - 2)
sage: M = b.matrix(); M
[0 1]
[2 0]
sage: M.base_ring() is K
True
```

Absolute number field:

```
sage: M = L.absolute_field('c').gen().matrix(); M
[  0   1   0   0   0   0]
[  0   0   1   0   0   0]
[  0   0   0   1   0   0]
[  0   0   0   0   1   0]
[  0   0   0   0   0   1]
[-17 -60 -12 -10   6   0]
sage: M.base_ring() is QQ
True
```

More complicated relative number field:

```
sage: L.<b> = K.extension(K['x'].0^2 - a); L
Number Field in b with defining polynomial x^2 - a over its base field
sage: M = b.matrix(); M
```

```
[0 1]
[a 0]
sage: M.base_ring() is K
True
```

An example where we explicitly give the subfield or the embedding:

```
sage: K.<a> = NumberField(x^4 + 1); L.<a2> = NumberField(x^2 + 1)
sage: a.matrix(L)
[ 0  1]
[a2  0]
```

Notice that if we compute all embeddings and choose a different one, then the matrix is changed as it should be:

```
sage: v = L.embeddings(K)
sage: a.matrix(v[1])
[  0   1]
[-a2   0]
```

The norm is also changed:

```
sage: a.norm(v[1])
a2
sage: a.norm(v[0])
-a2
```

TESTS:

```
sage: F.<z> = CyclotomicField(5) ; t = 3*z**3 + 4*z**2 + 2
sage: t.matrix(F)
[3*z^3 + 4*z^2 + 2]
```

minpoly()

Return the minimal polynomial of this number field element.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2+3)
sage: a.minpoly('x')
x^2 + 3
sage: R.<X> = K['X']
sage: L.<b> = K.extension(X^2-(22 + a))
sage: b.minpoly('t')
t^2 - a - 22
sage: b.absolute_minpoly('t')
t^4 - 44*t^2 + 487
sage: b^2 - (22+a)
0
```

multiplicative_order()

Return the multiplicative order of this number field element.

EXAMPLES:

```
sage: K.<z> = CyclotomicField(5)
sage: z.multiplicative_order()
5
sage: (-z).multiplicative_order()
10
sage: (1+z).multiplicative_order()
+Infinity
```

```
sage: x = polygen(QQ)
sage: K.<a>=NumberField(x^40 - x^20 + 4)
sage: u = 1/4*a^30 + 1/4*a^10 + 1/2
sage: u.multiplicative_order()
6
sage: a.multiplicative_order()
+Infinity
```

An example in a relative extension:

sage: K.<a, b> = NumberField([x^2 + x + 1, x^2 - 3]) sage: z = (a - 1)*b/3 sage: z.multiplicative_order() 12 sage: z^12==1 and z^6!=1 and z^4!=1 True

norm()

Return the absolute or relative norm of this number field element.

If K is given then K must be a subfield of the parent L of self, in which case the norm is the relative norm from L to K. In all other cases, the norm is the absolute norm down to QQ.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3 + x^2 + x - 132/7); K
Number Field in a with defining polynomial x^3 + x^2 + x - 132/7
sage: a.norm()
132/7
sage: factor(a.norm())
2^2 * 3 * 7^-1 * 11
sage: K(0).norm()
0
```

Some complicated relatives norms in a tower of number fields.

```
sage: K.<a,b,c> = NumberField([x^2 + 1, x^2 + 3, x^2 + 5])
sage: L = K.base_field(); M = L.base_field()
sage: a.norm()
1
sage: a.norm(L)
1
sage: a.norm(M)
1
sage: a
a
sage: (a+b+c).norm()
121
sage: (a+b+c).norm(L)
2*c*b - 7
sage: (a+b+c).norm(M)
-11
```

We illustrate that norm is compatible with towers:

```
sage: z = (a+b+c).norm(L); z.norm(M)
-11
```

If we are in an order, the norm is an integer:

```
sage: K.<a> = NumberField(x^3-2)
sage: a.norm().parent()
Rational Field
sage: R = K.ring_of_integers()
sage: R(a).norm().parent()
Integer Ring
```

TESTS:

```
sage: F.<z> = CyclotomicField(5)
sage: t = 3*z**3 + 4*z**2 + 2
sage: t.norm(F)
3*z^3 + 4*z^2 + 2
```

nth_root()

Return an nth root of self in the given number field.

EXAMPLES:

```
sage: K.<a> = NumberField(x^4-7)
sage: K(7).nth_root(2)
a^2
sage: K((a-3)^5).nth_root(5)
a - 3
```

ALGORITHM: Use Pari to factor x^n - `self` in K.

polynomial()

Return the underlying polynomial corresponding to this number field element.

The resulting polynomial is currently *not* cached.

EXAMPLES:

```
sage: K.<a> = NumberField(x^5 - x - 1)
sage: f = (-2/3 + 1/3*a)^4; f
1/81*a^4 - 8/81*a^3 + 8/27*a^2 - 32/81*a + 16/81
sage: g = f.polynomial(); g
1/81*x^4 - 8/81*x^3 + 8/27*x^2 - 32/81*x + 16/81
sage: parent(g)
Univariate Polynomial Ring in x over Rational Field
```

Note that the result of this function is not cached (should this be changed?):

```
sage: g is f.polynomial()
False
```

sqrt()

Returns the square root of this number in the given number field.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 - 3)
sage: K(3).sqrt()
a
sage: K(3).sqrt(all=True)
[a, -a]
sage: K(a^10).sqrt()
9*a
sage: K(49).sqrt()
7
sage: K(1+a).sqrt()
...
ValueError: a + 1 not a square in Number Field in a with defining polynomial x^2 - 3
sage: K(0).sqrt()
0
sage: K((7+a)^2).sqrt(all=True)
[a + 7, -a - 7]

sage: K.<a> = CyclotomicField(7)
sage: a.sqrt()
a^4
```

```
sage: K.<a> = NumberField(x^5 - x + 1)
sage: (a^4 + a^2 - 3*a + 2).sqrt()
a^3 - a^2
```

ALGORITHM: Use Pari to factor x^2 - `self` in K.

support()

Return the support of this number field element.

OUTPUT: A sorted list of the primes ideals at which this number field element has nonzero valuation. An error is raised if the element is zero.

EXAMPLES:

```
sage: x = ZZ['x'].gen()
sage: F.<t> = NumberField(x^3 - 2)

sage: P5s = F(5).support()
sage: P5s
[Fractional ideal (-t^2 - 1), Fractional ideal (t^2 - 2*t - 1)]
sage: all(5 in P5 for P5 in P5s)
True
sage: all(P5.is_prime() for P5 in P5s)
True
sage: [ P5.norm() for P5 in P5s ]
[5, 25]
```

TESTS:

It doesn't make sense to factor the ideal (0):

```
sage: F(0).support()
...
ArithmeticError: Support of 0 is not defined.
```

trace()

Return the absolute or relative trace of this number field element.

If K is given then K must be a subfield of the parent L of self, in which case the trace is the relative trace from L to K. In all other cases, the trace is the absolute trace down to QQ.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3 -132/7*x^2 + x + 1); K
Number Field in a with defining polynomial x^3 - 132/7*x^2 + x + 1
sage: a.trace()
132/7
sage: (a+1).trace() == a.trace() + 3
True
```

If we are in an order, the trace is an integer:

```
sage: K.<zeta> = CyclotomicField(17)
sage: R = K.ring_of_integers()
sage: R(zeta).trace().parent()
Integer Ring
```

TESTS:

```
sage: F.<z> = CyclotomicField(5) ; t = 3*z**3 + 4*z**2 + 2
sage: t.trace(F)
3*z^3 + 4*z^2 + 2
```

valuation()

Returns the valuation of self at a given prime ideal P.

INPUT:

- P - a prime ideal of the parent of self

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^4+3*x^2-17)
sage: P = K.ideal(61).factor()[0][0]
sage: b = a^2 + 30
sage: b.valuation(P)
1
sage: type(b.valuation(P))
<type 'sage.rings.integer.Integer'>
```

vector()

Return vector representation of self in terms of the basis for the ambient number field.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 1)
sage: (2/3*a - 5/6).vector()
(-5/6, 2/3)
sage: (-5/6, 2/3)
(-5/6, 2/3)
sage: O = K.order(2*a)
sage: (O.1).vector()
(0, 2)
sage: K.<a,b> = NumberField([x^2 + 1, x^2 - 3])
sage: (a + b).vector()
(b, 1)
sage: O = K.order([a,b])
sage: (O.1).vector()
(-b, 1)
sage: (O.2).vector()
(1, -b)
```

class **NumberFieldElement_absolute**()

absolute_charpoly()

Return the characteristic polynomial of this element over

For the meaning of the optional argument algorithm, see charpoly().

EXAMPLES:

```
sage: x = ZZ['x'].0
sage: K.<a> = NumberField(x^4 + 2, 'a')
sage: a.absolute_charpoly()
x^4 + 2
sage: a.absolute_charpoly('y')
y^4 + 2
sage: (-a^2).absolute_charpoly()
x^4 + 4*x^2 + 4
sage: (-a^2).absolute_minpoly()
x^2 + 2

sage: a.absolute_charpoly(algorithm='pari') == a.absolute_charpoly(algorithm='sage')
True
```

absolute_minpoly()

Return the minimal polynomial of this element over $\mathbf{Q}$.

For the meaning of the optional argument algorithm, see charpoly().

EXAMPLES:

```
sage: x = ZZ['x'].0
sage: f = x^10 - 5*x^9 + 15*x^8 - 68*x^7 + 81*x^6 - 221*x^5 + 141*x^4 - 242*x^3 - 13*x^2 - 3
sage: K.<a> = NumberField(f, 'a')
sage: a.absolute_charpoly()
x^10 - 5*x^9 + 15*x^8 - 68*x^7 + 81*x^6 - 221*x^5 + 141*x^4 - 242*x^3 - 13*x^2 - 33*x - 135
sage: a.absolute_charpoly('y')
y^10 - 5*y^9 + 15*y^8 - 68*y^7 + 81*y^6 - 221*y^5 + 141*y^4 - 242*y^3 - 13*y^2 - 33*y - 135
sage: b = -79/9995*a^9 + 52/9995*a^8 + 271/9995*a^7 + 1663/9995*a^6 + 13204/9995*a^5 + 5573/
sage: b.absolute_charpoly()
x^10 + 10*x^9 + 25*x^8 - 80*x^7 - 438*x^6 + 80*x^5 + 2950*x^4 + 1520*x^3 - 10439*x^2 - 5130*
sage: b.absolute_minpoly()
x^5 + 5*x^4 - 40*x^2 - 19*x + 135

sage: b.absolute_minpoly(algorithm='pari') == b.absolute_minpoly(algorithm='sage')
True
```

charpoly()

The characteristic polynomial of this element, over $\mathbf{Q}$ if self is an element of a field, and over $\mathbf{Z}$ is self is an element of an order.

This is the same as `self.absolute_charpoly` since this is an element of an absolute extension.

The optional argument algorithm controls how the characteristic polynomial is computed: 'pari' uses Pari, 'sage' uses charpoly for Sage matrices. The default value None means that 'pari' is used for small degrees (up to the value of the constant TUNE_CHARPOLY_NF, currently at 25), otherwise 'sage' is used. The constant TUNE_CHARPOLY_NF should give reasonable performance on all architectures; however, if you feel the need to customize it to your own machine, see trac ticket 5213 for a tuning script.

EXAMPLES:

We compute the characteristic polynomial of the cube root of 2.

```
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^3-2)
sage: a.charpoly('x')
x^3 - 2
sage: a.charpoly('y').parent()
Univariate Polynomial Ring in y over Rational Field
```

TESTS:

```
sage: R = K.ring_of_integers()
sage: R(a).charpoly()
x^3 - 2
sage: R(a).charpoly().parent()
Univariate Polynomial Ring in x over Integer Ring

sage: R(a).charpoly(algorithm='pari') == R(a).charpoly(algorithm='sage')
True
```

list()

Return the list of coefficients of self written in terms of a power basis.

EXAMPLE:

```
sage: K.<z> = CyclotomicField(3)
sage: (2+3/5*z).list()
[2, 3/5]
sage: (5*z).list()
[0, 5]
sage: K(3).list()
[3, 0]
```

minpoly()

Return the minimal polynomial of this number field element.

For the meaning of the optional argument algorithm, see charpoly().

EXAMPLES:

We compute the characteristic polynomial of cube root of 2.

```
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^3-2)
sage: a.minpoly('x')
x^3 - 2
sage: a.minpoly('y').parent()
Univariate Polynomial Ring in y over Rational Field
```

TESTS:

```
sage: R = K.ring_of_integers()
sage: R(a).minpoly()
x^3 - 2
sage: R(a).minpoly().parent()
Univariate Polynomial Ring in x over Integer Ring

sage: R(a).minpoly(algorithm='pari') == R(a).minpoly(algorithm='sage')
True
```

class **NumberFieldElement_relative**()

The current relative number field element implementation does everything in terms of absolute polynomials.

All conversions from relative polynomials, lists, vectors, etc should happen in the parent.

absolute_charpoly()

The characteristic polynomial of this element over $\mathbf{Q}$.

We construct a relative extension and find the characteristic polynomial over $\mathbf{Q}$.

The optional argument algorithm controls how the characteristic polynomial is computed: 'pari' uses Pari, 'sage' uses charpoly for Sage matrices. The default value None means that 'pari' is used for small degrees (up to the value of the constant TUNE_CHARPOLY_NF, currently at 25), otherwise 'sage' is used. The constant TUNE_CHARPOLY_NF should give reasonable performance on all architectures; however, if you feel the need to customize it to your own machine, see trac ticket 5213 for a tuning script.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^3-2)
sage: S.<X> = K[]
sage: L.<b> = NumberField(X^3 + 17); L
Number Field in b with defining polynomial X^3 + 17 over its base field
sage: b.absolute_charpoly()
x^9 + 51*x^6 + 867*x^3 + 4913
sage: b.charpoly()(b)
0
sage: a = L.0; a
b
sage: a.absolute_charpoly('x')
x^9 + 51*x^6 + 867*x^3 + 4913
sage: a.absolute_charpoly('y')
y^9 + 51*y^6 + 867*y^3 + 4913

sage: a.absolute_charpoly(algorithm='pari') == a.absolute_charpoly(algorithm='sage')
True
```

absolute_minpoly()

Return the minimal polynomial over $\mathbf{Q}$ of this element.

For the meaning of the optional argument algorithm, see absolute_charpoly().

EXAMPLES:

```
sage: K.<a, b> = NumberField([x^2 + 2, x^2 + 1000*x + 1])
sage: y = K['y'].0
sage: L.<c> = K.extension(y^2 + a*y + b)
sage: c.absolute_charpoly()
x^8 - 1996*x^6 + 996006*x^4 + 1997996*x^2 + 1
sage: c.absolute_minpoly()
x^8 - 1996*x^6 + 996006*x^4 + 1997996*x^2 + 1
sage: L(a).absolute_charpoly()
x^8 + 8*x^6 + 24*x^4 + 32*x^2 + 16
sage: L(a).absolute_minpoly()
x^2 + 2
sage: L(b).absolute_charpoly()
x^8 + 4000*x^7 + 6000004*x^6 + 4000012000*x^5 + 1000012000006*x^4 + 4000012000*x^3 + 6000004
sage: L(b).absolute_minpoly()
x^2 + 1000*x + 1
```

charpoly()

The characteristic polynomial of this element over its base field.

EXAMPLES:

```
sage: x = ZZ['x'].0
sage: K.<a, b> = QQ.extension([x^2 + 2, x^5 + 400*x^4 + 11*x^2 + 2])
sage: a.charpoly()
x^2 + 2
sage: b.charpoly()
x^2 - 2*b*x + b^2
sage: b.minpoly()
x - b

sage: K.<a, b> = NumberField([x^2 + 2, x^2 + 1000*x + 1])
sage: y = K['y'].0
sage: L.<c> = K.extension(y^2 + a*y + b)
sage: c.charpoly()
x^2 + a*x + b
sage: c.minpoly()
x^2 + a*x + b
sage: L(a).charpoly()
x^2 - 2*a*x - 2
sage: L(a).minpoly()
x - a
sage: L(b).charpoly()
x^2 - 2*b*x - 1000*b - 1
sage: L(b).minpoly()
x - b
```

list()

Return the list of coefficients of self written in terms of a power basis.

EXAMPLES:

```
sage: K.<a,b> = NumberField([x^3+2, x^2+1])
sage: a.list()
[0, 1, 0]
sage: v = (K.base_field().0 + a)^2 ; v
a^2 + 2*b*a - 1
sage: v.list()
[-1, 2*b, 1]
```

class `OrderElement_absolute`()

Element of an order in an absolute number field.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 1)
sage: O2 = K.order(2*a)
sage: w = O2.1; w
2*a
sage: parent(w)
Order in Number Field in a with defining polynomial x^2 + 1

sage: w.absolute_charpoly()
x^2 + 4
sage: w.absolute_charpoly().parent()
Univariate Polynomial Ring in x over Integer Ring
sage: w.absolute_minpoly()
x^2 + 4
sage: w.absolute_minpoly().parent()
Univariate Polynomial Ring in x over Integer Ring
```

`inverse_mod`()

Return an inverse of self modulo the given ideal.

INPUT:

- `I` - may be an ideal of self.parent(), or an element or list of elements of self.parent() generating a nonzero ideal. A TypeError is raised if I is non-integral, and a ValueError if the generators are all zero. A ZeroDivisionError is raised if I + (x) != (1).

EXAMPLES:

```
sage: OE = NumberField(x^3 - x + 2, 'w').ring_of_integers()
sage: w = OE.ring_generators()[0]
sage: w.inverse_mod(13*OE)
-7*w^2 - 13*w + 7
sage: w * (w.inverse_mod(13)) - 1 in 13*OE
True
sage: w.inverse_mod(2*OE)
...
ZeroDivisionError: w is not invertible modulo Fractional ideal (2)
```

class `OrderElement_relative`()

Element of an order in a relative number field.

EXAMPLES:

```
sage: O = EquationOrder([x^2 + x + 1, x^3 - 2],'a,b')
sage: c = O.1; c
(-2*b^2 - 2)*a - 2*b^2 - b
sage: type(c)
<type 'sage.rings.number_field.number_field_element.OrderElement_relative'>
```

`absolute_charpoly`()

The absolute characteristic polynomial of this order element over ZZ.

EXAMPLES:

```
sage: x = ZZ['x'].0
sage: K.<a,b> = NumberField([x^2 + 1, x^2 - 3])
sage: OK = K.maximal_order()
sage: _, u, _, v = OK.basis()
```

```
sage: t = 2*u - v; t
-b
sage: t.absolute_charpoly()
x^4 - 6*x^2 + 9
sage: t.absolute_minpoly()
x^2 - 3
sage: t.absolute_charpoly().parent()
Univariate Polynomial Ring in x over Integer Ring
```

absolute_minpoly()

The absolute minimal polynomial of this order element over ZZ.

EXAMPLES:

```
sage: x = ZZ['x'].0
sage: K.<a,b> = NumberField([x^2 + 1, x^2 - 3])
sage: OK = K.maximal_order()
sage: _, u, _, v = OK.basis()
sage: t = 2*u - v; t
-b
sage: t.absolute_charpoly()
x^4 - 6*x^2 + 9
sage: t.absolute_minpoly()
x^2 - 3
sage: t.absolute_minpoly().parent()
Univariate Polynomial Ring in x over Integer Ring
```

charpoly()

The characteristic polynomial of this order element over its base ring.

This special implementation works around bug #4738. At this time the base ring of relative order elements is ZZ; it should be the ring of integers of the base field.

EXAMPLES:

```
sage: x = ZZ['x'].0
sage: K.<a,b> = NumberField([x^2 + 1, x^2 - 3])
sage: OK = K.maximal_order(); OK.basis()
[1, 1/2*a - 1/2*b, -1/2*b*a + 1/2, a]
sage: charpoly(OK.1)
x^2 + b*x + 1
sage: charpoly(OK.1).parent()
Univariate Polynomial Ring in x over Maximal Order in Number Field in b with defining polync
sage: [ charpoly(t) for t in OK.basis() ]
[x^2 - 2*x + 1, x^2 + b*x + 1, x^2 - x + 1, x^2 + 1]
```

minpoly()

The minimal polynomial of this order element over its base ring.

This special implementation works around bug #4738. At this time the base ring of relative order elements is ZZ; it should be the ring of integers of the base field.

EXAMPLES:

```
sage: x = ZZ['x'].0
sage: K.<a,b> = NumberField([x^2 + 1, x^2 - 3])
sage: OK = K.maximal_order(); OK.basis()
[1, 1/2*a - 1/2*b, -1/2*b*a + 1/2, a]
sage: minpoly(OK.1)
x^2 + b*x + 1
sage: charpoly(OK.1).parent()
Univariate Polynomial Ring in x over Maximal Order in Number Field in b with defining polync
sage: _, u, _, v = OK.basis()
```

```
sage: t = 2*u - v; t
-b
sage: t.charpoly()
x^2 + 2*b*x + 3
sage: t.minpoly()
x + b

sage: t.absolute_charpoly()
x^4 - 6*x^2 + 9
sage: t.absolute_minpoly()
x^2 - 3
```

is_NumberFieldElement()

Return True if x is of type NumberFieldElement, i.e., an element of a number field.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field_element import is_NumberFieldElement
sage: is_NumberFieldElement(2)
False
sage: k.<a> = NumberField(x^7 + 17*x + 1)
sage: is_NumberFieldElement(a+1)
True
```

26.3 Number Field Ideals

AUTHORS:

- Steven Sivek (2005-05-16)
- William Stein (2007-09-06): vastly improved the doctesting
- William Stein and John Cremona (2007-01-28): new class NumberFieldFractionalIdeal now used for all except the 0 ideal

We test that pickling works:

```
sage: K.<a> = NumberField(x^2 - 5)
sage: I = K.ideal(2/(5+a))
sage: I == loads(dumps(I))
True
```

class LiftMap(*OK, M_OK_map, Q, I*)

Class to hold data needed by lifting maps from residue fields to number field orders.

class NumberFieldFractionalIdeal(*field, gens, coerce=True*)

A fractional ideal in a number field.

denominator()

Return the denominator ideal of this fractional ideal. Each fractional ideal has a unique expression as N/D where N, D are coprime integral ideals; the denominator is D.

EXAMPLES:

```
sage: K.<i>=NumberField(x^2+1)
sage: I = K.ideal((3+4*i)/5); I
Fractional ideal (4/5*i + 3/5)
```

```
sage: I.denominator()
Fractional ideal (2*i + 1)
sage: I.numerator()
Fractional ideal (-i - 2)
sage: I.numerator().is_integral() and I.denominator().is_integral()
True
sage: I.numerator() + I.denominator() == K.unit_ideal()
True
sage: I.numerator()/I.denominator() == I
True
```

divides(*other*)

Returns True if this ideal divides other and False otherwise.

EXAMPLES:

```
sage: K.<a> = CyclotomicField(11); K
Cyclotomic Field of order 11 and degree 10
sage: I = K.factor(31)[0][0]; I
Fractional ideal (-3*a^7 - 4*a^5 - 3*a^4 - 3*a^2 - 3*a - 3)
sage: I.divides(I)
True
sage: I.divides(31)
True
sage: I.divides(29)
False
```

element_1_mod(*other*)

Returns an element r in this ideal such that 1-r is in other

An error is raised if either ideal is not integral of if they are not coprime.

INPUT: other – another ideal of the same field, or generators of an ideal.

OUTPUT: r – an element of the ideal self such that 1-r is in the ideal other

AUTHOR: Maite Aranes

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-2)
sage: A = K.ideal(a+1); A; A.norm()
Fractional ideal (a + 1)
3
sage: B = K.ideal(a^2-4*a+2); B; B.norm()
Fractional ideal (a^2 - 4*a + 2)
68
sage: r = A.element_1_mod(B); r
a^2 + 5
sage: r in A
True
sage: 1-r in B
True
```

TESTS:

```
sage: K.<a> = NumberField(x^3-2)
sage: A = K.ideal(a+1)
sage: B = K.ideal(a^2-4*a+1); B; B.norm()
Fractional ideal (a^2 - 4*a + 1)
99
sage: A.element_1_mod(B)
...
TypeError: Fractional ideal (a + 1), Fractional ideal (a^2 - 4*a + 1) are not coprime ideals
```

```
sage: B = K.ideal(1/a); B
Fractional ideal (1/2*a^2)
sage: A.element_1_mod(B)
Traceback (most recent call last):
TypeError: element_1_mod only defined for integral ideals
```

euler_phi()

Returns the Euler φ-function of this integral ideal.

This is the order of the multiplicative group of the quotient modulo the ideal.

An error is raised if the ideal is not integral.

EXAMPLES:

```
sage: K.<i>=NumberField(x^2+1)
sage: I = K.ideal(2+i)
sage: [r for r in I.residues() if I.is_coprime(r)]
[-2, -1, 1, 2]
sage: I.euler_phi()
4
sage: J = I^3
sage: J.euler_phi()
100
sage: len([r for r in J.residues() if J.is_coprime(r)])
100
sage: J = K.ideal(3-2*i)
sage: I.is_coprime(J)
True
sage: I.euler_phi()*J.euler_phi() == (I*J).euler_phi()
True
sage: L.<b> = K.extension(x^2 - 7)
sage: L.ideal(3).euler_phi()
64
```

factor()

Factorization of this ideal in terms of prime ideals.

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 + 23); K
Number Field in a with defining polynomial x^4 + 23
sage: I = K.ideal(19); I
Fractional ideal (19)
sage: F = I.factor(); F
(Fractional ideal (a^2 + 2*a + 2)) * (Fractional ideal (a^2 - 2*a + 2))
sage: type(F)
<class 'sage.structure.factorization.Factorization'>
sage: list(F)
[(Fractional ideal (a^2 + 2*a + 2), 1), (Fractional ideal (a^2 - 2*a + 2), 1)]
sage: F.prod()
Fractional ideal (19)
```

idealcoprime(*J*)

Returns l such that l*self is coprime to J.

INPUT:

- J - another integral ideal of the same field as self, which must also be integral.

OUTPUT:

- l - an element such that l*self is coprime to the ideal J

TODO: Extend the implementation to non-integral ideals.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 23)
sage: A = k.ideal(a+1)
sage: B = k.ideal(3)
sage: A.is_coprime(B)
False
sage: lam = A.idealcoprime(B); lam
-1/6*a + 1/6
sage: (lam*A).is_coprime(B)
True
```

ALGORITHM: Uses Pari function `idealcoprime`.

ideallog(*x*)

Returns the discrete logarithm of x with respect to the generators given in the `bid` structure of the ideal self.

INPUT:

- **x - a non-zero element of the number field of self,** which must have valuation equal to 0 at all prime ideals in the support of the ideal self.

OUTPUT:

- `l` - a list of integers (x_i) such that $0 \leq x_i < d_i$ and $x = \prod_i g_i^{x_i}$ in $(R/I)^*$, where I = self, R = ring of integers of the field, and g_i are the generators of $(R/I)^*$, of orders d_i respectively, as given in the `bid` structure of the ideal self.

EXAMPLES:

```
sage: k.<a> = NumberField(x^3 - 11)
sage: A = k.ideal(5)
sage: G = A.idealstar(2)
sage: l = A.ideallog(a^2 +3)
sage: r = prod([G.gens()[i]**l[i] for i in range(len(l))])
sage: (a^2 + 3) - r in A
True
sage: A.small_residue(r) # random
a^2 - 2
```

ALGORITHM: Uses Pari function `ideallog`

idealstar(*flag=1*)

Returns the finite abelian group $(O_K/I)^*$, where I is the ideal self of the number field K, and O_K is the ring of integers of K.

INPUT:

- flag – when flag=2, it also computes the generators of the group $(O_K/I)^*$, which takes more time. By default flag=1 (no generators are computed). In both cases the special pari structure `bid` is computed as well. If flag=0 (deprecated) it computes only the group structure of $(O_K/I)^*$ (with generators) and not the special `bid` structure. OUTPUT: The finite abelian group $(O_K/I)^*$.

Note: Uses the pari function `idealstar`. The pari function outputs a special `bid` structure which is stored in the internal field `_bid` of the ideal (when flag=1,2). The special structure `bid` is used in the pari function `ideallog` to compute discrete logarithms.

EXAMPLES:

```
sage: k.<a> = NumberField(x^3 - 11)
sage: A = k.ideal(5)
sage: G = A.idealstar(); G
Multiplicative Abelian Group isomorphic to C24 x C4
sage: G.gens()
(f0, f1)
```

```
sage: G = A.idealstar(2)
sage: all([G.gens()[i] in k for i in range(G.ngens())])
True
```

ALGORITHM: Uses Pari function `idealstar`

invertible_residues(*reduce=True*)

Returns a iterator through a list of invertible residues modulo this integral ideal.

An error is raised if this fractional ideal is not integral.

INPUT:

- `reduce` - bool. If True (default), use `small_residue` to get small representatives of the residues.

OUTPUT:

- An iterator through a list of invertible residues modulo this ideal I, i.e. a list of elements in the ring of integers R representing the elements of $(R/I)^*$.

METHOD: Use pari's `idealstar` to find the group structure and generators of the multiplicative group modulo the ideal.

EXAMPLES:

```
sage: K.<i>=NumberField(x^2+1)
sage: ires =  K.ideal(2).invertible_residues(); ires  # random address
<generator object at 0xa2feb6c>
sage: list(ires)
[-1, -i]
sage: list(K.ideal(2+i).invertible_residues())
[1, 2, -1, -2]
sage: list(K.ideal(i).residues())
[0]
sage: list(K.ideal(i).invertible_residues())
[1]
sage: I = K.ideal(3+6*i)
sage: units=I.invertible_residues()
sage: len(list(units))==I.euler_phi()
True

sage: K.<a> = NumberField(x^3-10)
sage: I = K.ideal(a-1)
sage: len(list(I.invertible_residues())) == I.euler_phi()
True

sage: K.<z> = CyclotomicField(10)
sage: len(list(K.primes_above(3)[0].invertible_residues()))
80
```

AUTHOR: John Cremona

invertible_residues_mod(*subgp_gens=, [], reduce=True*)

Returns a iterator through a list of representatives for the invertible residues modulo this integral ideal, modulo the subgroup generated by the elements in the list `subgp_gens`.

INPUT:

- `subgp_gens` - either None or a list of elements of the number field of self. These need not be integral, but should be coprime to the ideal self. If the list is empty or None, the function returns an iterator through a list of representatives for the invertible residues modulo the integral ideal self.
- `reduce` - bool. If True (default), use `small_residues` to get small representatives of the residues.

Note: See also invertible_residues() for a simpler version without the subgroup.

OUTPUT:

- An iterator through a list of representatives for the invertible residues modulo self and modulo the group generated by `subgp_gens`, i.e. a list of elements in the ring of integers R representing the elements of $(R/I)^*/U$, where I is this ideal and U is the subgroup of $(R/I)^*$ generated by `subgp_gens`.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 +23)
sage: I = k.ideal(a)
sage: list(I.invertible_residues_mod([-1]))
[1, 5, 2, 10, 4, -3, 8, -6, -7, 11, 9]
sage: list(I.invertible_residues_mod([1/2]))
[1, 5]
sage: list(I.invertible_residues_mod([23]))
...
TypeError: the element must be invertible mod the ideal

sage: K.<a> = NumberField(x^3-10)
sage: I = K.ideal(a-1)
sage: len(list(I.invertible_residues_mod([]))) == I.euler_phi()
True

sage: I = K.ideal(1)
sage: list(I.invertible_residues_mod([]))
[1]

sage: K.<z> = CyclotomicField(10)
sage: len(list(K.primes_above(3)[0].invertible_residues_mod([])))
80
```

AUTHOR: Maite Aranes.

is_coprime(*other*)

Returns True if this ideal is coprime to the other, else False.

INPUT: other – another ideal of the same field, or generators of an ideal.

OUTPUT: True if self and other are coprime, else False.

NOTE: This function works for fractional ideals as well as integral ideals.

AUTHOR: John Cremona

EXAMPLES:

```
sage: K.<i>=NumberField(x^2+1)
sage: I = K.ideal(2+i)
sage: J = K.ideal(2-i)
sage: I.is_coprime(J)
True
sage: (I^-1).is_coprime(J^3)
True
sage: I.is_coprime(5)
False
sage: I.is_coprime(6+i)
True

# See trac \# 4536:
sage: E.<a> = NumberField(x^5 + 7*x^4 + 18*x^2 + x - 3)
sage: OE = E.ring_of_integers()
sage: i,j,k = [u[0] for u in factor(3*OE)]
sage: (i/j).is_coprime(j/k)
False
sage: (j/k).is_coprime(j/k)
```

```
False

sage: F.<a, b> = NumberField([x^2 - 2, x^2 - 3])
sage: F.ideal(3 - a*b).is_coprime(F.ideal(3))
False
```

is_maximal()

Return True if this ideal is maximal. This is equivalent to self being prime, since it is nonzero.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3 + 3); K
Number Field in a with defining polynomial x^3 + 3
sage: K.ideal(5).is_maximal()
False
sage: K.ideal(7).is_maximal()
True
```

is_trivial(*proof=None*)

Returns True if this is a trivial ideal.

EXAMPLES:

```
sage: F.<a> = QuadraticField(-5)
sage: I = F.ideal(3)
sage: I.is_trivial()
False
sage: J = F.ideal(5)
sage: J.is_trivial()
False
sage: (I+J).is_trivial()
True
```

numerator()

Return the numerator ideal of this fractional ideal.

Each fractional ideal has a unique expression as N/D where N, D are coprime integral ideals. The numerator is N.

EXAMPLES:

```
sage: K.<i>=NumberField(x^2+1)
sage: I = K.ideal((3+4*i)/5); I
Fractional ideal (4/5*i + 3/5)
sage: I.denominator()
Fractional ideal (2*i + 1)
sage: I.numerator()
Fractional ideal (-i - 2)
sage: I.numerator().is_integral() and I.denominator().is_integral()
True
sage: I.numerator() + I.denominator() == K.unit_ideal()
True
sage: I.numerator()/I.denominator() == I
True
```

prime_factors()

Return a list of the prime ideal factors of self

OUTPUT: list – list of prime ideals (a new list is returned each time this function is called)

EXAMPLES:

```
sage: K.<w> = NumberField(x^2 + 23)
sage: I = ideal(w+1)
```

```
sage: I.prime_factors()
[Fractional ideal (2, 1/2*w - 1/2),
Fractional ideal (2, 1/2*w + 1/2),
Fractional ideal (3, -1/2*w - 1/2)]
```

prime_to_idealM_part(*M*)

Version for integral ideals of the prime_to_m_part function over ZZ. Returns the largest divisor of self that is coprime to the ideal M.

INPUT: M – an integral ideal of the same field, or generators of an ideal

OUTPUT: An ideal which is the largest divisor of self that is coprime to M.

AUTHOR: Maite Aranes

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 23)
sage: I = k.ideal(a+1)
sage: M = k.ideal(2, 1/2*a - 1/2)
sage: J = I.prime_to_idealM_part(M); J
Fractional ideal (12, 1/2*a + 13/2)
sage: J.is_coprime(M)
True

sage: J = I.prime_to_idealM_part(2); J
Fractional ideal (3, -1/2*a - 1/2)
sage: J.is_coprime(M)
True
```

ramification_index()

Return the ramification index of this fractional ideal, assuming it is prime. Otherwise, raise a ValueError.

The ramification index is the power of this prime appearing in the factorization of the prime in $\mathbf{Z}$ that this prime lies over.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 2); K
Number Field in a with defining polynomial x^2 + 2
sage: f = K.factor(2); f
(Fractional ideal (-a))^2
sage: f[0][0].ramification_index()
2
sage: K.ideal(13).ramification_index()
1
sage: K.ideal(17).ramification_index()
...
ValueError: the fractional ideal (= Fractional ideal (17)) is not prime
```

residue_class_degree()

Return the residue class degree of this fractional ideal, assuming it is prime. Otherwise, raise a ValueError.

The residue class degree of a prime ideal I is the degree of the extension O_K/I of its prime subfield.

EXAMPLES:

```
sage: K.<a> = NumberField(x^5 + 2); K
Number Field in a with defining polynomial x^5 + 2
sage: f = K.factor(19); f
(Fractional ideal (a^2 + a - 3)) * (Fractional ideal (-2*a^4 - a^2 + 2*a - 1)) * (Fractional
sage: [i.residue_class_degree() for i, _ in f]
[2, 2, 1]
```

residue_field(*names=None*)

Return the residue class field of this fractional ideal, which must be prime.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-7)
sage: P = K.ideal(29).factor()[0][0]
sage: P.residue_field()
Residue field in abar of Fractional ideal (2*a^2 + 3*a - 10)
sage: P.residue_field('z')
Residue field in z of Fractional ideal (2*a^2 + 3*a - 10)
```

Another example:

```
sage: K.<a> = NumberField(x^3-7)
sage: P = K.ideal(389).factor()[0][0]; P
Fractional ideal (389, a^2 - 44*a - 9)
sage: P.residue_class_degree()
2
sage: P.residue_field()
Residue field in abar of Fractional ideal (389, a^2 - 44*a - 9)
sage: P.residue_field('z')
Residue field in z of Fractional ideal (389, a^2 - 44*a - 9)
sage: FF.<w> = P.residue_field()
sage: FF
Residue field in w of Fractional ideal (389, a^2 - 44*a - 9)
sage: FF((a+1)^390)
36
sage: FF(a)
w
```

An example of reduction maps to the residue field: these are defined on the whole valuation ring, i.e. the subring of the number field consisting of elements with non-negative valuation. This shows that the issue raised in trac #1951 has been fixed:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: P1, P2 = [g[0] for g in K.factor(5)]; (P1,P2)
(Fractional ideal (-i - 2), Fractional ideal (2*i + 1))
sage: a = 1/(1+2*i)
sage: F1, F2 = [g.residue_field() for g in [P1,P2]]; (F1,F2)
(Residue field of Fractional ideal (-i - 2),
Residue field of Fractional ideal (2*i + 1))
sage: a.valuation(P1)
0
sage: F1(i/7)
4
sage: F1(a)
3
sage: a.valuation(P2)
-1
sage: F2(a)
Traceback (most recent call last):
ZeroDivisionError: Cannot reduce field element -2/5*i + 1/5 modulo Fractional ideal (2*i + 1
```

residues()

Returns a iterator through a complete list of residues modulo this integral ideal.

An error is raised if this fractional ideal is not integral.

AUTHOR: John Cremona

EXAMPLES:

```
sage: K.<i>=NumberField(x^2+1)
sage: res =  K.ideal(2).residues(); res  # random address
xmrange_iter([[0, 1], [0, 1]], <function <lambda> at 0xa252144>)
```

```
sage: list(res)
[0, 1, i, i + 1]
sage: list(K.ideal(2+i).residues())
[-2, -1, 0, 1, 2]
sage: list(K.ideal(i).residues())
[0]
sage: I = K.ideal(3+6*i)
sage: reps=I.residues()
sage: len(list(reps)) == I.norm()
True
sage: all([r==s or not (r-s) in I for r in reps for s in reps])
True

sage: K.<a> = NumberField(x^3-10)
sage: I = K.ideal(a-1)
sage: len(list(I.residues())) == I.norm()
True

sage: K.<z> = CyclotomicField(11)
sage: len(list(K.primes_above(3)[0].residues())) == 3**5
True
```

small_residue(*f*)

Given an element f of the ambient number field, returns an element g such that f - g belongs to the ideal self (which must be integral), and g is small.

Note: The reduced representative returned is not uniquely determined.

ALGORITHM: Uses Pari function `nfeltreduce`.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 5)
sage: I = k.ideal(a)
sage: I.small_residue(14)
-1

sage: K.<a> = NumberField(x^5 + 7*x^4 + 18*x^2 + x - 3)
sage: I = K.ideal(5)
sage: I.small_residue(a^2 -13)
a^2 + 5*a - 3
```

class **NumberFieldIdeal**(*field, gens, coerce=True*)

An ideal of a number field.

absolute_norm()

A synonym for norm.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.ideal(1 + 2*i).absolute_norm()
5
```

absolute_ramification_index()

A synonym for ramification_index.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.ideal(1 + i).absolute_ramification_index()
2
```

artin_symbol()

Return the Artin symbol (K / Q, self), where K is the number field of self. This is the unique element s of the decomposition group of self such that s(x) = x^p mod self where p is the residue characteristic of self. (Here self should be prime and unramified.)

See the artin_symbol method of the GaloisGroup_v2 class for further documentation and examples.

EXAMPLE:

```
sage: QuadraticField(-23, 'w').primes_above(7)[0].artin_symbol()
(1,2)
```

basis()

Return an immutable sequence of elements of this ideal (note: their parent is the number field) that form a basis for this ideal viewed as a ZZ-module.

OUTPUT: basis – an immutable sequence.

EXAMPLES:

```
sage: K.<z> = CyclotomicField(7)
sage: I = K.factor(11)[0][0]
sage: I.basis()           # warning -- choice of basis can be somewhat random
[11, 11*z, 11*z^2, z^3 + 5*z^2 + 4*z + 10, z^4 + z^2 + z + 5, z^5 + z^4 + z^3 + 2*z^2 + 6*z
```

An example of a non-integral ideal.:

```
sage: J = 1/I
sage: J          # warning -- choice of generators can be somewhat random
Fractional ideal (2/11*z^5 + 2/11*z^4 + 3/11*z^3 + 2/11)
sage: J.basis()           # warning -- choice of basis can be somewhat random
[1, z, z^2, 1/11*z^3 + 7/11*z^2 + 6/11*z + 10/11, 1/11*z^4 + 1/11*z^2 + 1/11*z + 7/11, 1/11*
```

coordinates(*x*)

Returns the coordinate vector of x with respect to this ideal.

INPUT: x – an element of the number field (or ring of integers) of this ideal.

OUTPUT: A vector of length n (the degree of the field) giving the coordinates of x with respect to the integral basis of the ideal. In general this will be a vector of rationals; it will consist of integers if and only if x is in the ideal.

AUTHOR: John Cremona 2008-10-31 Uses linear algebra. The change-of-basis matrix is cached. Provides simpler implementations for _contains_(), is_integral() and smallest_integer().

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: I = K.ideal(7+3*i)
sage: Ibasis = I.integral_basis(); Ibasis
[58, i + 41]
sage: a = 23-14*i
sage: acoords = I.coordinates(a); acoords
(597/58, -14)
sage: sum([Ibasis[j]*acoords[j] for j in range(2)]) == a
True
sage: b = 123+456*i
sage: bcoords = I.coordinates(b); bcoords
(-18573/58, 456)
sage: sum([Ibasis[j]*bcoords[j] for j in range(2)]) == b
True
```

decomposition_group()

Return the decomposition group of self, as a subset of the automorphism group of the number field of self. Raises an error if the field isn't Galois. See the decomposition_group method of the GaloisGroup_v2 class for further examples and doctests.

EXAMPLE:

```
sage: QuadraticField(-23, 'w').primes_above(7)[0].decomposition_group()
Galois group of Number Field in w with defining polynomial x^2 + 23
```

free_module()

Return the free ZZ-module contained in the vector space associated to the ambient number field, that corresponds to this ideal.

EXAMPLES:

```
sage: K.<z> = CyclotomicField(7)
sage: I = K.factor(11)[0][0]; I
Fractional ideal (-2*z^4 - 2*z^2 - 2*z + 1)
sage: A = I.free_module()
sage: A              # warning -- choice of basis can be somewhat random
Free module of degree 6 and rank 6 over Integer Ring
User basis matrix:
[11  0  0  0  0  0]
[ 0 11  0  0  0  0]
[ 0  0 11  0  0  0]
[10  4  5  1  0  0]
[ 5  1  1  0  1  0]
[ 5  6  2  1  1  1]
```

However, the actual ZZ-module is not at all random:

```
sage: A.basis_matrix().change_ring(ZZ).echelon_form()
[ 1  0  0  5  1  1]
[ 0  1  0  1  1  7]
[ 0  0  1  7  6 10]
[ 0  0  0 11  0  0]
[ 0  0  0  0 11  0]
[ 0  0  0  0  0 11]
```

The ideal doesn't have to be integral:

```
sage: J = I^(-1)
sage: B = J.free_module()
sage: B.echelonized_basis_matrix()
[ 1/11     0     0  7/11  1/11  1/11]
[    0  1/11     0  1/11  1/11  5/11]
[    0     0  1/11  5/11  4/11 10/11]
[    0     0     0     1     0     0]
[    0     0     0     0     1     0]
[    0     0     0     0     0     1]
```

This also works for relative extensions:

```
sage: K.<a,b> = NumberField([x^2 + 1, x^2 + 2])
sage: I = K.fractional_ideal(4)
sage: I.free_module()
Free module of degree 4 and rank 4 over Integer Ring
User basis matrix:
[  4   0   0   0]
[ -3   7  -1   1]
[  3   7   1   1]
[  0 -10   0  -2]
sage: J = I^(-1); J.free_module()
Free module of degree 4 and rank 4 over Integer Ring
User basis matrix:
[  1/4     0     0     0]
[-3/16  7/16 -1/16  1/16]
```

```
[ 3/16  7/16  1/16  1/16]
[    0  -5/8     0  -1/8]
```

An example of intersecting ideals by intersecting free modules.:

```
sage: K.<a> = NumberField(x^3 + x^2 - 2*x + 8)
sage: I = K.factor(2)
sage: p1 = I[0][0]; p2 = I[1][0]
sage: N = p1.free_module().intersection(p2.free_module()); N
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[  1 1/2 1/2]
[  0   1   1]
[  0   0   2]
sage: N.index_in(p1.free_module()).abs()
2
```

TESTS:

Sage can find the free module associated to quite large ideals quickly (see trac #4627):

```
sage: y = polygen(ZZ)
sage: M.<a> = NumberField(y^20 - 2*y^19 + 10*y^17 - 15*y^16 + 40*y^14 - 64*y^13 + 46*y^12 +
sage: M.ideal(prod(prime_range(6000, 6200))).free_module()
Free module of degree 20 and rank 20 over Integer Ring
User basis matrix:
20 x 20 dense matrix over Rational Field
```

gens_reduced(*proof=None*)

Express this ideal in terms of at most two generators, and one if possible.

Note that if the ideal is not principal, then this uses PARI's `idealtwoelt` function, which takes exponential time, the first time it is called for each ideal. Also, this indirectly uses `bnfisprincipal`, so set `proof=True` if you want to prove correctness (which *is* the default).

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<i> = NumberField(x^2+1, 'i')
sage: J = K.ideal([i+1, 2])
sage: J.gens()
(i + 1, 2)
sage: J.gens_reduced()
(i + 1,)
```

TESTS:

```
sage: all(j.parent() is K for j in J.gens())
True
sage: all(j.parent() is K for j in J.gens_reduced())
True

sage: K.<a> = NumberField(x^4 + 10*x^2 + 20)
sage: J = K.prime_above(5)
sage: J.is_principal()
False
sage: J.gens_reduced()
(5, a)
sage: all(j.parent() is K for j in J.gens())
True
sage: all(j.parent() is K for j in J.gens_reduced())
True
```

inertia_group()

Return the inertia group of self, i.e. the set of elements s of the Galois group of the number field of self (which we assume is Galois) such that s acts trivially modulo self. This is the same as the 0th ramification group of self. See the inertia_group method of the GaloisGroup_v2 class for further examples and doctests.

EXAMPLE:

```
sage: QuadraticField(-23, 'w').primes_above(23)[0].inertia_group()
Galois group of Number Field in w with defining polynomial x^2 + 23
```

integral_basis()

Return a list of generators for this ideal as a $\mathbf{Z}$-module.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<i> = NumberField(x^2 + 1)
sage: J = K.ideal(i+1)
sage: J.integral_basis()
[2, i + 1]
```

integral_split()

Return a tuple (I, d), where I is an integral ideal, and d is the smallest positive integer such that this ideal is equal to I/d.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^2-5)
sage: I = K.ideal(2/(5+a))
sage: I.is_integral()
False
sage: J,d = I.integral_split()
sage: J
Fractional ideal (-1/2*a + 5/2)
sage: J.is_integral()
True
sage: d
5
sage: I == J/d
True
```

is_integral()

Return True if this ideal is integral.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^5-x+1)
sage: K.ideal(a).is_integral()
True
sage: (K.ideal(1) / (3*a+1)).is_integral()
False
```

is_maximal()

Return True if this ideal is maximal. This is equivalent to self being prime and nonzero.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3 + 3); K
Number Field in a with defining polynomial x^3 + 3
sage: K.ideal(5).is_maximal()
False
```

```
sage: K.ideal(7).is_maximal()
True
```

is_prime()

Return True if this ideal is prime.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 - 17); K
Number Field in a with defining polynomial x^2 - 17
sage: K.ideal(5).is_prime()
True
sage: K.ideal(13).is_prime()
False
sage: K.ideal(17).is_prime()
False
```

is_principal(*proof=None*)

Return True if this ideal is principal.

Since it uses the PARI method code{bnfisprincipal}, specify code{proof=True} (this is the default setting) to prove the correctness of the output.

EXAMPLES:

We create equal ideals in two different ways, and note that they are both actually principal ideals.:

```
sage: K = QuadraticField(-119,'a')
sage: P = K.ideal([2]).factor()[1][0]
sage: I = P^5
sage: I.is_principal()
True
```

is_zero()

Return True iff self is the zero ideal

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 2); K
Number Field in a with defining polynomial x^2 + 2
sage: K.ideal(3).is_zero()
False
sage: I=K.ideal(0); I.is_zero()
True
sage: I
Ideal (0) of Number Field in a with defining polynomial x^2 + 2

(0 is a NumberFieldIdeal, not a NumberFieldFractionIdeal)
```

norm()

Return the norm of this fractional ideal as a rational number.

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 + 23); K
Number Field in a with defining polynomial x^4 + 23
sage: I = K.ideal(19); I
Fractional ideal (19)
sage: factor(I.norm())
19^4
sage: F = I.factor()
sage: F[0][0].norm().factor()
19^2
```

number_field()
Return the number field that this is a fractional ideal in.
EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 2); K
Number Field in a with defining polynomial x^2 + 2
sage: K.ideal(3).number_field()
Number Field in a with defining polynomial x^2 + 2
sage: K.ideal(0).number_field() # not tested (not implemented)
Number Field in a with defining polynomial x^2 + 2
```

pari_hnf()
Return PARI's representation of this ideal in Hermite normal form.
EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^3 - 2)
sage: I = K.ideal(2/(5+a))
sage: I.pari_hnf()
[2, 0, 50/127; 0, 2, 244/127; 0, 0, 2/127]
```

ramification_group(v)
Return the vth ramification group of self, i.e. the set of elements s of the Galois group of the number field of self (which we assume is Galois) such that s acts trivially modulo self^(v+1). See the ramification_group method of the GaloisGroup class for further examples and doctests.
EXAMPLE:

```
sage: QuadraticField(-23, 'w').primes_above(23)[0].ramification_group(0)
Galois group of Number Field in w with defining polynomial x^2 + 23
sage: QuadraticField(-23, 'w').primes_above(23)[0].ramification_group(1)
Subgroup [()] of Galois group of Number Field in w with defining polynomial x^2 + 23
```

reduce_equiv()
Return a small ideal that is equivalent to self in the group of fractional ideals modulo principal ideals. Very often (but not always) if self is principal then this function returns the unit ideal.
ALGORITHM: Calls pari's idealred function.
EXAMPLES:

```
sage: K.<w> = NumberField(x^2 + 23)
sage: I = ideal(w*23^5); I
Fractional ideal (6436343*w)
sage: I.reduce_equiv()
Fractional ideal (1)
sage: I = K.class_group().0.ideal()^10; I
Fractional ideal (1024, 1/2*w + 979/2)
sage: I.reduce_equiv()
Fractional ideal (2, 1/2*w - 1/2)
```

relative_norm()
A synonym for norm.
EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.ideal(1 + 2*i).relative_norm()
5
```

relative_ramification_index()
A synonym for ramification_index.
EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: K.ideal(1 + i).relative_ramification_index()
2
```

smallest_integer()

Return the smallest non-negative integer in $I \cap \mathbf{Z}$, where I is this ideal. If $I = 0$, returns 0.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^2+6)
sage: I = K.ideal([4,a])/7
sage: I.smallest_integer()
2

sage: K.<i> = QuadraticField(-1)
sage: P1, P2 = [P for P,e in K.factor(13)]
sage: all([(P1^i*P2^j).smallest_integer() == 13^max(i,j,0) for i in range(-3,3) for j in ran
True
sage: I = K.ideal(0)
sage: I.smallest_integer()
0

# See trac\# 4392:
sage: K.<a>=QuadraticField(-5)
sage: I=K.ideal(7)
sage: I.smallest_integer()
7

sage: K.<z> = CyclotomicField(13)
sage: a = K([-8, -4, -4, -6, 3, -4, 8, 0, 7, 4, 1, 2])
sage: I = K.ideal(a)
sage: I.smallest_integer()
146196692151
sage: I.norm()
1315770229359
sage: I.norm() / I.smallest_integer()
9
```

valuation(p)

Return the valuation of this fractional ideal at the prime $\mathfrak{p}$. If $\mathfrak{p}$ is not prime, raise a ValueError.

INPUT: p – a prime ideal of this number field.

OUTPUT: integer

EXAMPLES:

```
sage: K.<a> = NumberField(x^5 + 2); K
Number Field in a with defining polynomial x^5 + 2
sage: i = K.ideal(38); i
Fractional ideal (38)
sage: i.valuation(K.factor(19)[0][0])
1
sage: i.valuation(K.factor(2)[0][0])
5
sage: i.valuation(K.factor(3)[0][0])
0
sage: i.valuation(0)
...
ValueError: p (= 0) must be nonzero
```

class QuotientMap(*K, M_OK_change, Q, I*)

Class to hold data needed by quotient maps from number field orders to residue fields. These are only partial maps: the exact domain is the appropriate valuation ring. For examples, see residue_field().

basis_to_module(*B, K*)

Given a basis B of elements for a ZZ-submodule of a number field K, return the corresponding ZZ-submodule.

EXAMPLES:

```
sage: K.<w> = NumberField(x^4 + 1)
sage: from sage.rings.number_field.number_field_ideal import basis_to_module
sage: basis_to_module([K.0, K.0^2 + 3], K)
Free module of degree 4 and rank 2 over Integer Ring
User basis matrix:
[0 1 0 0]
[3 0 1 0]
```

convert_from_zk_basis(*field, hnf*)

Used internally in the number field ideal implementation for converting from the order basis to the number field basis.

INPUT:

- `field` - a number field
- `hnf` - a pari HNF matrix, output by the pari_hnf() function.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field_ideal import convert_from_zk_basis
sage: k.<a> = NumberField(x^2 + 23)
sage: I = k.factor(3)[0][0]; I
Fractional ideal (3, -1/2*a + 1/2)
sage: hnf = I.pari_hnf(); hnf
[3, 0; 0, 1]
sage: convert_from_zk_basis(k, hnf)
[3, 1/2*x - 1/2]
```

is_NumberFieldFractionalIdeal(*x*)

Return True if x is a fractional ideal of a number field.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field_ideal import is_NumberFieldFractionalIdeal
sage: is_NumberFieldFractionalIdeal(2/3)
False
sage: is_NumberFieldFractionalIdeal(ideal(5))
False
sage: k.<a> = NumberField(x^2 + 2)
sage: I = k.ideal([a + 1]); I
Fractional ideal (a + 1)
sage: is_NumberFieldFractionalIdeal(I)
True
sage: Z = k.ideal(0); Z
Ideal (0) of Number Field in a with defining polynomial x^2 + 2
sage: is_NumberFieldFractionalIdeal(Z)
False
```

is_NumberFieldIdeal(*x*)

Return True if x is an ideal of a number field.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field_ideal import is_NumberFieldIdeal
sage: is_NumberFieldIdeal(2/3)
False
sage: is_NumberFieldIdeal(ideal(5))
False
sage: k.<a> = NumberField(x^2 + 2)
sage: I = k.ideal([a + 1]); I
Fractional ideal (a + 1)
sage: is_NumberFieldIdeal(I)
True
sage: Z = k.ideal(0); Z
Ideal (0) of Number Field in a with defining polynomial x^2 + 2
sage: is_NumberFieldIdeal(Z)
True
```

quotient_char_p(*I*, *p*)

Given an integral ideal I that contains a prime number p, compute a vector space V = (OK mod p) / (I mod p), along with a homomorphism OK –> V and a section V –> OK.

EXAMPLES:

```
sage: from sage.rings.number_field.number_field_ideal import quotient_char_p

sage: K.<i> = NumberField(x^2 + 1); O = K.maximal_order(); I = K.fractional_ideal(15)
sage: quotient_char_p(I, 5)[0]
Vector space quotient V/W of dimension 2 over Finite Field of size 5 where
V: Vector space of dimension 2 over Finite Field of size 5
W: Vector space of degree 2 and dimension 0 over Finite Field of size 5
Basis matrix:
[]
sage: quotient_char_p(I, 3)[0]
Vector space quotient V/W of dimension 2 over Finite Field of size 3 where
V: Vector space of dimension 2 over Finite Field of size 3
W: Vector space of degree 2 and dimension 0 over Finite Field of size 3
Basis matrix:
[]

sage: I = K.factor(13)[0][0]; I
Fractional ideal (-3*i - 2)
sage: I.residue_class_degree()
1
sage: quotient_char_p(I, 13)[0]
Vector space quotient V/W of dimension 1 over Finite Field of size 13 where
V: Vector space of dimension 2 over Finite Field of size 13
W: Vector space of degree 2 and dimension 1 over Finite Field of size 13
Basis matrix:
[1 8]
```

26.4 Class Groups of Number Fields

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 23)
sage: I = K.class_group().gen(); I
Fractional ideal class (2, 1/2*a - 1/2)
```

```
sage: J = I * I; J
Fractional ideal class (2, 1/2*a + 1/2)
sage: O = K.OK(); O
Maximal Order in Number Field in a with defining polynomial x^2 + 23
sage: O*(2, 1/2*a + 1/2)
Fractional ideal (2, 1/2*a + 1/2)
sage: (O*(2, 1/2*a + 1/2)).is_principal()
False
sage: (O*(2, 1/2*a + 1/2))^3
Fractional ideal (1/2*a - 3/2)
```

class ClassGroup (*invariants, names, number_field, gens*)

The class group of a number field.

gen (*i=0*)

Return the i-th generator for this class group.

EXAMPLES:

```
sage: C = NumberField(x^2 + 120071, 'a').class_group(); C
Class group of order 500 with structure C250 x C2 of Number Field in a with defining polynom
sage: C.gen(0)
Fractional ideal class (130, 1/2*a + 137/2)
sage: C.gen(1)
Fractional ideal class (7, a)
```

gens ()

Return generators for the class group.

EXAMPLES:

```
sage: K.<a> = NumberField(x^4 + 23)
sage: K.class_group().gens()   # random gens (platform dependent)
[Fractional ideal class (2, 1/2*a^2 - a + 3/2)]
```

ngens ()

Return the number of generators of the class group.

EXAMPLES:

```
sage: C = NumberField(x^2 + x + 23899, 'a').class_group(); C
Class group of order 68 with structure C34 x C2 of Number Field in a with defining polynomia
sage: C.ngens()
2
```

number_field ()

Return the number field that this class group is attached to.

EXAMPLES:

```
sage: C = NumberField(x^2 + 23, 'w').class_group(); C
Class group of order 3 with structure C3 of Number Field in w with defining polynomial x^2 +
sage: C.number_field()
Number Field in w with defining polynomial x^2 + 23
```

class FractionalIdealClass (*ideal, class_group*)

A fractional ideal class in a number field.

EXAMPLES:

```
sage: G = NumberField(x^2 + 23,'a').class_group(); G
Class group of order 3 with structure C3 of Number Field in a with defining polynomial x^2 + 23
sage: I = G.0; I
```

```
Fractional ideal class (2, 1/2*a - 1/2)
sage: I*I
Fractional ideal class (2, 1/2*a + 1/2)
sage: I*I*I
Trivial principal fractional ideal class
```

gens()

Return generators for a representative ideal in this ideal class.

EXAMPLES:

```
sage: K.<w>=QuadraticField(-23)
sage: OK=K.ring_of_integers()
sage: C=OK.class_group()
sage: P2a,P2b=[P for P,e in (2*OK).factor()]
sage: c=C(P2a); c
Fractional ideal class (2, 1/2*w - 1/2)
sage: c.gens()
(2, 1/2*w - 1/2)
```

ideal()

Return a representative ideal in this ideal class.

EXAMPLE:

```
sage: K.<w>=QuadraticField(-23)
sage: OK=K.ring_of_integers()
sage: C=OK.class_group()
sage: P2a,P2b=[P for P,e in (2*OK).factor()]
sage: c=C(P2a); c
Fractional ideal class (2, 1/2*w - 1/2)
sage: c.ideal()
Fractional ideal (2, 1/2*w - 1/2)
```

is_principal()

Returns True iff this ideal class is the trivial (principal) class

EXAMPLES:

```
sage: K.<w>=QuadraticField(-23)
sage: OK=K.ring_of_integers()
sage: C=OK.class_group()
sage: P2a,P2b=[P for P,e in (2*OK).factor()]
sage: c=C(P2a)
sage: c.is_principal()
False
sage: (c^2).is_principal()
False
sage: (c^3).is_principal()
True
```

multiplicative_order()

Return the order of this ideal class in the class group.

EXAMPLE:

```
sage: K.<w>=QuadraticField(-23)
sage: OK=K.ring_of_integers()
sage: C=OK.class_group()
sage: h=C.order(); h
3
sage: P2a,P2b=[P for P,e in (2*OK).factor()]
```

```
sage: c=C(P2a); c
Fractional ideal class (2, 1/2*w - 1/2)
sage: c.order()
3

sage: k.<a> = NumberField(x^2 + 20072); G = k.class_group(); G
Class group of order 76 with structure C38 x C2 of Number Field in a with defining polynomia
sage: [c.order() for c in G.gens()]
[38, 2]
```

order()

Return the order of this ideal class in the class group.

EXAMPLE:

```
sage: K.<w>=QuadraticField(-23)
sage: OK=K.ring_of_integers()
sage: C=OK.class_group()
sage: h=C.order(); h
3
sage: P2a,P2b=[P for P,e in (2*OK).factor()]
sage: c=C(P2a); c
Fractional ideal class (2, 1/2*w - 1/2)
sage: c.order()
3

sage: k.<a> = NumberField(x^2 + 20072); G = k.class_group(); G
Class group of order 76 with structure C38 x C2 of Number Field in a with defining polynomia
sage: [c.order() for c in G.gens()]
[38, 2]
```

reduce()

Return representative for this ideal class that has been reduced using PARI's idealred.

EXAMPLES:

```
sage: k.<a> = NumberField(x^2 + 20072); G = k.class_group(); G
Class group of order 76 with structure C38 x C2 of Number Field in a with defining polynomia
sage: I = G.0; I
Fractional ideal class (41, a + 10)
sage: J = G(I.ideal()^5); J
Fractional ideal class (115856201, 1/2*a + 40407883)
sage: J.reduce()
Fractional ideal class (57, 1/2*a + 44)
```

26.5 Galois Groups of Number Fields

AUTHORS:

- William Stein (2004, 2005): initial version
- David Loeffler (2009): rewrite to give explicit homomorphism groups

TESTS:

Standard test of pickleability:

```
sage: G = NumberField(x^3 + 2, 'alpha').galois_group(type="pari"); G
Galois group PARI group [6, -1, 2, "S3"] of degree 3 of the Number Field in alpha with defining polyn
sage: G == loads(dumps(G))
True

sage: G = NumberField(x^3 + 2, 'alpha').galois_group(names='beta'); G
Galois group of Galois closure in beta of Number Field in alpha with defining polynomial x^3 + 2
sage: G == loads(dumps(G))
True
```

class GaloisGroup (*group, number_field*)

A wrapper around a class representing an abstract transitive group.

This is just a fairly minimal object at present. To get the underlying group, do `G.group()`, and to get the corresponding number field do `G.number_field()`. For a more sophisticated interface use the `type=None` option.

EXAMPLES:

```
sage: K = QQ[2^(1/3)]
sage: G = K.galois_group(type="pari"); G
Galois group PARI group [6, -1, 2, "S3"] of degree 3 of the Number Field in a with defining poly
sage: G.order()
6
sage: G.group()
PARI group [6, -1, 2, "S3"] of degree 3
sage: G.number_field()
Number Field in a with defining polynomial x^3 - 2
```

group()

Return the underlying abstract group.

EXAMPLES:

```
sage: G = NumberField(x^3 + 2*x + 2, 'theta').galois_group(type="pari")
sage: H = G.group(); H
PARI group [6, -1, 2, "S3"] of degree 3
sage: P = H.permutation_group(); P  # optional -- requires Gap optional databases
Transitive group number 2 of degree 3
sage: list(P)                       # optional
[(), (2,3), (1,2), (1,2,3), (1,3,2), (1,3)]
```

number_field()

Return the number field of which this is the Galois group.

EXAMPLES:

```
sage: G = NumberField(x^6 + 2, 't').galois_group(type="pari"); G
Galois group PARI group [12, -1, 3, "D(6) = S(3)[x]2"] of degree 6 of the Number Field in t
sage: G.number_field()
Number Field in t with defining polynomial x^6 + 2
```

order()

Return the order of this Galois group.

EXAMPLES:

```
sage: G = NumberField(x^5 + 2, 'theta_1').galois_group(type="pari"); G
Galois group PARI group [20, -1, 3, "F(5) = 5:4"] of degree 5 of the Number Field in theta_1
sage: G.order()
20
```

class GaloisGroupElement()

An element of a Galois group. This is stored as a permutation, but may also be made to act on elements of the field (generally returning elements of its Galois closure).

EXAMPLE:

```
sage: K.<w> = QuadraticField(-7); G = K.galois_group()
sage: G[1]
(1,2)
sage: G[1](w + 2)
-w + 2

sage: L.<v> = NumberField(x^3 - 2); G = L.galois_group(names='y')
sage: G[1]
(1,2)(3,4)(5,6)
sage: G[1](v)
1/84*y^4 + 13/42*y
sage: G[1]( G[1](v) )
-1/252*y^4 - 55/126*y
```

as_hom()

Return the homomorphism L -> L corresponding to self, where L is the Galois closure of the ambient number field.

EXAMPLE:

```
sage: G = QuadraticField(-7,'w').galois_group()
sage: G[1].as_hom()
Ring endomorphism of Number Field in w with defining polynomial x^2 + 7
Defn: w |--> -w
```

ramification_degree(*P*)

Return the greatest value of v such that s acts trivially modulo P^v. Should only be used if P is prime and s is in the decomposition group of P.

EXAMPLE:

```
sage: K.<b> = NumberField(x^3 - 3,'a').galois_closure()
sage: G=K.galois_group()
sage: P = K.primes_above(3)[0]
sage: s = hom(K, K, 1/54*b^4 + 1/18*b)
sage: G(s).ramification_degree(P)
4
```

class GaloisGroup_subgroup(*ambient, elts*)

A subgroup of a Galois group, as returned by functions such as `decomposition_group`.

fixed_field()

Return the fixed field of this subgroup (as a subfield of the Galois closure of the number field associated to the ambient Galois group).

EXAMPLE:

```
sage: L.<a> = NumberField(x^4 + 1)
sage: G = L.galois_group()
sage: H = G.decomposition_group(L.primes_above(3)[0])
sage: H.fixed_field()
(Number Field in a0 with defining polynomial x^2 + 2, Ring morphism:
From: Number Field in a0 with defining polynomial x^2 + 2
To:   Number Field in a with defining polynomial x^4 + 1
Defn: a0 |--> a^3 + a)
```

class GaloisGroup_v1 (*group, number_field*)

A wrapper around a class representing an abstract transitive group.

This is just a fairly minimal object at present. To get the underlying group, do `G.group()`, and to get the corresponding number field do `G.number_field()`. For a more sophisticated interface use the `type=None` option.

EXAMPLES:

```
sage: K = QQ[2^(1/3)]
sage: G = K.galois_group(type="pari"); G
Galois group PARI group [6, -1, 2, "S3"] of degree 3 of the Number Field in a with defining poly
sage: G.order()
6
sage: G.group()
PARI group [6, -1, 2, "S3"] of degree 3
sage: G.number_field()
Number Field in a with defining polynomial x^3 - 2
```

group()

Return the underlying abstract group.

EXAMPLES:

```
sage: G = NumberField(x^3 + 2*x + 2, 'theta').galois_group(type="pari")
sage: H = G.group(); H
PARI group [6, -1, 2, "S3"] of degree 3
sage: P = H.permutation_group(); P  # optional -- requires Gap optional databases
Transitive group number 2 of degree 3
sage: list(P)                       # optional
[(), (2,3), (1,2), (1,2,3), (1,3,2), (1,3)]
```

number_field()

Return the number field of which this is the Galois group.

EXAMPLES:

```
sage: G = NumberField(x^6 + 2, 't').galois_group(type="pari"); G
Galois group PARI group [12, -1, 3, "D(6) = S(3)[x]2"] of degree 6 of the Number Field in t
sage: G.number_field()
Number Field in t with defining polynomial x^6 + 2
```

order()

Return the order of this Galois group.

EXAMPLES:

```
sage: G = NumberField(x^5 + 2, 'theta_1').galois_group(type="pari"); G
Galois group PARI group [20, -1, 3, "F(5) = 5:4"] of degree 5 of the Number Field in theta_1
sage: G.order()
20
```

class GaloisGroup_v2 (*number_field, names=None*)

The Galois group of an (absolute) number field.

Note: We define the Galois group of a non-normal field K to be the Galois group of its Galois closure L, and elements are stored as permutations of the roots of the defining polynomial of L, *not* as permutations of the roots (in L) of the defining polynomial of K. The latter would probably be preferable, but is harder to implement. Thus the permutation group that is returned is always simply-transitive.

The 'arithmetical' features (decomposition and ramification groups, Artin symbols etc) are only available for Galois fields.

artin_symbol(*P*)

Return the Artin symbol $\left(\frac{K/\mathbf{Q}}{\mathfrak{P}}\right)$, where K is the number field of self, and $\mathfrak{P}$ is an unramified prime ideal. This is the unique element s of the decomposition group of $\mathfrak{P}$ such that $s(x) = x^p \bmod \mathfrak{P}$, where p is the residue characteristic of $\mathfrak{P}$.

EXAMPLES:

```
sage: K.<b> = NumberField(x^4 - 2*x^2 + 2, 'a').galois_closure()
sage: G = K.galois_group()
sage: [G.artin_symbol(P) for P in K.primes_above(7)]
[(1,4)(2,3)(5,8)(6,7), (1,4)(2,3)(5,8)(6,7), (1,5)(2,6)(3,7)(4,8), (1,5)(2,6)(3,7)(4,8)]
sage: G.artin_symbol(17)
...
ValueError: Fractional ideal (17) is not prime
sage: QuadraticField(-7,'c').galois_group().artin_symbol(13)
(1,2)
sage: G.artin_symbol(K.primes_above(2)[0])
...
ValueError: Fractional ideal (...) is ramified
```

decomposition_group(*P*)

Decomposition group of a prime ideal P, i.e. the subgroup of elements that map P to itself. This is the same as the Galois group of the extension of local fields obtained by completing at P.

This function will raise an error if P is not prime or the given number field is not Galois.

EXAMPLE:

```
sage: K.<a> = NumberField(x^4 - 2*x^2 + 2,'b').galois_closure()
sage: P = K.ideal([17, a^2])
sage: G = K.galois_group()
sage: G.decomposition_group(P)
Subgroup [(), (1,8)(2,7)(3,6)(4,5)] of Galois group of Number Field in a with defining polyr
sage: G.decomposition_group(P^2)
...
ValueError: Fractional ideal (...) is not prime
sage: G.decomposition_group(17)
...
ValueError: Fractional ideal (17) is not prime
```

inertia_group(*P*)

Return the inertia group of the prime P, i.e. the group of elements acting trivially modulo P. This is just the 0th ramification group of P.

EXAMPLE:

```
sage: K.<b> = NumberField(x^2 - 3,'a')
sage: G = K.galois_group()
sage: G.inertia_group(K.primes_above(2)[0])
Galois group of Number Field in b with defining polynomial x^2 - 3
sage: G.inertia_group(K.primes_above(5)[0])
Subgroup [()] of Galois group of Number Field in b with defining polynomial x^2 - 3
```

is_galois()

Return True if the underlying number field of self is actually Galois.

EXAMPLE:

```
sage: NumberField(x^3 - x + 1,'a').galois_group(names='b').is_galois()
False
sage: NumberField(x^2 - x + 1,'a').galois_group().is_galois()
True
```

list()
List of the elements of self.
EXAMPLE:

```
sage: NumberField(x^3 - 3*x + 1,'a').galois_group().list()
[(), (1,2,3), (1,3,2)]
```

ngens()
Number of generators of self.
EXAMPLE:

```
sage: QuadraticField(-23, 'a').galois_group().ngens()
1
```

number_field()
The ambient number field.
EXAMPLE:

```
sage: K = NumberField(x^3 - x + 1, 'a')
sage: K.galois_group(names='b').number_field() is K
True
```

ramification_breaks(*P*)
Return the set of ramification breaks of the prime ideal P, i.e. the set of indices i such that the ramification group $G_{i+1} \neq G_i$. This is only defined for Galois fields.
EXAMPLE:

```
sage: K.<b> = NumberField(x^8 - 20*x^6 + 104*x^4 - 40*x^2 + 1156)
sage: G = K.galois_group()
sage: P = K.primes_above(2)[0]
sage: G.ramification_breaks(P)
{1, 3, 5}
sage: min( [ G.ramification_group(P, i).order() / G.ramification_group(P, i+1).order() for i
2
```

ramification_group(*P*, *v*)
Return the vth ramification group of self for the prime P, i.e. the set of elements s of self such that s acts trivially modulo P^(v+1). This is only defined for Galois fields.
EXAMPLE:

```
sage: K.<b> = NumberField(x^3 - 3,'a').galois_closure()
sage: G=K.galois_group()
sage: P = K.primes_above(3)[0]
sage: G.ramification_group(P, 3)
Subgroup [(), (1,3,6)(2,4,5), (1,6,3)(2,5,4)] of Galois group of Number Field in b with defi
sage: G.ramification_group(P, 5)
Subgroup [()] of Galois group of Number Field in b with defining polynomial x^6 + 60*x^3 + 3
```

splitting_field()
The Galois closure of the ambient number field.
EXAMPLE:

```
sage: K = NumberField(x^3 - x + 1, 'a')
sage: K.galois_group(names='b').splitting_field()
Number Field in b with defining polynomial x^6 - 14*x^4 - 20*x^3 + 49*x^2 + 140*x + 307
sage: L = QuadraticField(-23, 'c'); L.galois_group().splitting_field() is L
True
```

subgroup(*elts*)
Return the subgroup of self with the given elements. Mostly for internal use.

EXAMPLE:

```
sage: G = NumberField(x^3 - x - 1,'a').galois_closure('b').galois_group()
sage: G.subgroup([ G(1), G([(1,5,2),(3,4,6)]), G([(1,2,5),(3,6,4)])])
Subgroup [(), (1,5,2)(3,4,6), (1,2,5)(3,6,4)] of Galois group of Number Field in b with defi
```

26.6 Field of Algebraic Numbers

AUTHOR:

- Carl Witty (2007-01-27): initial version
- Carl Witty (2007-10-29): massive rewrite to support complex as well as real numbers

This is an implementation of the algebraic numbers (the complex numbers which are the zero of a polynomial in $\mathbf{Z}[x]$; in other words, the algebraic closure of $\mathbf{Q}$, with an embedding into $\mathbf{C}$). All computations are exact. We also include an implementation of the algebraic reals (the intersection of the algebraic numbers with $\mathbf{R}$). The field of algebraic numbers $\overline{\mathbf{Q}}$ is available with abbreviation `QQbar`; the field of algebraic reals has abbreviation `AA`.

As with many other implementations of the algebraic numbers, we try hard to avoid computing a number field and working in the number field; instead, we use floating-point interval arithmetic whenever possible (basically whenever we need to prove non-equalities), and resort to symbolic computation only as needed (basically to prove equalities).

Algebraic numbers exist in one of the following forms:

- a rational number
- the product of a rational number and an n'th root of unity
- the sum, difference, product, or quotient of algebraic numbers
- the negation, inverse, absolute value, norm, real part, imaginary part, or complex conjugate of an algebraic number
- a particular root of a polynomial, given as a polynomial with algebraic coefficients together with an isolating interval (given as a `RealIntervalFieldElement`) which encloses exactly one root, and the multiplicity of the root
- a polynomial in one generator, where the generator is an algebraic number given as the root of an irreducible polynomial with integral coefficients and the polynomial is given as a `NumberFieldElement`.

The multiplicative subgroup of the algebraic numbers generated by the rational numbers and the roots of unity is handled particularly efficiently, as long as these roots of unity come from the `QQbar.zeta()` method. Cyclotomic fields in general are fairly efficient, again as long as they are derived from `QQbar.zeta()`.

An algebraic number can be coerced into `ComplexIntervalField` (or `RealIntervalField`, for algebraic reals); every algebraic number has a cached interval of the highest precision yet calculated.

Everything is done with intervals except for comparisons. By default, comparisons compute the two algebraic numbers with 128-bit precision intervals; if this does not suffice to prove that the numbers are different, then we fall back on exact computation.

Note that division involves an implicit comparison of the divisor against zero, and may thus trigger exact computation.

Also, using an algebraic number in the leading coefficient of a polynomial also involves an implicit comparison against zero, which again may trigger exact computation.

Note that we work fairly hard to avoid computing new number fields; to help, we keep a lattice of already-computed number fields and their inclusions.

EXAMPLES:

```
sage: sqrt(AA(2)) > 0
True
sage: (sqrt(5 + 2*sqrt(QQbar(6))) - sqrt(QQbar(3)))^2 == 2
True
sage: AA((sqrt(5 + 2*sqrt(6)) - sqrt(3))^2) == 2
True
```

For a monic cubic polynomial $x^3 + bx^2 + cx + d$ with roots $s1$, $s2$, $s3$, the discriminant is defined as $(s1 - s2)^2(s1 - s3)^2(s2 - s3)^2$ and can be computed as $b^2c^2 - 4b^3d - 4c^3 + 18bcd - 27d^2$. We can test that these definitions do give the same result:

```
sage: def disc1(b, c, d):
...       return b^2*c^2 - 4*b^3*d - 4*c^3 + 18*b*c*d - 27*d^2
sage: def disc2(s1, s2, s3):
...       return ((s1-s2)*(s1-s3)*(s2-s3))^2
sage: x = polygen(AA)
sage: p = x*(x-2)*(x-4)
sage: cp = AA.common_polynomial(p)
sage: d, c, b, _ = p.list()
sage: s1 = AA.polynomial_root(cp, RIF(-1, 1))
sage: s2 = AA.polynomial_root(cp, RIF(1, 3))
sage: s3 = AA.polynomial_root(cp, RIF(3, 5))
sage: disc1(b, c, d) == disc2(s1, s2, s3)
True
sage: p = p + 1
sage: cp = AA.common_polynomial(p)
sage: d, c, b, _ = p.list()
sage: s1 = AA.polynomial_root(cp, RIF(-1, 1))
sage: s2 = AA.polynomial_root(cp, RIF(1, 3))
sage: s3 = AA.polynomial_root(cp, RIF(3, 5))
sage: disc1(b, c, d) == disc2(s1, s2, s3)
True
sage: p = (x-sqrt(AA(2)))*(x-AA(2).nth_root(3))*(x-sqrt(AA(3)))
sage: cp = AA.common_polynomial(p)
sage: d, c, b, _ = p.list()
sage: s1 = AA.polynomial_root(cp, RIF(1.4, 1.5))
sage: s2 = AA.polynomial_root(cp, RIF(1.7, 1.8))
sage: s3 = AA.polynomial_root(cp, RIF(1.2, 1.3))
sage: disc1(b, c, d) == disc2(s1, s2, s3)
True
```

We can coerce from symbolic expressions:

```
sage: QQbar(sqrt(-5))
2.236067977499790?*I
sage: AA(sqrt(2) + sqrt(3))
3.146264369941973?
sage: AA(sqrt(2)) + sqrt(3)
3.146264369941973?
sage: sqrt(2) + QQbar(sqrt(3))
3.146264369941973?
sage: QQbar(I)
1*I
sage: AA(I)
...
TypeError: Illegal initializer for algebraic number
```

```
sage: QQbar(I * golden_ratio)
1.618033988749895?*I
sage: AA(golden_ratio)^2 - AA(golden_ratio)
1
sage: QQbar((-8)^(1/3))
1.000000000000000? + 1.732050807568878?*I
sage: AA((-8)^(1/3))
-2
sage: QQbar((-4)^(1/4))
1 + 1*I
sage: AA((-4)^(1/4))
...
ValueError: Cannot coerce algebraic number with non-zero imaginary part to algebraic real
```

Note the different behavior in taking roots: for `AA` we prefer real roots if they exist, but for `QQbar` we take the principal root:

```
sage: AA(-1)^(1/3)
-1
sage: QQbar(-1)^(1/3)
0.500000000000000? + 0.866025403784439?*I
```

We can explicitly coerce from $\mathbf{Q}[I]$. (Technically, this is not quite kosher, since $\mathbf{Q}[I]$ doesn't come with an embedding; we do not know whether the field generator is supposed to map to $+I$ or $-I$. We assume that for any quadratic field with polynomial $x^2 + 1$, the generator maps to $+I$.):

```
sage: K.<im> = QQ[I]
sage: pythag = QQbar(3/5 + 4*im/5); pythag
4/5*I + 3/5
sage: pythag.abs() == 1
True
```

However, implicit coercion from $\mathbf{Q}[I]$ is not allowed:

```
sage: QQbar(1) + im
...
TypeError: unsupported operand parent(s) for '+': 'Algebraic Field' and 'Number Field in I with defi
```

We can implicitly coerce from algebraic reals to algebraic numbers:

```
sage: a = QQbar(1); print a, a.parent()
1 Algebraic Field
sage: b = AA(1); print b, b.parent()
1 Algebraic Real Field
sage: c = a + b; print c, c.parent()
2 Algebraic Field
```

Some computation with radicals:

```
sage: phi = (1 + sqrt(AA(5))) / 2
sage: phi^2 == phi + 1
True
sage: tau = (1 - sqrt(AA(5))) / 2
sage: tau^2 == tau + 1
True
sage: phi + tau == 1
```

```
True
sage: tau < 0
True

sage: rt23 = sqrt(AA(2/3))
sage: rt35 = sqrt(AA(3/5))
sage: rt25 = sqrt(AA(2/5))
sage: rt23 * rt35 == rt25
True
```

The Sage rings `AA` and `QQbar` can decide equalities between radical expressions (over the reals and complex numbers respectively):

```
sage: a = AA((2/(3*sqrt(3)) + 10/27)^(1/3) - 2/(9*(2/(3*sqrt(3)) + 10/27)^(1/3)) + 1/3)
sage: a
1.000000000000000?
sage: a == 1
True
```

Algebraic numbers which are known to be rational print as rationals; otherwise they print as intervals (with 53-bit precision):

```
sage: AA(2)/3
2/3
sage: QQbar(5/7)
5/7
sage: QQbar(1/3 - 1/4*I)
-1/4*I + 1/3
sage: two = QQbar(4).nth_root(4)^2; two
2.000000000000000?
sage: two == 2; two
True
2
sage: phi
1.618033988749895?
```

We can find the real and imaginary parts of an algebraic number (exactly):

```
sage: r = QQbar.polynomial_root(x^5 - x - 1, CIF(RIF(0.1, 0.2), RIF(1.0, 1.1))); r
0.1812324444698754? + 1.083954101317711?*I
sage: r.real()
0.1812324444698754?
sage: r.imag()
1.083954101317711?
sage: r.minpoly()
x^5 - x - 1
sage: r.real().minpoly()
x^10 + 3/16*x^6 + 11/32*x^5 - 1/64*x^2 + 1/128*x - 1/1024
sage: r.imag().minpoly() # this takes a long time (143s on my laptop)
x^20 - 5/8*x^16 - 95/256*x^12 - 625/1024*x^10 - 5/512*x^8 - 1875/8192*x^6 + 25/4096*x^4 - 625/32768*x
```

We can find the absolute value and norm of an algebraic number exactly. (Note that we define the norm as the product of a number and its complex conjugate; this is the algebraic definition of norm, if we view `QQbar` as `AA[I]`.):

```
sage: R.<x> = QQ[]
sage: r = (x^3 + 8).roots(QQbar, multiplicities=False)[2]; r
```

```
1.000000000000000? + 1.732050807568878?*I
sage: r.abs() == 2
True
sage: r.norm() == 4
True
sage: (r+I).norm().minpoly()
x^2 - 10*x + 13
sage: r = AA.polynomial_root(x^2 - x - 1, RIF(-1, 0)); r
-0.6180339887498895?
sage: r.abs().minpoly()
x^2 + x - 1
```

We can compute the multiplicative order of an algebraic number:

```
sage: QQbar(-1/2 + I*sqrt(3)/2).multiplicative_order()
3
sage: QQbar(-sqrt(3)/2 + I/2).multiplicative_order()
12
sage: QQbar.zeta(12345).multiplicative_order()
12345
```

Cyclotomic fields are very fast as long as we only multiply and divide:

```
sage: z3_3 = QQbar.zeta(3) * 3
sage: z4_4 = QQbar.zeta(4) * 4
sage: z5_5 = QQbar.zeta(5) * 5
sage: z6_6 = QQbar.zeta(6) * 6
sage: z20_20 = QQbar.zeta(20) * 20
sage: z3_3 * z4_4 * z5_5 * z6_6 * z20_20
7200
```

And they are still pretty fast even if you add and subtract, and trigger exact computation:

```
sage: (z3_3 + z4_4 + z5_5 + z6_6 + z20_20)._exact_value()
4*zeta60^15 + 5*zeta60^12 + 9*zeta60^10 + 20*zeta60^3 - 3 where a^16 + a^14 - a^10 - a^8 - a^6 + a^2
```

The paper "ARPREC: An Arbitrary Precision Computation Package" by Bailey, Yozo, Li and Thompson discusses this result. Evidently it is difficult to find, but we can easily verify it.

```
sage: alpha = QQbar.polynomial_root(x^10 + x^9 - x^7 - x^6 - x^5 - x^4 - x^3 + x + 1, RIF(1, 1.2))
sage: lhs = alpha^630 - 1
sage: rhs_num = (alpha^315 - 1) * (alpha^210 - 1) * (alpha^126 - 1)^2 * (alpha^90 - 1) * (alpha^3 - 1
sage: rhs_den = (alpha^35 - 1) * (alpha^15 - 1)^2 * (alpha^14 - 1)^2 * (alpha^5 - 1)^6 * alpha^68
sage: rhs = rhs_num / rhs_den
sage: lhs
2.642040335819351?e44
sage: rhs
2.642040335819351?e44
sage: lhs - rhs
0.?e29
sage: lhs == rhs
True
sage: lhs - rhs
0
sage: lhs._exact_value()
-2424946098563164022648228330623508477694745 40*a^9 + 8622954720682894724916548377859479062346807 03*a
```

Given an algebraic number, we can produce a string that will reproduce that algebraic number if you type the string into Sage. We can see that until exact computation is triggered, an algebraic number keeps track of the computation steps used to produce that number:

```
sage: rt2 = AA(sqrt(2))
sage: rt3 = AA(sqrt(3))
sage: n = (rt2 + rt3)^5; n
308.30180017229752?
sage: sage_input(n)
v1 = sqrt(AA(2)) + sqrt(AA(3))
v2 = v1*v1
v2*v2*v1
```

But once exact computation is triggered, the computation tree is discarded, and we get a way to produce the number directly:

```
sage: n == 109*rt2 + 89*rt3
True
sage: sage_input(n)
R.<x> = AA[]
v = AA.polynomial_root(AA.common_polynomial(x^4 - 4*x^2 + 1), RIF(RR(0.51763809020504148), RR(0.5176
-109*v^3 - 89*v^2 + 327*v + 178
```

We can also see that some computations (basically, those which are easy to perform exactly) are performed directly, instead of storing the computation tree:

```
sage: z3_3 = QQbar.zeta(3) * 3
sage: z4_4 = QQbar.zeta(4) * 4
sage: z5_5 = QQbar.zeta(5) * 5
sage: sage_input(z3_3 * z4_4 * z5_5)
-60*QQbar.zeta(60)^17
```

Note that the `verify=True` argument to `sage_input` will always trigger exact computation, so running `sage_input` twice in a row on the same number will actually give different answers. In the following, running `sage_input` on `n` will also trigger exact computation on `rt2`, as you can see by the fact that the third output is different than the first:

```
sage: rt2 = AA(sqrt(2))
sage: n = rt2^2
sage: sage_input(n, verify=True)
# Verified
v = sqrt(AA(2))
v*v
sage: sage_input(n, verify=True)
# Verified
AA(2)
sage: n = rt2^2
sage: sage_input(n, verify=True)
# Verified
AA(2)
```

Just for fun, let's try `sage_input` on a very complicated expression:

```
sage: rt2 = sqrt(AA(2))
sage: rt3 = sqrt(QQbar(3))
sage: x = polygen(QQbar)
```

```
sage: nrt3 = AA.polynomial_root((x-rt2)*(x+rt3), RIF(-2, -1))
sage: one = AA.polynomial_root((x-rt2)*(x-rt3)*(x-nrt3)*(x-1-rt3-nrt3), RIF(0.9, 1.1))
sage: one
1.000000000000000?
sage: sage_input(one, verify=True)
# Verified
R.<x> = QQbar[]
v1 = AA(2)
v2 = QQbar(sqrt(v1))
v3 = QQbar(3)
v4 = sqrt(v3)
v5 = v2*v4
v6 = (1 - v2)*(1 - v4) - 1 - v5
v7 = QQbar(sqrt(v1))
v8 = sqrt(v3)
si1 = v7*v8
cp = AA.common_polynomial(x^2 + ((1 - v7)*(1 + v8) - 1 + si1)*x - si1)
v9 = QQbar.polynomial_root(cp, RIF(-RR(1.7320508075688774), -RR(1.7320508075688772)))
v10 = 1 - v9
v11 = v6 + (v10 - 1)
v12 = -1 - v4 - QQbar.polynomial_root(cp, RIF(-RR(1.7320508075688774), -RR(1.7320508075688772)))
v13 = 1 + v12
v14 = v10*(v6 + v5) - (v6 - v5*v9)
si2 = v5*v9
AA.polynomial_root(AA.common_polynomial(x^4 + (v11 + (v13 - 1))*x^3 + (v14 + (v13*v11 - v11))*x^2 +
sage: one
1
```

We can pickle and unpickle algebraic fields (and they are globally unique):

```
sage: loads(dumps(AlgebraicField())) is AlgebraicField()
True
sage: loads(dumps(AlgebraicRealField())) is AlgebraicRealField()
True
```

We can pickle and unpickle algebraic numbers:

```
sage: loads(dumps(QQbar(10))) == QQbar(10)
True
sage: loads(dumps(QQbar(5/2))) == QQbar(5/2)
True
sage: loads(dumps(QQbar.zeta(5))) == QQbar.zeta(5)
True

sage: t = QQbar(sqrt(2)); type(t._descr)
<class 'sage.rings.qqbar.ANRoot'>
sage: loads(dumps(t)) == QQbar(sqrt(2))
True

sage: t.exactify(); type(t._descr)
<class 'sage.rings.qqbar.ANExtensionElement'>
sage: loads(dumps(t)) == QQbar(sqrt(2))
True

sage: t = ~QQbar(sqrt(2)); type(t._descr)
<class 'sage.rings.qqbar.ANUnaryExpr'>
sage: loads(dumps(t)) == 1/QQbar(sqrt(2))
```

```
True

sage: t = QQbar(sqrt(2)) + QQbar(sqrt(3)); type(t._descr)
<class 'sage.rings.qqbar.ANBinaryExpr'>
sage: loads(dumps(t)) == QQbar(sqrt(2)) + QQbar(sqrt(3))
True
```

We can convert elements of QQbar and AA into the following types: float, complex, RDF, CDF, RR, CC, RIF, CIF, ZZ, and QQ, with a few exceptions. (For the arbitrary-precision types, RR, CC, RIF, and CIF, it can convert into a field of arbitrary precision.)

Converting from QQbar to a real type (float, RDF, RR, RIF, ZZ, or QQ) succeeds only if the QQbar is actually real (has an imaginary component of exactly zero). Converting from either AA or QQbar to ZZ or QQ succeeds only if the number actually is an integer or rational. If conversion fails, a ValueError will be raised.

Here are examples of all of these conversions:

```
sage: all_vals = [AA(42), AA(22/7), AA(golden_ratio), QQbar(-13), QQbar(89/55), QQbar(-sqrt(7)), QQba
sage: def convert_test_all(ty):
...       def convert_test(v):
...           try:
...               return ty(v)
...           except ValueError:
...               return None
...       return map(convert_test, all_vals)
sage: convert_test_all(float)
[42.0, 3.1428571428571432, 1.6180339887498949, -13.0, 1.6181818181818182, -2.6457513110645907, None]
sage: convert_test_all(complex)
[(42+0j), (3.1428571428571432+0j), (1.6180339887498949+0j), (-13+0j), (1.6181818181818182+0j), (-2.64
sage: convert_test_all(RDF)
[42.0, 3.14285714286, 1.61803398875, -13.0, 1.61818181818, -2.64575131106, None]
sage: convert_test_all(CDF)
[42.0, 3.14285714286, 1.61803398875, -13.0, 1.61818181818, -2.64575131106, 0.309016994375 + 0.951056
sage: convert_test_all(RR)
[42.0000000000000, 3.14285714285714, 1.61803398874989, -13.0000000000000, 1.61818181818182, -2.645751
sage: convert_test_all(CC)
[42.0000000000000, 3.14285714285714, 1.61803398874989, -13.0000000000000, 1.61818181818182, -2.645751
sage: convert_test_all(RIF)
[42, 3.142857142857143?, 1.618033988749895?, -13, 1.618181818181819?, -2.645751311064591?, None]
sage: convert_test_all(CIF)
[42, 3.142857142857143?, 1.618033988749895?, -13, 1.618181818181819?, -2.645751311064591?, 0.309016994
sage: convert_test_all(ZZ)
[42, None, None, -13, None, None, None]
sage: convert_test_all(QQ)
[42, 22/7, None, -13, 89/55, None, None]
```

class ANBinaryExpr (*left, right, op*)

exactify()

TESTS:

We check to make sure that this method still works even. We do this by increasing the recursion level at each step and decrease it before we return:

```
sage: import sys; sys.getrecursionlimit()
1000
sage: s = SFASchur(QQ)
sage: a=s([3,2]).expand(8)(flatten([[QQbar.zeta(3)^d for d in range(3)], [QQbar.zeta(5)^d fo
```

```
sage: a.exactify(); a #long
0
sage: sys.getrecursionlimit()
1000
```

handle_sage_input(*sib, coerce, is_qqbar*)

Produce an expression which will reproduce this value when evaluated, and an indication of whether this value is worth sharing (always True for ANBinaryExpr).

EXAMPLES:

```
sage: sage_input(2 + sqrt(AA(2)), verify=True)
# Verified
2 + sqrt(AA(2))
sage: sage_input(sqrt(AA(2)) + 2, verify=True)
# Verified
sqrt(AA(2)) + 2
sage: sage_input(2 - sqrt(AA(2)), verify=True)
# Verified
2 - sqrt(AA(2))
sage: sage_input(2 / sqrt(AA(2)), verify=True)
# Verified
2/sqrt(AA(2))
sage: sage_input(2 + (-1*sqrt(AA(2))), verify=True)
# Verified
2 - sqrt(AA(2))
sage: sage_input(2*sqrt(AA(2)), verify=True)
# Verified
2*sqrt(AA(2))
sage: rt2 = sqrt(AA(2))
sage: one = rt2/rt2
sage: n = one+3
sage: sage_input(n)
v = sqrt(AA(2))
v/v + 3
sage: one == 1
True
sage: sage_input(n)
1 + AA(3)
sage: rt3 = QQbar(sqrt(3))
sage: one = rt3/rt3
sage: n = sqrt(AA(2))+one
sage: one == 1
True
sage: sage_input(n)
QQbar(sqrt(AA(2))) + 1
sage: from sage.rings.qqbar import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: binexp = ANBinaryExpr(AA(3), AA(5), '*')
sage: binexp.handle_sage_input(sib, False, False)
({binop:* {atomic:3} {call: {atomic:AA}({atomic:5})}}, True)
sage: binexp.handle_sage_input(sib, False, True)
({call: {atomic:QQbar}({binop:* {atomic:3} {call: {atomic:AA}({atomic:5})}})}, True)
```

is_complex()

kind()

class **ANDescr**()

An AlgebraicNumber or AlgebraicReal is a wrapper around an ANDescr object. ANDescr is an

abstract base class, which should never be directly instantiated; its concrete subclasses are ANRational, ANBinaryExpr, ANUnaryExpr, ANRootOfUnity, ANRoot, and ANExtensionElement. ANDescr and all of its subclasses are private, and should not be used directly.

abs(*n*)

conjugate(*n*)

imag(*n*)

invert(*n*)

is_exact()

Returns True if self is an ANRational, ANRootOfUnity, or ANExtensionElement.

EXAMPLES:

```
sage: from sage.rings.qqbar import ANRational
sage: ANRational(1/2).is_exact()
True
sage: QQbar(3+I)._descr.is_exact()
True
sage: QQbar.zeta(17)._descr.is_exact()
True
```

is_field_element()

Returns True if self is an ANExtensionElement.

EXAMPLES:

```
sage: from sage.rings.qqbar import ANExtensionElement, ANRoot, AlgebraicGenerator
sage: _.<y> = QQ['y']
sage: x = polygen(QQbar)
sage: nf2 = NumberField(y^2 - 2, name='a', check=False)
sage: root2 = ANRoot(x^2 - 2, RIF(1, 2))
sage: gen2 = AlgebraicGenerator(nf2, root2)
sage: sqrt2 = ANExtensionElement(gen2, nf2.gen())
sage: sqrt2.is_field_element()
True
```

is_rational()

Returns True if self is an ANRational object. (Note that the constructors for ANExtensionElement and ANRootOfUnity will actually return ANRational objects for rational numbers.)

EXAMPLES:

```
sage: from sage.rings.qqbar import ANRational
sage: ANRational(3/7).is_rational()
True
```

is_simple()

Checks whether this descriptor represents a value with the same algebraic degree as the number field associated with the descriptor.

Returns True if self is an ANRational, ANRootOfUnit, or a minimal ANExtensionElement.

EXAMPLES:

```
sage: from sage.rings.qqbar import ANRational
sage: ANRational(1/2).is_simple()
True
sage: rt2 = AA(sqrt(2))
sage: rt3 = AA(sqrt(3))
sage: rt2b = rt3 + rt2 - rt3
sage: rt2.exactify()
```

```
sage: rt2._descr.is_simple()
True
sage: rt2b.exactify()
sage: rt2b._descr.is_simple()
False
sage: rt2b.simplify()
sage: rt2b._descr.is_simple()
True
```

neg(*n*)

norm(*n*)

real(*n*)

class **ANExtensionElement**(*generator, value*)

The subclass of ANDescr that represents a number field element in terms of a specific generator. Consists of a polynomial with rational coefficients in terms of the generator, and the generator itself, an AlgebraicGenerator.

abs(*n*)

conjugate(*n*)

exactify()

field_element_value()

gaussian_value()

generator()

handle_sage_input(*sib, coerce, is_qqbar*)

Produce an expression which will reproduce this value when evaluated, and an indication of whether this value is worth sharing (always True, for ANExtensionElement).

EXAMPLES:

```
sage: I = QQbar(I)
sage: sage_input(3+4*I, verify=True)
# Verified
QQbar(3 + 4*I)
sage: v = QQbar.zeta(3) + QQbar.zeta(5)
sage: v - v == 0
True
sage: sage_input(vector(QQbar, (4-3*I, QQbar.zeta(7))), verify=True)
# Verified
vector(QQbar, [4 - 3*I, QQbar.zeta(7)])
sage: sage_input(v, verify=True)
# Verified
v = QQbar.zeta(15)
v^5 + v^3
sage: v = QQbar(sqrt(AA(2)))
sage: v.exactify()
sage: sage_input(v, verify=True)
# Verified
R.<x> = AA[]
QQbar(AA.polynomial_root(AA.common_polynomial(x^2 - 2), RIF(RR(1.4142135623730949), RR(1.414
sage: from sage.rings.qqbar import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: extel = ANExtensionElement(QQbar_I_generator, QQbar_I_generator.field().gen() + 1)
sage: extel.handle_sage_input(sib, False, True)
({call: {atomic:QQbar}({binop:+ {atomic:1} {atomic:I}})}, True)
```

invert(*n*)

is_complex()

is_exact()

is_field_element()

is_simple()

Checks whether this descriptor represents a value with the same algebraic degree as the number field associated with the descriptor.

For `ANExtensionElement` elements, we check this by comparing the degree of the minimal polynomial to the degree of the field.

EXAMPLES:

```
sage: rt2 = AA(sqrt(2))
sage: rt3 = AA(sqrt(3))
sage: rt2b = rt3 + rt2 - rt3
sage: rt2.exactify()
sage: rt2._descr
a where a^2 - 2 = 0 and a in 1.414213562373095?
sage: rt2._descr.is_simple()
True

sage: rt2b.exactify()
sage: rt2b._descr
a^3 - 3*a where a^4 - 4*a^2 + 1 = 0 and a in 1.931851652578137?
sage: rt2b._descr.is_simple()
False
```

kind()

minpoly()

Compute the minimal polynomial of this algebraic number.

EXAMPLES:

```
sage: a = AA(sqrt(2)) + QQbar(I); a
1.414213562373095? + 1*I
sage: p = a.minpoly(); p
x^4 - 2*x^2 + 9
sage: p(a)
0
```

neg(*n*)

norm(*n*)

rational_argument(*n*)

If the argument of self is 2π times some rational number, return that rational; otherwise, return None.

simplify(*n*)

Compute an exact representation for this descriptor, in the smallest possible number field.

INPUT:

- n – The element of `AA` or `QQbar` corresponding to this descriptor.

EXAMPLES:

```
sage: rt2 = AA(sqrt(2))
sage: rt3 = AA(sqrt(3))
sage: rt2b = rt3 + rt2 - rt3
sage: rt2b.exactify()
sage: rt2b._descr
a^3 - 3*a where a^4 - 4*a^2 + 1 = 0 and a in 1.931851652578137?
```

```
sage: rt2b._descr.simplify(rt2b)
a where a^2 - 2 = 0 and a in 1.414213562373095?
```

class ANRational(*x*)

The subclass of ANDescr that represents an arbitrary rational. This class is private, and should not be used directly.

abs(*n*)

angle()

exactify()

gaussian_value()

generator()

handle_sage_input(*sib, coerce, is_qqbar*)

Produce an expression which will reproduce this value when evaluated, and an indication of whether this value is worth sharing (always False, for rationals).

EXAMPLES:

```
sage: sage_input(QQbar(22/7), verify=True)
# Verified
QQbar(22/7)
sage: sage_input(-AA(3)/5, verify=True)
# Verified
AA(-3/5)
sage: sage_input(vector(AA, (0, 1/2, 1/3)), verify=True)
# Verified
vector(AA, [0, 1/2, 1/3])
sage: from sage.rings.qqbar import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: rat = ANRational(9/10)
sage: rat.handle_sage_input(sib, False, True)
({call: {atomic:QQbar}({binop:/ {atomic:9} {atomic:10}})}, False)
```

invert(*n*)

is_complex()

is_exact()

is_rational()

is_simple()

Checks whether this descriptor represents a value with the same algebraic degree as the number field associated with the descriptor.

This is always true for rational numbers.

EXAMPLES:

```
sage: AA(1/2)._descr.is_simple()
True
```

kind()

minpoly()

neg(*n*)

rational_argument(*n*)

rational_value()

scale()

class ANRoot (*poly, interval, multiplicity=1, is_pow=None*)

The subclass of ANDescr that represents a particular root of a polynomial with algebraic coefficients. This class is private, and should not be used directly.

conjugate (*n*)

exactify ()

Returns either an ANRational or an ANExtensionElement with the same value as this number.

EXAMPLES:

```
sage: from sage.rings.qqbar import ANRoot
sage: x = polygen(QQbar)
sage: two = ANRoot((x-2)*(x-sqrt(QQbar(2))), RIF(1.9, 2.1))
sage: two.exactify()
2
sage: two.exactify().rational_value()
2
sage: strange = ANRoot(x^2 + sqrt(QQbar(3))*x - sqrt(QQbar(2)), RIF(-0, 1))
sage: strange.exactify()
a where a^8 - 6*a^6 + 5*a^4 - 12*a^2 + 4 = 0 and a in 0.6051012265139511?
```

handle_sage_input (*sib, coerce, is_qqbar*)

Produce an expression which will reproduce this value when evaluated, and an indication of whether this value is worth sharing (always True, for ANRoot).

EXAMPLES:

```
sage: sage_input((AA(3)^(1/2))^(1/3), verify=True)
# Verified
sqrt(AA(3)).nth_root(3)
```

These two examples are too big to verify quickly. (Verification would create a field of degree 28.):

```
sage: sage_input((sqrt(AA(3))^(5/7))^(9/4))
(sqrt(AA(3))^(5/7))^(9/4)
sage: sage_input((sqrt(QQbar(-7))^(5/7))^(9/4))
(sqrt(QQbar(-7))^(5/7))^(9/4)
sage: x = polygen(QQ)
sage: sage_input(AA.polynomial_root(x^2-x-1, RIF(1, 2)), verify=True)
# Verified
R.<x> = AA[]
AA.polynomial_root(AA.common_polynomial(x^2 - x - 1), RIF(RR(1.6180339887498947), RR(1.61803
sage: sage_input(QQbar.polynomial_root(x^3-5, CIF(RIF(-3, 0), RIF(0, 3))), verify=True)
# Verified
R.<x> = AA[]
QQbar.polynomial_root(AA.common_polynomial(x^3 - 5), CIF(RIF(-RR(0.85498797333834853), -RR(0
sage: from sage.rings.qqbar import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: rt = ANRoot(x^3 - 2, RIF(0, 4))
sage: rt.handle_sage_input(sib, False, True)
({call: {getattr: {atomic:QQbar}.polynomial_root}({call: {getattr: {atomic:AA}.common_polync
```

is_complex ()

kind ()

refine_interval (*interval, prec*)

class ANRootOfUnity (*angle, scale*)

The subclass of ANDescr that represents a rational multiplied by a root of unity. This class is private, and should not be used directly.

Such numbers are represented by a "rational angle" and a rational scale. The "rational angle" is the argument of the number, divided by 2π; so given angle α and scale s, the number is: $s(\cos(2\pi\alpha) + \sin(2\pi\alpha)i)$; or equivalently $s(e^{2\pi\alpha i})$.

We normalize so that $0 < \alpha < \frac{1}{2}$; this requires allowing both positive and negative scales. (Attempts to create an `ANRootOfUnity` with an angle which is a multiple of $\frac{1}{2}$ end up creating an `ANRational` instead.)

abs(*n*)

angle()

conjugate(*n*)

exactify()

field_element_value()

gaussian_value()

generator()

handle_sage_input(*sib, coerce, is_qqbar*)

Produce an expression which will reproduce this value when evaluated, and an indication of whether this value is worth sharing (False for imaginary numbers, True for others).

EXAMPLES:

```
sage: sage_input(22/7*QQbar.zeta(4), verify=True)
# Verified
QQbar(22/7*I)
sage: sage_input((2*QQbar.zeta(12))^4, verify=True)
# Verified
16*QQbar.zeta(3)
sage: sage_input(QQbar.zeta(5)^2, verify=True)
# Verified
QQbar.zeta(5)^2
sage: sage_input(QQbar.zeta(5)^3, verify=True)
# Verified
-QQbar.zeta(10)
sage: sage_input(vector(QQbar, (I, 3*QQbar.zeta(9))), verify=True)
# Verified
vector(QQbar, [I, 3*QQbar.zeta(9)])
sage: from sage.rings.qqbar import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: rtofunity = ANRootOfUnity(137/500, 1/1000)
sage: rtofunity.handle_sage_input(sib, False, True)
({binop:* {binop:/ {atomic:1} {atomic:1000}} {binop:** {call: {getattr: {atomic:QQbar}.zeta}
```

invert(*n*)

is_complex()

is_exact()

is_simple()

Checks whether this descriptor represents a value with the same algebraic degree as the number field associated with the descriptor.

This is always true for `ANRootOfUnity` elements.

EXAMPLES:

```
sage: a = QQbar.zeta(17)^5 * 4/3; a._descr
4/3*e^(2*pi*I*5/17)
sage: a._descr.is_simple()
True
```

kind()

minpoly()

EXAMPLES:

```
sage: a = QQbar.zeta(7) * 2; a
1.246979603717467? + 1.563662964936060?*I
sage: a.minpoly()
x^6 + 2*x^5 + 4*x^4 + 8*x^3 + 16*x^2 + 32*x + 64
sage: a.minpoly()(a)
0.?e-15 + 0.?e-15*I
sage: a.minpoly()(a) == 0
True
```

neg(*n*)

norm(*n*)

rational_argument(*n*)

scale()

class **ANUnaryExpr**(*arg*, *op*)

exactify()

handle_sage_input(*sib*, *coerce*, *is_qqbar*)

Produce an expression which will reproduce this value when evaluated, and an indication of whether this value is worth sharing (always True for `ANUnaryExpr`).

EXAMPLES:

```
sage: sage_input(-sqrt(AA(2)), verify=True)
# Verified
-sqrt(AA(2))
sage: sage_input(~sqrt(AA(2)), verify=True)
# Verified
~sqrt(AA(2))
sage: sage_input(sqrt(QQbar(-3)).conjugate(), verify=True)
# Verified
sqrt(QQbar(-3)).conjugate()
sage: sage_input(QQbar.zeta(3).real(), verify=True)
# Verified
QQbar.zeta(3).real()
sage: sage_input(QQbar.zeta(3).imag(), verify=True)
# Verified
QQbar.zeta(3).imag()
sage: sage_input(abs(sqrt(QQbar(-3))), verify=True)
# Verified
abs(sqrt(QQbar(-3)))
sage: sage_input(sqrt(QQbar(-3)).norm(), verify=True)
# Verified
sqrt(QQbar(-3)).norm()
sage: sage_input(QQbar(QQbar.zeta(3).real()), verify=True)
# Verified
QQbar(QQbar.zeta(3).real())
sage: from sage.rings.qqbar import *
sage: from sage.misc.sage_input import SageInputBuilder
sage: sib = SageInputBuilder()
sage: unexp = ANUnaryExpr(sqrt(AA(2)), '~')
sage: unexp.handle_sage_input(sib, False, False)
({unop:~ {call: {atomic:sqrt}({call: {atomic:AA}({atomic:2})})}}, True)
```

```
sage: unexp.handle_sage_input(sib, False, True)
({call: {atomic:QQbar}({unop:~ {call: {atomic:sqrt}({call: {atomic:AA}({atomic:2})})}})}, Tr
```

is_complex()

kind()

class **AlgebraicField**()

The field of algebraic numbers.

algebraic_closure()

EXAMPLES:

```
sage: QQbar.algebraic_closure()
Algebraic Field
```

completion(*p, prec, extras={}*)

EXAMPLES:

```
sage: QQbar.completion(infinity, 500)
Complex Field with 500 bits of precision
sage: QQbar.completion(infinity, prec=53, extras={'type':'RDF'})
Complex Double Field
sage: QQbar.completion(infinity, 53) is CC
True
sage: QQbar.completion(3, 20)
...
NotImplementedError
```

gen(*n=0*)

gens()

has_coerce_map_from_impl(*from_par*)

is_atomic_repr()

ngens()

polynomial_root(*poly, interval, multiplicity=1*)

Given a polynomial with algebraic coefficients and an interval enclosing exactly one root of the polynomial, constructs an algebraic real representation of that root.

The polynomial need not be irreducible, or even squarefree; but if the given root is a multiple root, its multiplicity must be specified. (IMPORTANT NOTE: Currently, multiplicity-k roots are handled by taking the $(k-1)$-st derivative of the polynomial. This means that the interval must enclose exactly one root of this derivative.)

The conditions on the arguments (that the interval encloses exactly one root, and that multiple roots match the given multiplicity) are not checked; if they are not satisfied, an error may be thrown (possibly later, when the algebraic number is used), or wrong answers may result.

Note that if you are constructing multiple roots of a single polynomial, it is better to use `QQbar.common_polynomial` to get a shared polynomial.

EXAMPLES:

```
sage: x = polygen(QQbar)
sage: phi = QQbar.polynomial_root(x^2 - x - 1, RIF(0, 2)); phi
1.618033988749895?
sage: p = (x-1)^7 * (x-2)
sage: r = QQbar.polynomial_root(p, RIF(9/10, 11/10), multiplicity=7)
sage: r; r == 1
1
True
sage: p = (x-phi)*(x-sqrt(QQbar(2)))
```

```
sage: r = QQbar.polynomial_root(p, RIF(1, 3/2))
sage: r; r == sqrt(QQbar(2))
1.4142135623730951?
True
```

zeta(*n=4*)

Returns a primitive n'th root of unity, specifically $\exp(2 * \pi * i/n)$.

EXAMPLES:

```
sage: QQbar.zeta(1)
1
sage: QQbar.zeta(2)
-1
sage: QQbar.zeta(3)
-0.500000000000000? + 0.866025403784439?*I
sage: QQbar.zeta(4)
1*I
sage: QQbar.zeta()
1*I
sage: QQbar.zeta(5)
0.3090169943749474? + 0.9510565162951536?*I
sage: QQbar.zeta(314159)
0.9999999997999997? + 0.00002000001689195824?*I
```

class **AlgebraicField_common**()

characteristic()

common_polynomial(*poly*)

Given a polynomial with algebraic coefficients, returns a wrapper that caches high-precision calculations and factorizations. This wrapper can be passed to polynomial_root in place of the polynomial.

Using `common_polynomial` makes no semantic difference, but will improve efficiency if you are dealing with multiple roots of a single polynomial.

EXAMPLES:

```
sage: x = polygen(ZZ)
sage: p = AA.common_polynomial(x^2 - x - 1)
sage: phi = AA.polynomial_root(p, RIF(1, 2))
sage: tau = AA.polynomial_root(p, RIF(-1, 0))
sage: phi + tau == 1
True
sage: phi * tau == -1
True

sage: x = polygen(SR)
sage: p = (x - sqrt(-5)) * (x - sqrt(3)); p
x^2 + (-sqrt(-5) - sqrt(3))*x + sqrt(-5)*sqrt(3)
sage: p = QQbar.common_polynomial(p)
sage: a = QQbar.polynomial_root(p, CIF(RIF(-0.1, 0.1), RIF(2, 3))); a
0.?e-18 + 2.236067977499790?*I
sage: b = QQbar.polynomial_root(p, RIF(1, 2)); b
1.732050807568878?
```

These "common polynomials" can be shared between real and complex roots:

```
sage: p = AA.common_polynomial(x^3 - x - 1)
sage: r1 = AA.polynomial_root(p, RIF(1.3, 1.4)); r1
1.3247179572447467?
```

```
sage: r2 = QQbar.polynomial_root(p, CIF(RIF(-0.7, -0.6), RIF(0.5, 0.6))); r2
-0.6623589786223730? + 0.5622795120623013?*I
```

default_interval_prec()

is_finite()

order()

class AlgebraicGenerator(*field, root*)

An `AlgebraicGenerator` represents both an algebraic number α and the number field $\mathbf{Q}[\alpha]$. There is a single `AlgebraicGenerator` representing $\mathbf{Q}$ (with $\alpha = 0$).

The `AlgebraicGenerator` class is private, and should not be used directly.

conjugate()

If this generator is for the algebraic number α, return a generator for the complex conjugate of α.

field()

is_complex()

is_trivial()

Returns true iff this is the trivial generator (alpha == 1), which does not actually extend the rationals.

EXAMPLES:

```
sage: from sage.rings.qqbar import qq_generator
sage: qq_generator.is_trivial()
True
```

pari_field()

root_as_algebraic()

set_cyclotomic(*n*)

super_poly(*super, checked=None*)

Given a generator `gen` and another generator `super`, where `super` is the result of a tree of `union()` operations where one of the leaves is `gen`, `gen.super_poly(super)` returns a polynomial expressing the value of `gen` in terms of the value of `super` (except that if `gen` is `qq_generator`, `super_poly()` always returns None.)

EXAMPLES:

```
sage: from sage.rings.qqbar import AlgebraicGenerator, ANRoot, qq_generator
sage: _.<y> = QQ['y']
sage: x = polygen(QQbar)
sage: nf2 = NumberField(y^2 - 2, name='a', check=False)
sage: root2 = ANRoot(x^2 - 2, RIF(1, 2))
sage: gen2 = AlgebraicGenerator(nf2, root2)
sage: gen2
Number Field in a with defining polynomial y^2 - 2 with a in 1.414213562373095?
sage: nf3 = NumberField(y^2 - 3, name='a', check=False)
sage: root3 = ANRoot(x^2 - 3, RIF(1, 2))
sage: gen3 = AlgebraicGenerator(nf3, root3)
sage: gen3
Number Field in a with defining polynomial y^2 - 3 with a in 1.732050807568878?
sage: gen2_3 = gen2.union(gen3)
sage: gen2_3
Number Field in a with defining polynomial y^4 - 4*y^2 + 1 with a in 0.5176380902050415?
sage: qq_generator.super_poly(gen2) is None
True
sage: gen2.super_poly(gen2_3)
-a^3 + 3*a
sage: gen3.super_poly(gen2_3)
-a^2 + 2
```

union(*other*)

Given generators `alpha` and `beta`, `alpha.union(beta)` gives a generator for the number field $\mathbf{Q}[\alpha][\beta]$.

EXAMPLES:

```
sage: from sage.rings.qqbar import ANRoot, AlgebraicGenerator, qq_generator
sage: _.<y> = QQ['y']
sage: x = polygen(QQbar)
sage: nf2 = NumberField(y^2 - 2, name='a', check=False)
sage: root2 = ANRoot(x^2 - 2, RIF(1, 2))
sage: gen2 = AlgebraicGenerator(nf2, root2)
sage: gen2
Number Field in a with defining polynomial y^2 - 2 with a in 1.4142135623730950?
sage: nf3 = NumberField(y^2 - 3, name='a', check=False)
sage: root3 = ANRoot(x^2 - 3, RIF(1, 2))
sage: gen3 = AlgebraicGenerator(nf3, root3)
sage: gen3
Number Field in a with defining polynomial y^2 - 3 with a in 1.7320508075688780?
sage: gen2.union(qq_generator) is gen2
True
sage: qq_generator.union(gen3) is gen3
True
sage: gen2.union(gen3)
Number Field in a with defining polynomial y^4 - 4*y^2 + 1 with a in 0.5176380902050415?
```

class AlgebraicGeneratorRelation(*child1, child1_poly, child2, child2_poly, parent*)

A simple class for maintaining relations in the lattice of algebraic extensions.

class AlgebraicNumber(*x*)

The class for algebraic numbers (complex numbers which are the roots of a polynomial with integer coefficients). Much of its functionality is inherited from `AlgebraicNumber_base`.

complex_exact(*field*)

Given a `ComplexField`, return the best possible approximation of this number in that field. Note that if either component is sufficiently close to the halfway point between two floating-point numbers in the corresponding `RealField`, then this will trigger exact computation, which may be very slow.

EXAMPLES:

```
sage: a = QQbar.zeta(9) + I + QQbar.zeta(9).conjugate(); a
1.532088886237957? + 1.000000000000000?*I
sage: a.complex_exact(CIF)
1.532088886237957? + 1*I
```

complex_number(*field*)

Given a `ComplexField`, compute a good approximation to self in that field. The approximation will be off by at most two ulp's in each component, except for components which are very close to zero, which will have an abolute error at most `2**(-(field.prec()-1))`.

EXAMPLES:

```
sage: a = QQbar.zeta(5)
sage: a.complex_number(CIF)
0.309016994374947 + 0.951056516295154*I
sage: (a + a.conjugate()).complex_number(CIF)
0.618033988749895 - 5.42101086242752e-20*I
```

conjugate()

Returns the complex conjugate of self.

EXAMPLES:

```
sage: QQbar(3 + 4*I).conjugate()
3 - 4*I
sage: QQbar.zeta(7).conjugate()
0.6234898018587335? - 0.7818314824680299?*I
sage: QQbar.zeta(7) + QQbar.zeta(7).conjugate()
1.246979603717467? + 0.?e-18*I
```

imag()

interval_exact(*field*)

Given a ComplexIntervalField, compute the best possible approximation of this number in that field. Note that if either the real or imaginary parts of this number are sufficiently close to some floating-point number (and, in particular, if either is exactly representable in floating-point), then this will trigger exact computation, which may be very slow.

EXAMPLES:

```
sage: a = QQbar(I).sqrt(); a
0.7071067811865475? + 0.7071067811865475?*I
sage: a.interval_exact(CIF)
0.7071067811865475? + 0.7071067811865475?*I
sage: b = QQbar((1+I)*sqrt(2)/2)
sage: (a - b).interval(CIF)
0.?e-19 + 0.?e-18*I
sage: (a - b).interval_exact(CIF)
0
```

multiplicative_order()

Compute the multiplicative order of this algebraic real number. That is, find the smallest positive integer n such that $x^n = 1$. If there is no such n, returns +Infinity.

We first check that abs(x) is very close to 1. If so, we compute x exactly and examine its argument.

EXAMPLES:

```
sage: QQbar(-sqrt(3)/2 - I/2).multiplicative_order()
12
sage: QQbar(1).multiplicative_order()
1
sage: QQbar(-I).multiplicative_order()
4
sage: QQbar(707/1000 + 707/1000*I).multiplicative_order()
+Infinity
sage: QQbar(3/5 + 4/5*I).multiplicative_order()
+Infinity
```

norm()

Returns self * self.conjugate(). This is the algebraic definition of norm, if we view QQbar as AA[I].

EXAMPLES:

```
sage: QQbar(3 + 4*I).norm()
25
sage: type(QQbar(I).norm())
<class 'sage.rings.qqbar.AlgebraicReal'>
sage: QQbar.zeta(1007).norm()
1
```

rational_argument()

Returns the argument of self, divided by 2π, as long as this result is rational. Otherwise returns None. Always triggers exact computation.

EXAMPLES:

```
sage: QQbar((1+I)*(sqrt(2)+sqrt(5))).rational_argument()
1/8
sage: QQbar(-1 + I*sqrt(3)).rational_argument()
1/3
sage: QQbar(-1 - I*sqrt(3)).rational_argument()
-1/3
sage: QQbar(3+4*I).rational_argument() is None
True
sage: (QQbar.zeta(7654321)^65536).rational_argument()
65536/7654321
sage: (QQbar.zeta(3)^65536).rational_argument()
1/3
```

real()

class AlgebraicNumber_base(*parent, x*)

This is the common base class for algebraic numbers (complex numbers which are the zero of a polynomial in $\mathbf{Z}[x]$) and algebraic reals (algebraic numbers which happen to be real).

`AlgebraicNumber` objects can be created using `QQbar` (== `AlgebraicNumberField()`), and `AlgebraicReal` objects can be created using `AA` (== `AlgebraicRealField()`). They can be created either by coercing a rational or a symbolic expression, or by using the `QQbar.polynomial_root()` or `AA.polynomial_root()` method to construct a particular root of a polynomial with algebraic coefficients. Also, `AlgebraicNumber` and `AlgebraicReal` are closed under addition, subtraction, multiplication, division (except by 0), and rational powers (including roots), except that for a negative `AlgebraicReal`, taking a power with an even denominator returns an `AlgebraicNumber` instead of an `AlgebraicReal`.

`AlgebraicNumber` and `AlgebraicReal` objects can be approximated to any desired precision. They can be compared exactly; if the two numbers are very close, or are equal, this may require exact computation, which can be extremely slow.

As long as exact computation is not triggered, computation with algebraic numbers should not be too much slower than computation with intervals. As mentioned above, exact computation is triggered when comparing two algebraic numbers which are very close together. This can be an explicit comparison in user code, but the following list of actions (not necessarily complete) can also trigger exact computation:

- Dividing by an algebraic number which is very close to 0.
- Using an algebraic number which is very close to 0 as the leading coefficient in a polynomial.
- Taking a root of an alebraic number which is very close to 0.

The exact definition of "very close" is subject to change; currently, we compute our best approximation of the two numbers using 128-bit arithmetic, and see if that's sufficient to decide the comparison. Note that comparing two algebraic numbers which are actually equal will always trigger exact computation, unless they are actually the same object.

EXAMPLES:

```
sage: sqrt(QQbar(2))
1.414213562373095?
sage: sqrt(QQbar(2))^2 == 2
True
sage: x = polygen(QQbar)
sage: phi = QQbar.polynomial_root(x^2 - x - 1, RIF(1, 2))
sage: phi
1.618033988749895?
sage: phi^2 == phi+1
True
sage: AA(sqrt(65537))
256.0019531175495?
```

as_number_field_element(*minimal=False*)

Returns a number field containing this value, a representation of this value as an element of that number field, and a homomorphism from the number field back to `AA` or `QQbar`.

This may not return the smallest such number field, unless `minimal=True` is specified.

To compute a single number field containing multiple algebraic numbers, use the function `number_field_elements_from_algebraics` instead.

EXAMPLES:

```
sage: QQbar(sqrt(8)).as_number_field_element()
(Number Field in a with defining polynomial y^2 - 2, 2*a, Ring morphism:
    From: Number Field in a with defining polynomial y^2 - 2
    To:   Algebraic Real Field
    Defn: a |--> 1.414213562373095?)
sage: x = polygen(ZZ)
sage: p = x^3 + x^2 + x + 17
sage: (rt,) = p.roots(ring=AA, multiplicities=False); rt
-2.804642726932742?
sage: (nf, elt, hom) = rt.as_number_field_element(); (nf, elt, hom)
(Number Field in a with defining polynomial y^3 - y^2 + y - 17, -a, Ring morphism:
    From: Number Field in a with defining polynomial y^3 - y^2 + y - 17
    To:   Algebraic Real Field
    Defn: a |--> 2.804642726932742?)
sage: hom(elt) == rt
True
```

We see an example where we do not get the minimal number field unless we specify `minimal=True`:

```
sage: rt2 = AA(sqrt(2))
sage: rt3 = AA(sqrt(3))
sage: rt3b = rt2 + rt3 - rt2
sage: rt3b.as_number_field_element()
(Number Field in a with defining polynomial y^4 - 4*y^2 + 1, -a^2 + 2, Ring morphism:
    From: Number Field in a with defining polynomial y^4 - 4*y^2 + 1
    To:   Algebraic Real Field
    Defn: a |--> 0.5176380902050415?)
sage: rt3b.as_number_field_element(minimal=True)
(Number Field in a with defining polynomial y^2 - 3, a, Ring morphism:
    From: Number Field in a with defining polynomial y^2 - 3
    To:   Algebraic Real Field
    Defn: a |--> 1.732050807568878?)
```

degree()

Return the degree of this algebraic number (the degree of its minimal polynomial, or equivalently, the degree of the smallest algebraic extension of the rationals containing this number).

EXAMPLES:

```
sage: QQbar(5/3).degree()
1
sage: sqrt(QQbar(2)).degree()
2
sage: QQbar(17).nth_root(5).degree()
5
sage: sqrt(3+sqrt(QQbar(8))).degree()
2
```

exactify()

Compute an exact representation for this number.

EXAMPLES:

```
sage: two = QQbar(4).nth_root(4)^2
sage: two
2.000000000000000?
sage: two.exactify()
sage: two
2
```

interval (*field*)

Given an interval field (real or complex, as appropriate) of precision p, compute an interval representation of self with `diameter()` at most 2^{-p}; then round that representation into the given field. Here `diameter()` is relative diameter for intervals not containing 0, and absolute diameter for intervals that do contain 0; thus, if the returned interval does not contain 0, it has at least $p - 1$ good bits.

EXAMPLES:

```
sage: RIF64 = RealIntervalField(64)
sage: x = AA(2).sqrt()
sage: y = x*x
sage: y = 1000 * y - 999 * y
sage: y.interval_fast(RIF64)
2.000000000000000?
sage: y.interval(RIF64)
2.0000000000000000?
sage: CIF64 = ComplexIntervalField(64)
sage: x = QQbar.zeta(11)
sage: x.interval_fast(CIF64)
0.84125353283118116897? + 0.54064081745559758?*I
sage: x.interval(CIF64)
0.84125353283118116897? + 0.54064081745559758?*I
```

interval_diameter (*diam*)

Compute an interval representation of self with `diameter()` at most `diam`. The precision of the returned value is unpredictable.

EXAMPLES:

```
sage: AA(2).sqrt().interval_diameter(1e-10)
1.4142135623730950488?
sage: AA(2).sqrt().interval_diameter(1e-30)
1.41421356237309504880168872420969807857?
sage: QQbar(2).sqrt().interval_diameter(1e-10)
1.4142135623730950488?
sage: QQbar(2).sqrt().interval_diameter(1e-30)
1.41421356237309504880168872420969807857?
```

interval_fast (*field*)

Given a `RealIntervalField`, compute the value of this number using interval arithmetic of at least the precision of the field, and return the value in that field. (More precision may be used in the computation.) The returned interval may be arbitrarily imprecise, if this number is the result of a sufficiently long computation chain.

EXAMPLES:

```
sage: x = AA(2).sqrt()
sage: x.interval_fast(RIF)
1.414213562373095?
sage: x.interval_fast(RealIntervalField(200))
1.41421356237309504880168872420969807856967187537694807317667?
sage: x = QQbar(I).sqrt()
sage: x.interval_fast(CIF)
0.7071067811865475? + 0.7071067811865475?*I
```

```
sage: x.interval_fast(RIF)
...
TypeError: Unable to convert number to real interval.
```

minpoly()

Compute the minimal polynomial of this algebraic number. The minimal polynomial is the monic polynomial of least degree having this number as a root; it is unique.

EXAMPLES:

```
sage: QQbar(4).sqrt().minpoly()
x - 2
sage: ((QQbar(2).nth_root(4))^2).minpoly()
x^2 - 2
sage: v = sqrt(QQbar(2)) + sqrt(QQbar(3)); v
3.146264369941973?
sage: p = v.minpoly(); p
x^4 - 10*x^2 + 1
sage: p(RR(v.real()))
1.31006316905768e-14
```

nth_root(*n*)

Return the n-th root of this number.

Note that for odd n and negative real numbers, `AlgebraicReal` and `AlgebraicNumber` values give different answers: `AlgebraicReal` values prefer real results, and `AlgebraicNumber` values return the principal root.

EXAMPLES:

```
sage: AA(-8).nth_root(3)
-2
sage: QQbar(-8).nth_root(3)
1.000000000000000? + 1.732050807568878?*I
sage: QQbar.zeta(12).nth_root(15)
0.9993908270190957? + 0.03489949670250097?*I
```

simplify()

Compute an exact representation for this number, in the smallest possible number field.

EXAMPLES:

```
sage: rt2 = AA(sqrt(2))
sage: rt3 = AA(sqrt(3))
sage: rt2b = rt3 + rt2 - rt3
sage: rt2b.exactify()
sage: rt2b._exact_value()
a^3 - 3*a where a^4 - 4*a^2 + 1 = 0 and a in 1.931851652578137?
sage: rt2b.simplify()
sage: rt2b._exact_value()
a where a^2 - 2 = 0 and a in 1.414213562373095?
```

sqrt()

Return the square root of this number.

EXAMPLES:

```
sage: AA(2).sqrt()
1.414213562373095?
sage: QQbar(I).sqrt()
0.7071067811865475? + 0.7071067811865475?*I
```

class **AlgebraicPolynomialTracker**(*poly*)

Keeps track of a polynomial used for algebraic numbers.

If multiple algebraic numbers are created as roots of a single polynomial, this allows the polynomial and information about the polynomial to be shared. This reduces work if the polynomial must be recomputed at higher precision, or if it must be factored.

This class is private, and should only be constructed by `AA.common_polynomial()` or `QQbar.common_polynomial()`, and should only be used as an argument to `AA.polynomial_root()` or `QQbar.polynomial_root()`. (It doesn't matter whether you create the common polynomial with `AA.common_polynomial()` or `QQbar.common_polynomial()`.)

EXAMPLES:

```
sage: x = polygen(QQbar)
sage: P = QQbar.common_polynomial(x^2 - x - 1)
sage: P
x^2 - x - 1
sage: QQbar.polynomial_root(P, RIF(1, 2))
1.618033988749895?
```

complex_roots (*prec, multiplicity*)

EXAMPLES:

```
sage: x = polygen(ZZ)
sage: cp = AA.common_polynomial(x^4 - 2)
```

Note that the precision is not guaranteed to find the tightest possible interval since complex_roots() depends on the underlying BLAS implementation.

```
sage: cp.complex_roots(30, 1)
[-1.18920711500272...?,
 1.189207115002721?,
 -1.189207115002721?*I,
 1.189207115002721?*I]
```

exactify()

Compute a common field that holds all of the algebraic coefficients of this polynomial, then factor the polynomial over that field. Store the factors for later use (ignoring multiplicity).

factors()

generator()

is_complex()

poly()

class **AlgebraicReal** (*x*)

imag()

Returns the imaginary part of this algebraic real (so it always returns 0).

EXAMPLES:

```
sage: a = AA(sqrt(2) + sqrt(3))
sage: a.imag()
0
sage: parent(a.imag())
Algebraic Real Field
```

interval_exact (*field*)

Given a `RealIntervalField`, compute the best possible approximation of this number in that field. Note that if this number is sufficiently close to some floating-point number (and, in particular, if this number is exactly representable in floating-point), then this will trigger exact computation, which may be very slow.

EXAMPLES:

```
sage: x = AA(2).sqrt()
sage: y = x*x
sage: x.interval(RIF)
1.414213562373095?
sage: x.interval_exact(RIF)
1.414213562373095?
sage: y.interval(RIF)
2.000000000000000?
sage: y.interval_exact(RIF)
2
sage: z = 1 + AA(2).sqrt() / 2^200
sage: z.interval(RIF)
1.000000000000001?
sage: z.interval_exact(RIF)
1.000000000000001?
```

real()

Returns the real part of this algebraic real (so it always returns self).

EXAMPLES:

```
sage: a = AA(sqrt(2) + sqrt(3))
sage: a.real()
3.146264369941973?
sage: a.real() is a
True
```

real_exact(*field*)

Given a `RealField`, compute the best possible approximation of this number in that field. Note that if this number is sufficiently close to the halfway point between two floating-point numbers in the field (for the default round-to-nearest mode) or if the number is sufficiently close to a floating-point number in the field (for directed rounding modes), then this will trigger exact computation, which may be very slow.

The rounding mode of the field is respected.

EXAMPLES:

```
sage: x = AA(2).sqrt()^2
sage: x.real_exact(RR)
2.00000000000000
sage: x.real_exact(RealField(53, rnd='RNDD'))
2.00000000000000
sage: x.real_exact(RealField(53, rnd='RNDU'))
2.00000000000000
sage: x.real_exact(RealField(53, rnd='RNDZ'))
2.00000000000000
sage: (-x).real_exact(RR)
-2.00000000000000
sage: (-x).real_exact(RealField(53, rnd='RNDD'))
-2.00000000000000
sage: (-x).real_exact(RealField(53, rnd='RNDU'))
-2.00000000000000
sage: (-x).real_exact(RealField(53, rnd='RNDZ'))
-2.00000000000000
sage: (x-2).real_exact(RR)
0.000000000000000
sage: (x-2).real_exact(RealField(53, rnd='RNDD'))
0.000000000000000
sage: (x-2).real_exact(RealField(53, rnd='RNDU'))
0.000000000000000
sage: (x-2).real_exact(RealField(53, rnd='RNDZ'))
```

```
0.000000000000000
sage: y = AA(2).sqrt()
sage: y.real_exact(RR)
1.41421356237310
sage: y.real_exact(RealField(53, rnd='RNDD'))
1.41421356237309
sage: y.real_exact(RealField(53, rnd='RNDU'))
1.41421356237310
sage: y.real_exact(RealField(53, rnd='RNDZ'))
1.41421356237309
```

real_number(*field*)

Given a RealField, compute a good approximation to self in that field. The approximation will be off by at most two ulp's, except for numbers which are very close to 0, which will have an absolute error at most 2**(-(field.prec()-1)). Also, the rounding mode of the field is respected.

EXAMPLES:

```
sage: x = AA(2).sqrt()^2
sage: x.real_number(RR)
2.00000000000000
sage: x.real_number(RealField(53, rnd='RNDD'))
1.99999999999999
sage: x.real_number(RealField(53, rnd='RNDU'))
2.00000000000001
sage: x.real_number(RealField(53, rnd='RNDZ'))
1.99999999999999
sage: (-x).real_number(RR)
-2.00000000000000
sage: (-x).real_number(RealField(53, rnd='RNDD'))
-2.00000000000001
sage: (-x).real_number(RealField(53, rnd='RNDU'))
-1.99999999999999
sage: (-x).real_number(RealField(53, rnd='RNDZ'))
-1.99999999999999
sage: (x-2).real_number(RR)
5.42101086242752e-20
sage: (x-2).real_number(RealField(53, rnd='RNDD'))
-1.08420217248551e-19
sage: (x-2).real_number(RealField(53, rnd='RNDU'))
2.16840434497101e-19
sage: (x-2).real_number(RealField(53, rnd='RNDZ'))
0.000000000000000
sage: y = AA(2).sqrt()
sage: y.real_number(RR)
1.41421356237309
sage: y.real_number(RealField(53, rnd='RNDD'))
1.41421356237309
sage: y.real_number(RealField(53, rnd='RNDU'))
1.41421356237310
sage: y.real_number(RealField(53, rnd='RNDZ'))
1.41421356237309
```

sign()

Compute the sign of this algebraic number (return -1 if negative, 0 if zero, or 1 if positive).

Computes an interval enclosing this number using 128-bit interval arithmetic; if this interval includes 0, then fall back to exact computation (which can be very slow).

EXAMPLES:

```
sage: AA(-5).nth_root(7).sign()
-1
sage: (AA(2).sqrt() - AA(2).sqrt()).sign()
0
```

class AlgebraicRealField()

The field of algebraic reals.

algebraic_closure()

EXAMPLES:

```
sage: AA.algebraic_closure()
Algebraic Field
```

completion(*p, prec, extras={}*)

EXAMPLES:

```
sage: AA.completion(infinity, 500)
Real Field with 500 bits of precision
sage: AA.completion(infinity, prec=53, extras={'type':'RDF'})
Real Double Field
sage: AA.completion(infinity, 53) is RR
True
sage: AA.completion(7, 10)
...
NotImplementedError
```

gen(*n=0*)

gens()

has_coerce_map_from_impl(*from_par*)

is_atomic_repr()

ngens()

polynomial_root(*poly, interval, multiplicity=1*)

Given a polynomial with algebraic coefficients and an interval enclosing exactly one root of the polynomial, constructs an algebraic real representation of that root.

The polynomial need not be irreducible, or even squarefree; but if the given root is a multiple root, its multiplicity must be specified. (IMPORTANT NOTE: Currently, multiplicity-k roots are handled by taking the $(k-1)$-st derivative of the polynomial. This means that the interval must enclose exactly one root of this derivative.)

The conditions on the arguments (that the interval encloses exactly one root, and that multiple roots match the given multiplicity) are not checked; if they are not satisfied, an error may be thrown (possibly later, when the algebraic number is used), or wrong answers may result.

Note that if you are constructing multiple roots of a single polynomial, it is better to use `AA.common_polynomial` (or `QQbar.common_polynomial`; the two are equivalent) to get a shared polynomial.

EXAMPLES:

```
sage: x = polygen(AA)
sage: phi = AA.polynomial_root(x^2 - x - 1, RIF(1, 2)); phi
1.618033988749895?
sage: p = (x-1)^7 * (x-2)
sage: r = AA.polynomial_root(p, RIF(9/10, 11/10), multiplicity=7)
sage: r; r == 1
1.000000000000000?
True
sage: p = (x-phi)*(x-sqrt(AA(2)))
```

```
sage: r = AA.polynomial_root(p, RIF(1, 3/2))
sage: r; r == sqrt(AA(2))
1.414213562373095?
True
```

We allow complex polynomials, as long as the particular root in question is real.

```
sage: K.<im> = QQ[I]
sage: x = polygen(K)
sage: p = (im + 1) * (x^3 - 2); p
(I + 1)*x^3 - 2*I - 2
sage: r = AA.polynomial_root(p, RIF(1, 2)); r^3
2.000000000000000?
```

zeta(*n=2*)

an_addsub_element(*a, b, sub*)

an_addsub_expr(*a, b, sub*)

an_addsub_gaussian(*a, b, sub*)

an_addsub_rational(*a, b, sub*)

an_addsub_rootunity(*a, b, sub*)

an_addsub_zero(*a, b, sub*)

an_muldiv_element(*a, b, div*)

an_muldiv_expr(*a, b, div*)

an_muldiv_gaussian(*a, b, div*)

an_muldiv_rational(*a, b, div*)

an_muldiv_rootunity(*a, b, div*)

an_muldiv_zero(*a, b, div*)

clear_denominators(*poly*)

Takes a monic polynomial and rescales the variable to get a monic polynomial with "integral" coefficients. Works on any univariate polynomial whose base ring has a `denominator()` method that returns integers; for example, the base ring might be $\mathbf{Q}$ or a number field.

Returns the scale factor and the new polynomial.

(Inspired by Pari's `primitive_pol_to_monic()`.)

We assume that coefficient denominators are "small"; the algorithm factors the denominators, to give the smallest possible scale factor.

EXAMPLES:

```
sage: from sage.rings.qqbar import clear_denominators

sage: _.<x> = QQ['x']
sage: clear_denominators(x + 3/2)
(2, x + 3)
sage: clear_denominators(x^2 + x/2 + 1/4)
(2, x^2 + x + 1)
```

conjugate_expand(*v*)

If the interval `v` (which may be real or complex) includes some purely real numbers, return `v'` containing `v` such that `v' == v'.conjugate()`. Otherwise return `v` unchanged. (Note that if `v' == v'.conjugate()`,

and v' includes one non-real root of a real polynomial, then v' also includes the conjugate of that root. Also note that the diameter of the return value is at most twice the diameter of the input.)

EXAMPLES:

```
sage: from sage.rings.qqbar import conjugate_expand
sage: conjugate_expand(CIF(RIF(0, 1), RIF(1, 2))).str(style='brackets')
'[0.00000000000000000 .. 1.0000000000000000] + [1.0000000000000000 .. 2.0000000000000000]*I'
sage: conjugate_expand(CIF(RIF(0, 1), RIF(0, 1))).str(style='brackets')
'[0.00000000000000000 .. 1.0000000000000000] + [-1.0000000000000000 .. 1.0000000000000000]*I'
sage: conjugate_expand(CIF(RIF(0, 1), RIF(-2, 1))).str(style='brackets')
'[0.00000000000000000 .. 1.0000000000000000] + [-2.0000000000000000 .. 2.0000000000000000]*I'
sage: conjugate_expand(RIF(1, 2)).str(style='brackets')
'[1.0000000000000000 .. 2.0000000000000000]'
```

conjugate_shrink(*v*)

If the interval v includes some purely real numbers, return a real interval containing only those real numbers. Otherwise return v unchanged.

If v includes exactly one root of a real polynomial, and v was returned by conjugate_expand(), then conjugate_shrink(v) still includes that root, and is a RealIntervalFieldElement iff the root in question is real.

EXAMPLES:

```
sage: from sage.rings.qqbar import conjugate_shrink
sage: conjugate_shrink(RIF(3, 4)).str(style='brackets')
'[3.0000000000000000 .. 4.0000000000000000]'
sage: conjugate_shrink(CIF(RIF(1, 2), RIF(1, 2))).str(style='brackets')
'[1.0000000000000000 .. 2.0000000000000000] + [1.0000000000000000 .. 2.0000000000000000]*I'
sage: conjugate_shrink(CIF(RIF(1, 2), RIF(0, 1))).str(style='brackets')
'[1.0000000000000000 .. 2.0000000000000000]'
sage: conjugate_shrink(CIF(RIF(1, 2), RIF(-1, 2))).str(style='brackets')
'[1.0000000000000000 .. 2.0000000000000000]'
```

cyclotomic_generator(*n*)

do_polred(*poly*)

Find the polynomial of lowest discriminant that generates the same field as poly, out of those returned by the Pari polred routine. Returns a triple: (elt_fwd, elt_back, new_poly), where new_poly is the new polynomial, elt_fwd is a polynomial expression for a root of the new polynomial in terms of a root of the original polynomial, and elt_back is a polynomial expression for a root of the original polynomial in terms of a root of the new polynomial.

EXAMPLES:

```
sage: from sage.rings.qqbar import do_polred

sage: _.<x> = QQ['x']
sage: do_polred(x^2-5)
(-1/2*x + 1/2, -2*x + 1, x^2 - x - 1)
sage: do_polred(x^2-x-11)
(-1/3*x + 2/3, -3*x + 2, x^2 - x - 1)
sage: do_polred(x^3 + 123456)
(-1/4*x, -4*x, x^3 - 1929)
```

find_zero_result(*fn, l*)

l is a list of some sort. fn is a function which maps an element of l and a precision into an interval (either real or complex) of that precision, such that for sufficient precision, exactly one element of l results in an interval containing 0. Returns that one element of l.

EXAMPLES:

```
sage: from sage.rings.qqbar import find_zero_result
sage: _.<x> = QQ['x']
sage: delta = 10^(-70)
sage: p1 = x - 1
sage: p2 = x - 1 - delta
sage: p3 = x - 1 + delta
sage: p2 == find_zero_result(lambda p, prec: p(RealIntervalField(prec)(1 + delta)), [p1, p2, p3]
True
```

get_AA_golden_ratio()

is_AlgebraicField(*F*)

is_AlgebraicField_common(*F*)

is_AlgebraicNumber(*x*)

is_AlgebraicReal(*x*)

is_AlgebraicRealField(*F*)

isolating_interval(*intv_fn, pol*)

intv_fn is a function that takes a precision and returns an interval of that precision containing some particular root of pol. (It must return better approximations as the precision increases.) pol is an irreducible polynomial with rational coefficients.

Returns an interval containing at most one root of pol.

EXAMPLES:

```
sage: from sage.rings.qqbar import isolating_interval

sage: _.<x> = QQ['x']
sage: isolating_interval(lambda prec: sqrt(RealIntervalField(prec)(2)), x^2 - 2)
1.4142135623730950488?
```

And an example that requires more precision:

```
sage: delta = 10^(-70)
sage: p = (x - 1) * (x - 1 - delta) * (x - 1 + delta)
sage: isolating_interval(lambda prec: RealIntervalField(prec)(1 + delta), p)
1.000000000000000000000000000000000000000000000000000000000000000000000000000000010000000000000000000000000
```

The function also works with complex intervals and complex roots:

```
sage: p = x^2 - x + 13/36
sage: isolating_interval(lambda prec: ComplexIntervalField(prec)(1/2, 1/3), p)
0.50000000000000000000? + 0.33333333333333334?*I
```

late_import()

number_field_elements_from_algebraics(*numbers, minimal=False*)

Given a sequence of elements of either AA or QQbar (or a mixture), computes a number field containing all of these elements, these elements as members of that number field, and a homomorphism from the number field back to AA or QQbar.

This may not return the smallest such number field, unless minimal=True is specified.

Also, a single number can be passed, rather than a sequence; and any values which are not elements of AA or QQbar will automatically be coerced to QQbar

This function may be useful for efficiency reasons: doing exact computations in the corresponding number field will be faster than doing exact computations directly in `AA` or `QQbar`.

EXAMPLES:

We can use this to compute the splitting field of a polynomial. (Unfortunately this takes an unreasonably long time for non-toy examples.):

```
sage: x = polygen(QQ)
sage: p = x^3 + x^2 + x + 17
sage: rts = p.roots(ring=QQbar, multiplicities=False)
sage: splitting = number_field_elements_from_algebraics(rts)[0]; splitting
Number Field in a with defining polynomial y^6 + 169*y^4 + 7968*y^2 + 121088
sage: p.roots(ring=splitting)
[(-9/2176*a^4 - 1121/2176*a^2 - 1625/136, 1), (9/17408*a^5 + 9/4352*a^4 + 1121/17408*a^3 + 1121/

sage: rt2 = AA(sqrt(2)); rt2
1.414213562373095?
sage: rt3 = AA(sqrt(3)); rt3
1.732050807568878?
sage: qqI = QQbar.zeta(4); qqI
1*I
sage: z3 = QQbar.zeta(3); z3
-0.500000000000000? + 0.866025403784439?*I
sage: rt2b = rt3 + rt2 - rt3; rt2b
1.414213562373095?
sage: rt2c = z3 + rt2 - z3; rt2c
1.414213562373095? + 0.?e-18*I

sage: number_field_elements_from_algebraics(rt2)
(Number Field in a with defining polynomial y^2 - 2, a, Ring morphism:
    From: Number Field in a with defining polynomial y^2 - 2
    To:   Algebraic Real Field
    Defn: a |--> 1.414213562373095?)

sage: number_field_elements_from_algebraics((rt2,rt3))
(Number Field in a with defining polynomial y^4 - 4*y^2 + 1, [-a^3 + 3*a, -a^2 + 2], Ring morphi
    From: Number Field in a with defining polynomial y^4 - 4*y^2 + 1
    To:   Algebraic Real Field
    Defn: a |--> 0.5176380902050415?)
```

We've created `rt2b` in such a way that sage doesn't initially know that it's in a degree-2 extension of $\mathbf{Q}$:

```
sage: number_field_elements_from_algebraics(rt2b)
(Number Field in a with defining polynomial y^4 - 4*y^2 + 1, -a^3 + 3*a, Ring morphism:
    From: Number Field in a with defining polynomial y^4 - 4*y^2 + 1
    To:   Algebraic Real Field
    Defn: a |--> 0.5176380902050415?)
```

We can specify `minimal=True` if we want the smallest number field:

```
sage: number_field_elements_from_algebraics(rt2b, minimal=True)
(Number Field in a with defining polynomial y^2 - 2, a, Ring morphism:
    From: Number Field in a with defining polynomial y^2 - 2
    To:   Algebraic Real Field
    Defn: a |--> 1.414213562373095?)
```

Things work fine with rational numbers, too:

```
sage: number_field_elements_from_algebraics((QQbar(1/2), AA(17)))
(Rational Field, [1/2, 17], Ring morphism:
    From: Rational Field
    To:   Algebraic Real Field
    Defn: 1 |--> 1)
```

Or we can just pass in symbolic expressions, as long as they can be coerced into `QQbar`:

```
sage: number_field_elements_from_algebraics((sqrt(7), sqrt(9), sqrt(11)))
(Number Field in a with defining polynomial y^4 - 9*y^2 + 1, [-a^3 + 8*a, 3, -a^3 + 10*a], Ring
    From: Number Field in a with defining polynomial y^4 - 9*y^2 + 1
    To:   Algebraic Real Field
    Defn: a |--> 0.3354367396454047?)
```

Here we see an example of doing some computations with number field elements, and then mapping them back into `QQbar`:

```
sage: (fld,nums,hom) = number_field_elements_from_algebraics((rt2, rt3, qqI, z3))
sage: fld,nums,hom
(Number Field in a with defining polynomial y^8 - y^4 + 1, [-a^5 + a^3 + a, a^6 - 2*a^2, a^6, -a
    From: Number Field in a with defining polynomial y^8 - y^4 + 1
    To:   Algebraic Field
    Defn: a |--> -0.2588190451025208? - 0.9659258262890683?*I)
sage: (nfrt2, nfrt3, nfI, nfz3) = nums
sage: hom(nfrt2)
1.414213562373095? + 0.?e-18*I
sage: nfrt2^2
2
sage: nfrt3^2
3
sage: nfz3 + nfz3^2
-1
sage: nfI^2
-1
sage: sum = nfrt2 + nfrt3 + nfI + nfz3; sum
2*a^6 - a^5 - a^4 + a^3 - 2*a^2 + a
sage: hom(sum)
2.646264369941973? + 1.866025403784439?*I
sage: hom(sum) == rt2 + rt3 + qqI + z3
True
sage: [hom(n) for n in nums] == [rt2, rt3, qqI, z3]
True
```

TESTS:

```
sage: number_field_elements_from_algebraics(rt3)
(Number Field in a with defining polynomial y^2 - 3, a, Ring morphism:
    From: Number Field in a with defining polynomial y^2 - 3
    To:   Algebraic Real Field
    Defn: a |--> 1.732050807568878?)
sage: number_field_elements_from_algebraics((rt2,qqI))
(Number Field in a with defining polynomial y^4 + 1, [a^3 - a, -a^2], Ring morphism:
    From: Number Field in a with defining polynomial y^4 + 1
    To:   Algebraic Field
    Defn: a |--> -0.7071067811865475? + 0.7071067811865475?*I)
```

Note that for the first example, where sage doesn't realize that the number is real, we get a homomorphism to QQbar; but with minimal=True, we get a homomorphism to AA. Also note that the exact answer depends on a Pari function that gives different answers for 32-bit and 64-bit machines:

```
sage: number_field_elements_from_algebraics(rt2c)
(Number Field in a with defining polynomial y^4 + 2*y^2 + 4, 1/2*a^3, Ring morphism:
    From: Number Field in a with defining polynomial y^4 + 2*y^2 + 4
    To:   Algebraic Field
    Defn: a |--> -0.7071067811865475? + 1.224744871391589?*I) # 32-bit
    Defn: a |--> -0.7071067811865475? - 1.224744871391589?*I) # 64-bit
sage: number_field_elements_from_algebraics(rt2c, minimal=True)
(Number Field in a with defining polynomial y^2 - 2, a, Ring morphism:
    From: Number Field in a with defining polynomial y^2 - 2
    To:   Algebraic Real Field
    Defn: a |--> 1.414213562373095?)
```

prec_seq()

rational_exact_root(*r*, *d*)

Checks whether the rational r is an exact d'th power. If so, returns the d'th root of r; otherwise, returns None.

EXAMPLES:

```
sage: from sage.rings.qqbar import rational_exact_root
sage: rational_exact_root(16/81, 4)
2/3
sage: rational_exact_root(8/81, 3) is None
True
```

short_prec_seq()

tail_prec_seq()

CHAPTER

TWENTYSEVEN

P-ADICS

27.1 Introduction to the p-adics

This tutorial outlines what you need to know in order to use p-adics in Sage effectively.

Our goal is to create a rich structure of different options that will reflect the mathematical structures of the p-adics. This is very much a work in progress: some of the classes that we eventually intend to include have not yet been written, and some of the functionality for classes in existence has not yet been implemented. In addition, while we strive for perfect code, bugs (both subtle and not-so-subtle) continue to evade our clutches. As a user, you serve an important role. By writing non-trivial code that uses the p-adics, you both give us insight into what features are actually used and also expose problems in the code for us to fix.

Our design philosophy has been to create a robust, usable interface working first, with simple-minded implementations underneath. We want this interface to stabilize rapidly, so that users' code does not have to change. Once we get the framework in place, we can go back and work on the algorithms and implementations underneath. All of the current p-adic code is currently written in pure Python, which means that it does not have the speed advantage of compiled code. Thus our p-adics can be painfully slow at times when you're doing real computations. However, finding and fixing bugs in Python code is *far* easier than finding and fixing errors in the compiled alternative within Sage (Cython), and Python code is also faster and easier to write. We thus have significantly more functionality implemented and working than we would have if we had chosen to focus initially on speed. And at some point in the future, we will go back and improve the speed. Any code you have written on top of our p-adics will then get an immediate performance enhancement.

If you do find bugs, have feature requests or general comments, please email sage-support@groups.google.com or roed@math.harvard.edu.

27.1.1 Terminology and types of p-adics

To write down a general p-adic element completely would require an infinite amount of data. Since computers do not have infinite storage space, we must instead store finite approximations to elements. Thus, just as in the case of floating point numbers for representing reals, we have to store an element to a finite precision level. The different ways of doing this account for the different types of p-adics.

We can think of p-adics in two ways. First, as a projective limit of finite groups:

$$\mathbb{Z}_p = \lim_{\leftarrow n} \mathbb{Z}/p^n\mathbb{Z}.$$

Secondly, as Cauchy sequences of rationals (or integers, in the case of $\mathbb{Z}_p$) under the p-adic metric. Since we only need to consider these sequences up to equivalence, this second way of thinking of the p-adics is the same as considering power series in p with integral coefficients in the range 0 to $p-1$. If we only allow nonnegative powers of p then these power series converge to elements of $\mathbb{Z}_p$, and if we allow bounded negative powers of p then we get $\mathbb{Q}_p$.

Both of these representations give a natural way of thinking about finite approximations to a p-adic element. In

the first representation, we can just stop at some point in the projective limit, giving an element of $\mathbb{Z}/p^n\mathbb{Z}$. As $\mathbb{Z}_p/p^n\mathbb{Z}_p \cong \mathbb{Z}/p^n\mathbb{Z}$, this is equivalent to specifying our element modulo $p^n\mathbb{Z}_p$.

The *absolute precision* of a finite approximation $\bar{x} \in \mathbb{Z}/p^n\mathbb{Z}$ to $x \in \mathbb{Z}_p$ is the non-negative integer n.

In the second representation, we can achieve the same thing by truncating a series

$$a_0 + a_1 p + a_2 p^2 + \cdots$$

at p^n, yielding

$$a_0 + a_1 p + \cdots + a_{n-1} p^{n-1} + O(p^n).$$

As above, we call this n the absolute precision of our element.

Given any $x \in \mathbb{Q}_p$ with $x \neq 0$, we can write $x = p^v u$ where $v \in \mathbf{Z}$ and $u \in \mathbb{Z}_p^\times$. We could thus also store an element of $\mathbb{Q}_p$ (or $\mathbb{Z}_p$) by storing v and a finite approximation of u. This motivates the following definition: the *relative precision* of an approximation to x is defined as the absolute precision of the approximation minus the valuation of x. For example, if $x = a_k p^k + a_{k+1} p^{k+1} + \cdots + a_{n-1} p^{n-1} + O(p^n)$ then the absolute precision of x is n, the valuation of x is k and the relative precision of x is $n - k$.

There are three different representations of $\mathbb{Z}_p$ in Sage and one representation of $\mathbb{Q}_p$:

- the fixed modulus ring
- the capped absolute precision ring
- the capped relative precision ring, and
- the capped relative precision field.

Fixed Modulus Rings

The first, and simplest, type of $\mathbb{Z}_p$ is basically a wrapper around $\mathbb{Z}/p^n\mathbb{Z}$, providing a unified interface with the rest of the p-adics. You specify a precision, and all elements are stored to that absolute precision. If you perform an operation that would normally lose precision, the element does not track that it no longer has full precision.

The fixed modulus ring provides the lowest level of convenience, but it is also the one that has the lowest computational overhead. Once we have ironed out some bugs, the fixed modulus elements will be those most optimized for speed.

As with all of the implementations of $\mathbb{Z}_p$, one creates a new ring using the constructor `Zp`, and passing in `'fixed-mod'` for the `type` parameter. For example,

```
sage: R = Zp(5, prec = 10, type = 'fixed-mod', print_mode = 'series')
sage: R
5-adic Ring of fixed modulus 5^10
```

One can create elements as follows:

```
sage: a = R(375)
sage: a
3*5^3 + O(5^10)
sage: b = R(105)
sage: b
5 + 4*5^2 + O(5^10)
```

Now that we have some elements, we can do arithmetic in the ring.

```
sage: a + b
5 + 4*5^2 + 3*5^3 + O(5^10)
sage: a * b
3*5^4 + 2*5^5 + 2*5^6 + O(5^10)
```

Floor division (//) divides even though the result isn't really known to the claimed precision; note that division isn't defined:

```
sage: a // 5
3*5^2 + O(5^10)
```

```
sage: a / 5
...
ValueError: cannot invert non-unit
```

Since elements don't actually store their actual precision, one can only divide by units:

```
sage: a / 2
4*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 2*5^7 + 2*5^8 + 2*5^9 + O(5^10)
sage: a / b
...
ValueError: cannot invert non-unit
```

If you want to divide by a non-unit, do it using the `//` operator:

```
sage: a // b
3*5^2 + 3*5^3 + 2*5^5 + 5^6 + 4*5^7 + 2*5^8 + O(5^10)
```

Capped Absolute Rings

The second type of implementation of $\mathbb{Z}_p$ is similar to the fixed modulus implementation, except that individual elements track their known precision. The absolute precision of each element is limited to be less than the precision cap of the ring, even if mathematically the precision of the element would be known to greater precision (see Appendix A for the reasons for the existence of a precision cap).

Once again, use `Zp` to create a capped absolute p-adic ring.

```
sage: R = Zp(5, prec = 10, type = 'capped-abs', print_mode = 'series')
sage: R
5-adic Ring with capped absolute precision 10
```

We can do similar things as in the fixed modulus case:

```
sage: a = R(375)
sage: a
3*5^3 + O(5^10)
sage: b = R(105)
sage: b
5 + 4*5^2 + O(5^10)
sage: a + b
5 + 4*5^2 + 3*5^3 + O(5^10)
sage: a * b
3*5^4 + 2*5^5 + 2*5^6 + O(5^10)
sage: c = a // 5
```

```
sage: c
3*5^2 + O(5^9)
```

Note that when we divided by 5, the precision of c dropped. This lower precision is now reflected in arithmetic.

```
sage: c + b
5 + 2*5^2 + 5^3 + O(5^9)
```

Division is allowed: the element that results is a capped relative field element, which is discussed in the next section:

```
sage: 1 / (c + b)
5^-1 + 3 + 2*5 + 5^2 + 4*5^3 + 4*5^4 + 3*5^6 + O(5^7)
```

Capped Relative Rings and Fields

Instead of restricting the absolute precision of elements (which doesn't make much sense when elements have negative valuations), one can cap the relative precision of elements. This is analogous to floating point representations of real numbers. As in the reals, multiplication works very well: the valuations add and the relative precision of the product is the minimum of the relative precisions of the inputs. Addition, however, faces similar issues as floating point addition: relative precision is lost when lower order terms cancel.

To create a capped relative precision ring, use Zp as before. To create capped relative precision fields, use Qp.

```
sage: R = Zp(5, prec = 10, type = 'capped-rel', print_mode = 'series')
sage: R
5-adic Ring with capped relative precision 10
sage: K = Qp(5, prec = 10, type = 'capped-rel', print_mode = 'series')
sage: K
5-adic Field with capped relative precision 10
```

We can do all of the same operations as in the other two cases, but precision works a bit differently: the maximum precision of an element is limited by the precision cap of the ring.

```
sage: a = R(375)
sage: a
3*5^3 + O(5^13)
sage: b = K(105)
sage: b
5 + 4*5^2 + O(5^11)
sage: a + b
5 + 4*5^2 + 3*5^3 + O(5^11)
sage: a * b
3*5^4 + 2*5^5 + 2*5^6 + O(5^14)
sage: c = a // 5
sage: c
3*5^2 + O(5^12)
sage: c + 1
1 + 3*5^2 + O(5^10)
```

As with the capped absolute precision rings, we can divide, yielding a capped relative precision field element.

```
sage: 1 / (c + b)
5^-1 + 3 + 2*5 + 5^2 + 4*5^3 + 4*5^4 + 3*5^6 + 2*5^7 + 5^8 + O(5^9)
```

Unramified Extensions

One can create unramified extensions of $\mathbb{Z}_p$ and $\mathbb{Q}_p$ using the functions `Zq` and `Qq`.

In addition to requiring a prime power as the first argument, `Zq` also requires a name for the generator of the residue field. One can specify this name as follows:

```
sage: R.<c> = Zq(125, prec = 20); R
Unramified Extension of 5-adic Ring with capped absolute precision 20
in c defined by (1 + O(5^20))*x^3 + (3 + O(5^20))*x + (3 + O(5^20))
```

Eisenstein Extensions

It is also possible to create Eisenstein extensions of $\mathbb{Z}_p$ and $\mathbb{Q}_p$. In order to do so, create the ground field first:

```
sage: R = Zp(5, 2)
```

Then define the polynomial yielding the desired extension.:

```
sage: S.<x> = ZZ[]
sage: f = x^5 - 25*x^3 + 15*x - 5
```

Finally, use the `ext` function on the ground field to create the desired extension.:

```
sage: W.<w> = R.ext(f)
```

You can do arithmetic in this Eisenstein extension:

```
sage: (1 + w)^7
1 + 2*w + w^2 + w^5 + 3*w^6 + 3*w^7 + 3*w^8 + w^9 + O(w^10)
```

Note that the precision cap increased by a factor of 5, since the ramification index of this extension over $\mathbb{Z}_p$ is 5.

27.2 Factory.

This file contains the contructor classes and functions for p-adic rings and fields.

AUTHORS:

- David Roe

QpCR(*p, prec=20, print_mode=None, halt=40, names=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_terms=None, check=True*)
A shorcut function to create capped relative p-adic fields.

Same functionality as `Qp`. See documentation for `Qp` for a description of the input paramters.

EXAMPLES:

```
sage: QpCR(5, 40)
5-adic Field with capped relative precision 40
```

class `Qp_class`()

A creation function for p-adic fields.

INPUT:

- `p` – integer: the p in $\mathbb{Q}_p$
- `prec` – integer (default: `20`) the precision cap of the field. Individual elements keep track of their own precision. See TYPES and PRECISION below.
- `type` – string (default: `'capped-rel'`) Valid types are `'capped-rel'` and `'lazy'` (though `'lazy'` currently doesn't work). See TYPES and PRECISION below
- `print_mode` – string (default: `None`). Valid modes are 'series', 'val-unit', 'terse', 'digits', and 'bars'. See PRINTING below
- `halt` – currently irrelevant (to be used for lazy fields)
- `names` – string or tuple (defaults to a string representation of p). What to use whenever p is printed.
- `ram_name` – string. Another way to specify the name; for consistency with the `Qq` and `Zq` and extension functions.
- `print_pos` – bool (default `None`) Whether to only use positive integers in the representations of elements. See PRINTING below.
- `print_sep` – string (default `None`) The separator character used in the `'bars'` mode. See PRINTING below.
- `print_alphabet` – tuple (default `None`) The encoding into digits for use in the 'digits' mode. See PRINTING below.
- `print_max_terms` – integer (default `None`) The maximum number of terms shown. See PRINTING below.
- `check` – bool (default `True`) whether to check if p is prime. Non-prime input may cause seg-faults (but can also be useful for base n expansions for example)

OUTPUT:

- The corresponding p-adic field.

TYPES AND PRECISION:

There are two types of precision for a p-adic element. The first is relative precision, which gives the number of known p-adic digits:

```
sage: R = Qp(5, 20, 'capped-rel', 'series'); a = R(675); a
2*5^2 + 5^4 + O(5^22)
sage: a.precision_relative()
20
```

The second type of precision is absolute precision, which gives the power of p that this element is defined modulo:

```
sage: a.precision_absolute()
22
```

There are two types of p-adic fields: capped relative fields and lazy fields.

In the capped relative case, the relative precision of an element is restricted to be at most a certain value, specified at the creation of the field. Individual elements also store their own precision, so the effect of various arithmetic operations on precision is tracked. When you cast an exact element into a capped relative field, it truncates it to the precision cap of the field.:

```
sage: R = Qp(5, 5, 'capped-rel', 'series'); a = R(4006); a
1 + 5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(4025); b
5^2 + 2*5^3 + 5^4 + 5^5 + O(5^7)
sage: a + b
1 + 5 + 5^2 + 4*5^3 + 2*5^4 + O(5^5)
```

The lazy case will eventually support elements that can increase their precision upon request. It is not currently implemented.

PRINTING:

There are many different ways to print p-adic elements. The way elements of a given field print is controlled by options passed in at the creation of the field. There are five basic printing modes (series, val-unit, terse, digits and bars), as well as various options that either hide some information in the print representation or sometimes make print representations more compact. Note that the printing options affect whether different p-adic fields are considered equal.

1. **series**: elements are displayed as series in p.:

```
sage: R = Qp(5, print_mode='series'); a = R(70700); a
3*5^2 + 3*5^4 + 2*5^5 + 4*5^6 + O(5^22)
sage: b = R(-70700); b
2*5^2 + 4*5^3 + 5^4 + 2*5^5 + 4*5^7 + 4*5^8 + 4*5^9 + 4*5^10 + 4*5^11 + 4*5^12 + 4*5^13 + 4*
```

print_pos controls whether negatives can be used in the coefficients of powers of p.:

```
sage: S = Qp(5, print_mode='series', print_pos=False); a = S(70700); a
-2*5^2 + 5^3 - 2*5^4 - 2*5^5 + 5^7 + O(5^22)
sage: b = S(-70700); b
2*5^2 - 5^3 + 2*5^4 + 2*5^5 - 5^7 + O(5^22)
```

print_max_terms limits the number of terms that appear.:

```
sage: T = Qp(5, print_mode='series', print_max_terms=4); b = R(-70700); repr(b)
'2*5^2 + 4*5^3 + 5^4 + 2*5^5 + ... + O(5^22)'
```

names affects how the prime is printed.:

```
sage: U.<p> = Qp(5); p
p + O(p^21)
```

print_sep and *print_alphabet* have no effect in series mode.

Note that print options affect equality:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

1. **val-unit**: elements are displayed as p^k*u:

```
sage: R = Qp(5, print_mode='val-unit'); a = R(70700); a
5^2 * 2828 + O(5^22)
sage: b = R(-707/5); b
5^-1 * 95367431639918 + O(5^19)
```

print_pos controls whether to use a balanced representation or not.:

```
sage: S = Qp(5, print_mode='val-unit', print_pos=False); b = S(-70700); b
5^2 * (-2828) + O(5^22)
```

names affects how the prime is printed.:

```
sage: T = Qp(5, print_mode='val-unit', names='pi'); a = T(70700); a
pi^2 * 2828 + O(pi^22)
```

print_max_terms, *print_sep* and *print_alphabet* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

1. **terse**: elements are displayed as an integer in base 10 or the quotient of an integer by a power of p (still in base 10):

   ```
   sage: R = Qp(5, print_mode='terse'); a = R(70700); a
   70700 + O(5^22)
   sage: b = R(-70700); b
   2384185790944925 + O(5^22)
   sage: c = R(-707/5); c
   95367431639918/5 + O(5^19)
   ```

The denominator, as of version 3.3, is always printed explicitly as a power of p, for predictability.:

```
sage: d = R(707/5^2); d
707/5^2 + O(5^18)
```

print_pos controls whether to use a balanced representation or not.:

```
sage: S = Qp(5, print_mode='terse', print_pos=False); b = S(-70700); b
-70700 + O(5^22)
sage: c = S(-707/5); c
-707/5 + O(5^19)
```

name affects how the name is printed.:

```
sage: T.<unif> = Qp(5, print_mode='terse'); c = T(-707/5); c
95367431639918/unif + O(unif^19)
sage: d = T(-707/5^10); d
95367431639918/unif^10 + O(unif^10)
```

print_max_terms, *print_sep* and *print_alphabet* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

1. **digits**: elements are displayed as a string of base p digits

Restriction: you can only use the digits printing mode for small primes. Namely, p must be less than the length of the alphabet tuple (default alphabet has length 62).:

```
sage: R = Qp(5, print_mode='digits'); a = R(70700); repr(a)
'...4230300'
sage: b = R(-70700); repr(b)
'...444444444444440214200'
sage: c = R(-707/5); repr(c)
'...444444444444443413.3'
sage: d = R(-707/5^2); repr(d)
'...44444444444444341.33'
```

Note that it's not possible to read off the precision from the representation in this mode.

print_max_terms limits the number of digits that are printed. Note that if the valuation of the element is very negative, more digits will be printed.:

```
sage: S = Qp(5, print_mode='digits', print_max_terms=4); b = S(-70700); repr(b)
'...214200'
sage: d = S(-707/5^2); repr(d)
'...41.33'
sage: e = S(-707/5^6); repr(e)
'...?.434133'
sage: f = S(-707/5^6,absprec=-2); repr(f)
'...?.??4133'
sage: g = S(-707/5^4); repr(g)
'...?.4133'
```

print_alphabet controls the symbols used to substitute for digits greater than 9.

Defaults to ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'):

```
sage: T = Qp(5, print_mode='digits', print_max_terms=4, print_alphabet=('1','2','3','4','5')); b
'...325311'
```

print_pos, *name* and *print_sep* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

1. **bars**: elements are displayed as a string of base p digits with separators:

   ```
   sage: R = Qp(5, print_mode='bars'); a = R(70700); repr(a)
   '...4|2|3|0|3|0|0'
   sage: b = R(-70700); repr(b)
   '...4|4|4|4|4|4|4|4|4|4|4|4|4|4|4|0|2|1|4|2|0|0'
   sage: d = R(-707/5^2); repr(d)
   '...4|4|4|4|4|4|4|4|4|4|4|4|4|4|4|3|4|1|.|3|3'
   ```

Again, note that it's not possible to read of the precision from the representation in this mode.

print_pos controls whether the digits can be negative.:

```
sage: S = Qp(5, print_mode='bars',print_pos=False); b = S(-70700); repr(b)
'...-1|0|2|2|-1|2|0|0'
```

print_max_terms limits the number of digits that are printed. Note that if the valuation of the element is very negative, more digits will be printed.:

```
sage: T = Qp(5, print_mode='bars', print_max_terms=4); b = T(-70700); repr(b)
'...2|1|4|2|0|0'
sage: d = T(-707/5^2); repr(d)
'...4|1|.|3|3'
sage: e = T(-707/5^6); repr(e)
'...|.|4|3|4|1|3|3'
sage: f = T(-707/5^6,absprec=-2); repr(f)
'...|.|?|?|4|1|3|3'
sage: g = T(-707/5^4); repr(g)
'...|.|4|1|3|3'
```

print_sep controls the separation character.:

```
sage: U = Qp(5, print_mode='bars', print_sep='][', a = U(70700); repr(a)
'...4][2][3][0][3][0][0'
```

name and *print_alphabet* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

EXAMPLES:

```
sage: K = Qp(15, check=False); a = K(999); a
9 + 6*15 + 4*15^2 + O(15^20)
```

create_key(*p, prec=20, type='capped-rel', print_mode=None, halt=40, names=None, ram_name=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_terms=None, check=True*)

Creates a key from input parameters for `Qp`.

See the documentation for `Qp` for more information.

TESTS:

```
sage: Qp.create_key(5,40)
(5, 40, 'capped-rel', 'series', '5', True, '|', ('0', '1', '2', '3', '4', '5', '6', '7', '8'
```

create_object(*version, key*)

Creates an object using a given key.

See the documentation for `Qp` for more information.

TESTS:

```
sage: Qp.create_object((3,4,2),(5, 40, 'capped-rel', 'series', '5', True, '|', ('0', '1', '2
5-adic Field with capped relative precision 40
```

Qq(*q, prec=20, type='capped-rel', modulus=None, names=None, print_mode=None, halt=40, ram_name=None, res_name=None, print_pos=None, print_sep=None, print_max_ram_terms=None, print_max_unram_terms=None, print_max_terse_terms=None, check=True*)

Given a prime power $q = p^n$, return the unique unramified extension of $\mathbb{Q}_p$ of degree n.

INPUT:

- `q` – integer: the prime power in $\mathbb{Q}_q$. OR, if check=False, a factorization object or single element list `[(p, n)]` where `p` is a prime and `n` a positive integer.
- `prec` – integer (default: `20`) the precision cap of the field. Individual elements keep track of their own precision. See TYPES and PRECISION below.
- `type` – string (default: `'capped-rel'`) Valid types are `'capped-rel'` and `'lazy'` (though `'lazy'` doesn't currently work). See TYPES and PRECISION below

- `modulus` – polynomial (default `None`) A polynomial defining an unramified extension of $\mathbb{Q}_p$. See MODULUS below.
- `names` – string or tuple (`None` is only allowed when $q = p$). The name of the generator, reducing to a generator of the residue field.
- `print_mode` – string (default: `None`). Valid modes are `'series'`, `'val-unit'`, `'terse'`, and `'bars'`. See PRINTING below.
- `halt` – currently irrelevant (to be used for lazy fields)
- `ram_name` – string (defaults to string representation of p if None). `ram_name` controls how the prime is printed. See PRINTING below.
- `res_name` – string (defaults to `None`, which corresponds to adding a `'0'` to the end of the name). Controls how elements of the reside field print.
- `print_pos` – bool (default `None`) Whether to only use positive integers in the representations of elements. See PRINTING below.
- `print_sep` – string (default `None`) The separator character used in the `'bars'` mode. See PRINTING below.
- `print_max_ram_terms` – integer (default `None`) The maximum number of powers of p shown. See PRINTING below.
- `print_max_unram_terms` – integer (default `None`) The maximum number of entries shown in a coefficient of p. See PRINTING below.
- `print_max_terse_terms` – integer (default `None`) The maximum number of terms in the polynomial representation of an element (using `'terse'`). See PRINTING below.
- `check` – bool (default `True`) whether to check inputs.

OUTPUT:

- The corresponding unramified p-adic field.

TYPES AND PRECISION:

There are two types of precision for a p-adic element. The first is relative precision, which gives the number of known p-adic digits:

```
sage: R.<a> = Qq(25, 20, 'capped-rel', print_mode='series'); b = 25*a; b
a*5^2 + O(5^22)
sage: b.precision_relative()
20
```

The second type of precision is absolute precision, which gives the power of p that this element is defined modulo:

```
sage: b.precision_absolute()
22
```

There are two types of unramified p-adic fields: capped relative fields and lazy fields.

In the capped relative case, the relative precision of an element is restricted to be at most a certain value, specified at the creation of the field. Individual elements also store their own precision, so the effect of various arithmetic operations on precision is tracked. When you cast an exact element into a capped relative field, it truncates it to the precision cap of the field.:

```
sage: R.<a> = Qq(9, 5, 'capped-rel', print_mode='series'); b = (1+2*a)^4; b
2 + (2*a + 2)*3 + (2*a + 1)*3^2 + O(3^5)
sage: c = R(3249); c
3^2 + 3^4 + 3^5 + 3^6 + O(3^7)
sage: b + c
2 + (2*a + 2)*3 + (2*a + 2)*3^2 + 3^4 + O(3^5)
```

The lazy case will eventually support elements that can increase their precision upon request. It is not currently implemented.

MODULUS:

The modulus needs to define an unramified extension of $\mathbb{Q}_p$: when it is reduced to a polynomial over $\mathbb{F}_p$ it should be irreducible.

The modulus can be given in a number of forms.

1. A **polynomial**.

The base ring can be $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}_p$, $\mathbb{Q}_p$, $\mathbb{F}_p$.:

```
sage: P.<x> = ZZ[]
sage: R.<a> = Qq(27, modulus = x^3 + 2*x + 1); R.modulus()
(1 + O(3^20))*x^3 + (O(3^20))*x^2 + (2 + O(3^20))*x + (1 + O(3^20))
sage: P.<x> = QQ[]
sage: S.<a> = Qq(27, modulus = x^3 + 2*x + 1)
sage: P.<x> = Zp(3)[]
sage: T.<a> = Qq(27, modulus = x^3 + 2*x + 1)
sage: P.<x> = Qp(3)[]
sage: U.<a> = Qq(27, modulus = x^3 + 2*x + 1)
sage: P.<x> = GF(3)[]
sage: V.<a> = Qq(27, modulus = x^3 + 2*x + 1)
```

Which form the modulus is given in has no effect on the unramified extension produced:

```
sage: R == S, S == T, T == U, U == V
(True, True, True, False)
```

unless the precision of the modulus differs. In the case of V, the modulus is only given to precision 1, so the resulting field has a precision cap of 1.:

```
sage: V.precision_cap()
1
sage: U.precision_cap()
20
sage: P.<x> = Qp(3)[]
sage: modulus = x^3 + (2 + O(3^7))*x + (1 + O(3^10))
sage: modulus
(1 + O(3^20))*x^3 + (2 + O(3^7))*x + (1 + O(3^10))
sage: W.<a> = Qq(27, modulus = modulus); W.precision_cap()
7
```

1. The modulus can also be given as a **symbolic expression**.:

```
sage: x = var('x')
sage: X.<a> = Qq(27, modulus = x^3 + 2*x + 1); X.modulus()
(1 + O(3^20))*x^3 + (O(3^20))*x^2 + (2 + O(3^20))*x + (1 + O(3^20))
sage: X == R
True
```

By default, the polynomial chosen is the standard lift of the generator chosen for $\mathbb{F}_q$.:

```
sage: GF(125, 'a').modulus()
x^3 + 3*x + 3
sage: Y.<a> = Qq(125); Y.modulus()
(1 + O(5^20))*x^3 + (O(5^20))*x^2 + (3 + O(5^20))*x + (3 + O(5^20))
```

However, you can choose another polynomial if desired (as long as the reduction to $\mathbb{F}_p[x]$ is irreducible).:

```
sage: P.<x> = ZZ[]
sage: Z.<a> = Qq(125, modulus = x^3 + 3*x^2 + x + 1); Z.modulus()
(1 + O(5^20))*x^3 + (3 + O(5^20))*x^2 + (1 + O(5^20))*x + (1 + O(5^20))
sage: Y == Z
False
```

PRINTING:

There are many different ways to print p-adic elements. The way elements of a given field print is controlled by options passed in at the creation of the field. There are four basic printing modes (`'series'`, `'val-unit'`, `'terse'` and `'bars'`; `'digits'` is not available), as well as various options that either hide some information in the print representation or sometimes make print representations more compact. Note that the printing options affect whether different p-adic fields are considered equal.

1. **series**: elements are displayed as series in p.:

```
sage: R.<a> = Qq(9, 20, 'capped-rel', print_mode='series'); (1+2*a)^4
2 + (2*a + 2)*3 + (2*a + 1)*3^2 + O(3^20)
sage: -3*(1+2*a)^4
3 + a*3^2 + 3^3 + (2*a + 2)*3^4 + (2*a + 2)*3^5 + (2*a + 2)*3^6 + (2*a + 2)*3^7 + (2*a + 2)*
sage: ~(3*a+18)
(a + 2)*3^-1 + 1 + 2*3 + (a + 1)*3^2 + 3^3 + 2*3^4 + (a + 1)*3^5 + 3^6 + 2*3^7 + (a + 1)*3^8
```

print_pos controls whether negatives can be used in the coefficients of powers of p.:

```
sage: S.<b> = Qq(9, print_mode='series', print_pos=False); (1+2*b)^4
-1 - b*3 - 3^2 + (b + 1)*3^3 + O(3^20)
sage: -3*(1+2*b)^4
3 + b*3^2 + 3^3 + (-b - 1)*3^4 + O(3^21)
```

ram_name controls how the prime is printed.:

```
sage: T.<d> = Qq(9, print_mode='series', ram_name='p'); 3*(1+2*d)^4
2*p + (2*d + 2)*p^2 + (2*d + 1)*p^3 + O(p^21)
```

print_max_ram_terms limits the number of powers of p that appear.:

```
sage: U.<e> = Qq(9, print_mode='series', print_max_ram_terms=4); repr(-3*(1+2*e)^4)
'3 + e*3^2 + 3^3 + (2*e + 2)*3^4 + ... + O(3^21)'
```

print_max_unram_terms limits the number of terms that appear in a coefficient of a power of p.:

```
sage: V.<f> = Qq(128, prec = 8, print_mode='series'); repr((1+f)^9)
'(f^3 + 1) + (f^5 + f^4 + f^3 + f^2)*2 + (f^6 + f^5 + f^4 + f + 1)*2^2 + (f^5 + f^4 + f^2 + f +
sage: V.<f> = Qq(128, prec = 8, print_mode='series', print_max_unram_terms = 3); repr((1+f)^9)
'(f^3 + 1) + (f^5 + f^4 + ... + f^2)*2 + (f^6 + f^5 + ... + 1)*2^2 + (f^5 + f^4 + ... + 1)*2^3 +
sage: V.<f> = Qq(128, prec = 8, print_mode='series', print_max_unram_terms = 2); repr((1+f)^9)
'(f^3 + 1) + (f^5 + ... + f^2)*2 + (f^6 + ... + 1)*2^2 + (f^5 + ... + 1)*2^3 + (f^6 + ... + 1)*2
sage: V.<f> = Qq(128, prec = 8, print_mode='series', print_max_unram_terms = 1); repr((1+f)^9)
'(f^3 + ...) + (f^5 + ...)*2 + (f^6 + ...)*2^2 + (f^5 + ...)*2^3 + (f^6 + ...)*2^4 + (f^5 + ...)
sage: V.<f> = Qq(128, prec = 8, print_mode='series', print_max_unram_terms = 0); repr((1+f)^9 -
'(...)*2 + (...)*2^2 + (...)*2^3 + (...)*2^4 + (...)*2^5 + (...)*2^6 + (...)*2^7 + O(2^8)'
```

print_sep and *print_max_terse_terms* have no effect.

Note that print options affect equality:

```
sage: R == S, R == T, R == U, R == V, S == T, S == U, S == V, T == U, T == V, U == V
(False, False, False, False, False, False, False, False, False, False)
```

1. **val-unit**: elements are displayed as $p^k u$:

```
sage: R.<a> = Qq(9, 7, print_mode='val-unit'); b = (1+3*a)^9 - 1; b
3^3 * (15 + 64*a) + O(3^7)
sage: ~b
3^-3 * (41 + a) + O(3)
```

print_pos controls whether to use a balanced representation or not.:

```
sage: S.<a> = Qq(9, 7, print_mode='val-unit', print_pos=False); b = (1+3*a)^9 - 1; b
3^3 * (15 - 17*a) + O(3^7)
sage: ~b
3^-3 * (-40 + a) + O(3)
```

ram_name affects how the prime is printed.:

```
sage: A.<x> = Qp(next_prime(10^6), print_mode='val-unit')[]
sage: T.<a> = Qq(next_prime(10^6)^3, 4, print_mode='val-unit', ram_name='p', modulus=x^3+385831*
p^-2 * (503009563508519137754940 + 704413692798200940253892*a + 968097057817740999537581*a^2) +
sage: b * (a^2 + a - 4)
p^-2 * 1 + O(p^2)
```

print_max_terse_terms controls how many terms of the polynomial appear in the unit part.:

```
sage: U.<a> = Qq(17^4, 6, print_mode='val-unit', print_max_terse_terms=3); b = ~(17*(a^3-a+14));
17^-1 * (22110411 + 11317400*a + 20656972*a^2 + ...) + O(17^5)
sage: b*17*(a^3-a+14)
1 + O(17^6)
```

print_sep, *print_max_ram_terms* and *print_max_unram_terms* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

1. **terse**: elements are displayed as a polynomial of degree less than the degree of the extension.:

```
sage: R.<a> = Qq(125, print_mode='terse')
sage: (a+5)^177
68210977979428 + 90313850704069*a + 73948093055069*a^2 + O(5^20)
sage: (a/5+1)^177
68210977979428/5^177 + 90313850704069/5^177*a + 73948093055069/5^177*a^2 + O(5^-157)
```

As of version 3.3, if coefficients of the polynomial are non-integral, they are always printed with an explicit power of p in the denominator.:

```
sage: 5*a + a^2/25
5*a + 1/5^2*a^2 + O(5^18)
```

print_pos controls whether to use a balanced representation or not.:

```
sage: (a-5)^6
22864 + 95367431627998*a + 8349*a^2 + O(5^20)
sage: S.<a> = Qq(125, print_mode='terse', print_pos=False); b = (a-5)^6; b
22864 - 12627*a + 8349*a^2 + O(5^20)
sage: (a - 1/5)^6
-20624/5^6 + 18369/5^5*a + 1353/5^3*a^2 + O(5^14)
```

ram_name affects how the prime is printed.:

```
sage: T.<a> = Qq(125, print_mode='terse', ram_name='p'); (a - 1/5)^6
95367431620001/p^6 + 18369/p^5*a + 1353/p^3*a^2 + O(p^14)
```

print_max_terse_terms controls how many terms of the polynomial are shown.:

```
sage: U.<a> = Qq(625, print_mode='terse', print_max_terse_terms=2); (a-1/5)^6
106251/5^6 + 49994/5^5*a + ... + O(5^14)
```

print_sep, *print_max_ram_terms* and *print_max_unram_terms* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

1. **digits**: This print mode is not available when the residue field is not prime.

It might make sense to have a dictionary for small fields, but this isn't implemented.

1. **bars**: elements are displayed in a similar fashion to series, but more compactly.:

```
sage: R.<a> = Qq(125); (a+5)^6
(4*a^2 + 3*a + 4) + (3*a^2 + 2*a)*5 + (a^2 + a + 1)*5^2 + (3*a + 2)*5^3 + (3*a^2 + a + 3)*5^
sage: R.<a> = Qq(125, print_mode='bars', prec=8); repr((a+5)^6)
'...[2, 3, 2]|[3, 1, 3]|[2, 3]|[1, 1, 1]|[0, 2, 3]|[4, 3, 4]'
sage: repr((a-5)^6)
'...[0, 4]|[1, 4]|[2, 0, 2]|[1, 4, 3]|[2, 3, 1]|[4, 4, 3]|[2, 4, 4]|[4, 3, 4]'
```

Note that elements with negative valuation are shown with a decimal point at valuation 0.:

```
sage: repr((a+1/5)^6)
'...[3]|[4, 1, 3]|.|[1, 2, 3]|[3, 3]|[0, 0, 3]|[0, 1]|[0, 1]|[1]'
sage: repr((a+1/5)^2)
'...[0, 0, 1]|.|[0, 2]|[1]'
```

If not enough precision is known, `'?'` is used instead.:

```
sage: repr((a+R(1/5,relprec=3))^7)
'...|.|?|?|?|?|[0, 1, 1]|[0, 2]|[1]'
```

Note that it's not possible to read of the precision from the representation in this mode.:

```
sage: b = a + 3; repr(b)
'...[3, 1]'
sage: c = a + R(3, 4); repr(c)
'...[3, 1]'
sage: b.precision_absolute()
8
```

```
sage: c.precision_absolute()
4
```

print_pos controls whether the digits can be negative.:

```
sage: S.<a> = Qq(125, print_mode='bars', print_pos=False); repr((a-5)^6)
'...[1, -1, 1]|[2, 1, -2]|[2, 0, -2]|[-2, -1, 2]|[0, 0, -1]|[-2]|[-1, -2, -1]'
sage: repr((a-1/5)^6)
'...[0, 1, 2]|[-1, 1, 1]|.|[-2, -1, -1]|[2, 2, 1]|[0, 0, -2]|[0, -1]|[0, -1]|[1]'
```

print_max_ram_terms controls the maximum number of "digits" shown. Note that this puts a cap on the relative precision, not the absolute precision.:

```
sage: T.<a> = Qq(125, print_mode='bars', print_max_ram_terms=3, print_pos=False); repr((a-5)^6)
'...[0, 0, -1]|[-2]|[-1, -2, -1]'
sage: repr(5*(a-5)^6+50)
'...[0, 0, -1]|[]|[-1, -2, -1]|[]'
```

However, if the element has negative valuation, digits are shown up to the decimal point.:

```
sage: repr((a-1/5)^6)
'...|.|[-2, -1, -1]|[2, 2, 1]|[0, 0, -2]|[0, -1]|[0, -1]|[1]'
```

print_sep controls the separating character (`'|'` by default).:

```
sage: U.<a> = Qq(625, print_mode='bars', print_sep=''); b = (a+5)^6; repr(b)
'...[0, 1][4, 0, 2][3, 2, 2, 3][4, 2, 2, 4][0, 3][1, 1, 3][3, 1, 4, 1]'
```

print_max_unram_terms controls how many terms are shown in each "digit":

```
sage: with local_print_mode(U, {'max_unram_terms': 3}): repr(b)
'...[0, 1][4,..., 0, 2][3,..., 2, 3][4,..., 2, 4][0, 3][1,..., 1, 3][3,..., 4, 1]'
sage: with local_print_mode(U, {'max_unram_terms': 2}): repr(b)
'...[0, 1][4,..., 2][3,..., 3][4,..., 4][0, 3][1,..., 3][3,..., 1]'
sage: with local_print_mode(U, {'max_unram_terms': 1}): repr(b)
'...[..., 1][..., 2][..., 3][..., 4][..., 3][..., 3][..., 1]'
sage: with local_print_mode(U, {'max_unram_terms':0}): repr(b-75*a)
'...[...][...][...][...][][...][...]'
```

ram_name and *print_max_terse_terms* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

EXAMPLES

Unlike for `Qp`, you can't create `Qq(N)` when `N` is not a prime power.

However, you can use `check=False` to pass in a pair in order to not have to factor. If you do so, you need to use names explicitly rather than the `R.<a>` syntax.:

```
sage: p = next_prime(2^123)
sage: k = Qp(p)
sage: R.<x> = k[]
sage: K = Qq([(p, 5)], modulus=x^5+x+4, names='a', ram_name='p', print_pos=False, check=False)
sage: K.0^5
(-a - 4) + O(p^20)
```

In tests on sage.math.washington.edu, the creation of K as above took an average of 1.58ms, while:

```
sage: K = Qq(p^5, modulus=x^5+x+4, names='a', ram_name='p', print_pos=False, check=True)
```

took an average of 24.5ms. Of course, with smaller primes these savings disappear.

QqCR(*q, prec=20, modulus=None, names=None, print_mode=None, halt=40, ram_name=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_ram_terms=None, print_max_unram_terms=None, print_max_terse_terms=None, check=True*)
A shorcut function to create capped relative unramified p-adic fields.

Same functionality as Qq. See documentation for Qq for a description of the input paramters.

EXAMPLES:

```
sage: R.<a> = QqCR(25, 40); R
Unramified Extension of 5-adic Field with capped relative precision 40 in a defined by (1 + O(5^
```

ZpCA(*p, prec=20, print_mode=None, halt=40, names=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_terms=None, check=True*)
A shorcut function to create capped absolute p-adic rings.

See documentation for Zp for a description of the input paramters.

EXAMPLES:

```
sage: ZpCA(5, 40)
5-adic Ring with capped absolute precision 40
```

ZpCR(*p, prec=20, print_mode=None, halt=40, names=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_terms=None, check=True*)
A shorcut function to create capped relative p-adic rings.

Same functionality as Zp. See documentation for Zp for a description of the input paramters.

EXAMPLES:

```
sage: ZpCR(5, 40)
5-adic Ring with capped relative precision 40
```

ZpFM(*p, prec=20, print_mode=None, halt=40, names=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_terms=None, check=True*)
A shorcut function to create fixed modulus p-adic rings.

See documentation for Zp for a description of the input paramters.

EXAMPLES:

```
sage: ZpFM(5, 40)
5-adic Ring of fixed modulus 5^40
```

class Zp_class()
A creation function for p-adic rings.

INPUT:

- p – integer: the p in $\mathbb{Z}_p$
- prec – integer (default: 20) the precision cap of the ring. Except for the fixed modulus case, individual elements keep track of their own precision. See TYPES and PRECISION below.
- type – string (default: 'capped-rel') Valid types are 'capped-rel', 'capped-abs', 'fixed-mod' and 'lazy' (though lazy is not yet implemented). See TYPES and PRECISION below

- `print_mode` – string (default: `None`). Valid modes are `'series'`, `'val-unit'`, `'terse'`, `'digits'`, and `'bars'`. See PRINTING below
- `halt` – currently irrelevant (to be used for lazy fields)
- `names` – string or tuple (defaults to a string representation of p). What to use whenever p is printed.
- `print_pos` – bool (default `None`) Whether to only use positive integers in the representations of elements. See PRINTING below.
- `print_sep` – string (default `None`) The separator character used in the `'bars'` mode. See PRINTING below.
- `print_alphabet` – tuple (default `None`) The encoding into digits for use in the `'digits'` mode. See PRINTING below.
- `print_max_terms` – integer (default `None`) The maximum number of terms shown. See PRINTING below.
- `check` – bool (default `True`) whether to check if p is prime. Non-prime input may cause seg-faults (but can also be useful for base n expansions for example)

OUTPUT:

- The corresponding p-adic ring.

TYPES AND PRECISION:

There are two types of precision for a p-adic element. The first is relative precision, which gives the number of known p-adic digits:

```
sage: R = Zp(5, 20, 'capped-rel', 'series'); a = R(675); a
2*5^2 + 5^4 + O(5^22)
sage: a.precision_relative()
20
```

The second type of precision is absolute precision, which gives the power of p that this element is defined modulo:

```
sage: a.precision_absolute()
22
```

There are four types of p-adic rings: capped relative rings (type= `'capped-rel'`), capped absolute rings (type= `'capped-abs'`), fixed modulus ring (type= `'fixed-mod'`) and lazy rings (type= `'lazy'`).

In the capped relative case, the relative precision of an element is restricted to be at most a certain value, specified at the creation of the field. Individual elements also store their own precision, so the effect of various arithmetic operations on precision is tracked. When you cast an exact element into a capped relative field, it truncates it to the precision cap of the field.:

```
sage: R = Zp(5, 5, 'capped-rel', 'series'); a = R(4006); a
1 + 5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(4025); b
5^2 + 2*5^3 + 5^4 + 5^5 + O(5^7)
sage: a + b
1 + 5 + 5^2 + 4*5^3 + 2*5^4 + O(5^5)
```

In the capped absolute type, instead of having a cap on the relative precision of an element there is instead a cap on the absolute precision. Elements still store their own precisions, and as with the capped relative case, exact elements are truncated when cast into the ring.:

```
sage: R = Zp(5, 5, 'capped-abs', 'series'); a = R(4005); a
5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(4025); b
5^2 + 2*5^3 + 5^4 + O(5^5)
sage: a * b
5^3 + 2*5^4 + O(5^5)
sage: (a * b) // 5^3
1 + 2*5 + O(5^2)
```

The fixed modulus type is the leanest of the p-adic rings: it is basically just a wrapper around $\mathbb{Z}/p^n\mathbb{Z}$ providing a unified interface with the rest of the p-adics. This is the type you should use if your sole interest is speed. It does not track precision of elements.:

```
sage: R = Zp(5,5,'fixed-mod','series'); a = R(4005); a
5 + 2*5^3 + 5^4 + O(5^5)
sage: a // 5
1 + 2*5^2 + 5^3 + O(5^5)
```

The lazy case will eventually support elements that can increase their precision upon request. It is not currently implemented.

PRINTING

There are many different ways to print p-adic elements. The way elements of a given ring print is controlled by options passed in at the creation of the ring. There are five basic printing modes (series, val-unit, terse, digits and bars), as well as various options that either hide some information in the print representation or sometimes make print representations more compact. Note that the printing options affect whether different p-adic fields are considered equal.

1. **series**: elements are displayed as series in p.:

```
sage: R = Zp(5, print_mode='series'); a = R(70700); a
3*5^2 + 3*5^4 + 2*5^5 + 4*5^6 + O(5^22)
sage: b = R(-70700); b
2*5^2 + 4*5^3 + 5^4 + 2*5^5 + 4*5^7 + 4*5^8 + 4*5^9 + 4*5^10 + 4*5^11 + 4*5^12 + 4*5^13 + 4*
```

print_pos controls whether negatives can be used in the coefficients of powers of p.:

```
sage: S = Zp(5, print_mode='series', print_pos=False); a = S(70700); a
-2*5^2 + 5^3 - 2*5^4 - 2*5^5 + 5^7 + O(5^22)
sage: b = S(-70700); b
2*5^2 - 5^3 + 2*5^4 + 2*5^5 - 5^7 + O(5^22)
```

print_max_terms limits the number of terms that appear.:

```
sage: T = Zp(5, print_mode='series', print_max_terms=4); b = R(-70700); b
2*5^2 + 4*5^3 + 5^4 + 2*5^5 + ... + O(5^22)
```

names affects how the prime is printed.:

```
sage: U.<p> = Zp(5); p
p + O(p^21)
```

print_sep and *print_alphabet* have no effect.

Note that print options affect equality:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

1. **val-unit**: elements are displayed as $p^k u$:

```
sage: R = Zp(5, print_mode='val-unit'); a = R(70700); a
5^2 * 2828 + O(5^22)
sage: b = R(-707*5); b
5 * 95367431639918 + O(5^21)
```

print_pos controls whether to use a balanced representation or not.:

```
sage: S = Zp(5, print_mode='val-unit', print_pos=False); b = S(-70700); b
5^2 * (-2828) + O(5^22)
```

names affects how the prime is printed.:

```
sage: T = Zp(5, print_mode='val-unit', names='pi'); a = T(70700); a
pi^2 * 2828 + O(pi^22)
```

print_max_terms, *print_sep* and *print_alphabet* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

1. **terse**: elements are displayed as an integer in base 10:

```
sage: R = Zp(5, print_mode='terse'); a = R(70700); a
70700 + O(5^22)
sage: b = R(-70700); b
2384185790944925 + O(5^22)
```

print_pos controls whether to use a balanced representation or not.:

```
sage: S = Zp(5, print_mode='terse', print_pos=False); b = S(-70700); b
-70700 + O(5^22)
```

name affects how the name is printed. Note that this interacts with the choice of shorter string for denominators.:

```
sage: T.<unif> = Zp(5, print_mode='terse'); c = T(-707); c
95367431639918 + O(unif^20)
```

print_max_terms, *print_sep* and *print_alphabet* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

1. **digits**: elements are displayed as a string of base p digits

Restriction: you can only use the digits printing mode for small primes. Namely, p must be less than the length of the alphabet tuple (default alphabet has length 62).:

```
sage: R = Zp(5, print_mode='digits'); a = R(70700); repr(a)
'...4230300'
sage: b = R(-70700); repr(b)
'...444444444444440214200'
```

Note that it's not possible to read off the precision from the representation in this mode.

print_max_terms limits the number of digits that are printed.:

```
sage: S = Zp(5, print_mode='digits', print_max_terms=4); b = S(-70700); repr(b)
'...214200'
```

print_alphabet controls the symbols used to substitute for digits greater than 9. Defaults to ('0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z'):

```
sage: T = Zp(5, print_mode='digits', print_max_terms=4, print_alphabet=('1','2','3','4','5')); b
'...325311'
```

print_pos, *name* and *print_sep* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, S == T
(False, False, False)
```

1. **bars**: elements are displayed as a string of base p digits with separators

   sage: R = Zp(5, print_mode='bars'); a = R(70700); repr(a) '...4|2|3|0|3|0|0' sage: b = R(-70700); repr(b) '...4|4|4|4|4|4|4|4|4|4|4|4|4|4|4|0|2|1|4|2|0|0'

Again, note that it's not possible to read of the precision from the representation in this mode.

print_pos controls whether the digits can be negative.:

```
sage: S = Zp(5, print_mode='bars',print_pos=False); b = S(-70700); repr(b)
'...-1|0|2|2|-1|2|0|0'
```

print_max_terms limits the number of digits that are printed.:

```
sage: T = Zp(5, print_mode='bars', print_max_terms=4); b = T(-70700); repr(b)
'...2|1|4|2|0|0'
```

print_sep controls the separation character.:

```
sage: U = Zp(5, print_mode='bars', print_sep=']['); a = U(70700); repr(a)
'...4][2][3][0][3][0][0'
```

name and *print_alphabet* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

EXAMPLES:

We allow non-prime p, but only if `check = False`. Note that some features will not work.:

```
sage: K = Zp(15, check=False); a = K(999); a
9 + 6*15 + 4*15^2 + O(15^20)
```

We create rings with various parameters:

```
sage: Zp(7)
7-adic Ring with capped relative precision 20
sage: Zp(9)
...
ValueError: p must be prime
sage: Zp(17, 5)
17-adic Ring with capped relative precision 5
sage: Zp(17, 5)(-1)
16 + 16*17 + 16*17^2 + 16*17^3 + 16*17^4 + O(17^5)
```

It works even with a fairly huge cap:

```
sage: Zp(next_prime(10^50), 100000)
10000000000000000000000000000000000000000000000151-adic Ring with capped relative precision 100
```

We create each type of ring:

```
sage: Zp(7, 20, 'capped-rel')
7-adic Ring with capped relative precision 20
sage: Zp(7, 20, 'fixed-mod')
7-adic Ring of fixed modulus 7^20
sage: Zp(7, 20, 'capped-abs')
7-adic Ring with capped absolute precision 20
```

We create a capped relative ring with each print mode:

```
sage: k = Zp(7, 8, print_mode='series'); k
7-adic Ring with capped relative precision 8
sage: k(7*(19))
5*7 + 2*7^2 + O(7^9)
sage: k(7*(-19))
2*7 + 4*7^2 + 6*7^3 + 6*7^4 + 6*7^5 + 6*7^6 + 6*7^7 + 6*7^8 + O(7^9)
```

```
sage: k = Zp(7, print_mode='val-unit'); k
7-adic Ring with capped relative precision 20
sage: k(7*(19))
7 * 19 + O(7^21)
sage: k(7*(-19))
7 * 79792266297611982 + O(7^21)
```

```
sage: k = Zp(7, print_mode='terse'); k
7-adic Ring with capped relative precision 20
sage: k(7*(19))
133 + O(7^21)
sage: k(7*(-19))
558545864083283874 + O(7^21)
```

Note that p-adic rings are cached (via weak references):

```
sage: a = Zp(7); b = Zp(7)
sage: a is b
True
```

We create some elements in various rings:

```
sage: R = Zp(5); a = R(4); a
4 + O(5^20)
sage: S = Zp(5, 10, type = 'capped-abs'); b = S(2); b
2 + O(5^10)
sage: a + b
1 + 5 + O(5^10)
```

create_key(*p, prec=20, type='capped-rel', print_mode=None, halt=40, names=None, ram_name=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_terms=None, check=True*)

Creates a key from input parameters for `Zp`.

See the documentation for `Zp` for more information.

TESTS:

```
sage: Zp.create_key(5,40)
(5, 40, 'capped-rel', 'series', '5', True, '|', ('0', '1', '2', '3', '4', '5', '6', '7', '8'
```

create_object(*version, key*)

Creates an object using a given key.

See the documentation for `Zp` for more information.

TESTS:

```
sage: Zp.create_object((3,4,2),(5, 40, 'capped-rel', 'series', '5', True, '|', ('0', '1', '2
5-adic Ring with capped relative precision 40
```

Zq(*q, prec=20, type='capped-abs', modulus=None, names=None, print_mode=None, halt=40, ram_name=None, res_name=None, print_pos=None, print_sep=None, print_max_ram_terms=None, print_max_unram_terms=None, print_max_terse_terms=None, check=True*)

Given a prime power $q = p^n$, return the unique unramified extension of $\mathbb{Z}_p$ of degree n.

INPUT:

- `q` – integer: the prime power in $\mathbb{Q}_q$. OR, if `check=False`, a factorization object or single element list `[(p, n)]` where `p` is a prime and `n` a positive integer.
- `prec` – integer (default: `20`) the precision cap of the field. Individual elements keep track of their own precision. See TYPES and PRECISION below.
- `type` – string (default: `'capped-rel'`) Valid types are `'capped-rel'` and `'lazy'` (though `'lazy'` doesn't currently work). See TYPES and PRECISION below
- modulus – polynomial (default None) A polynomial defining an unramified extension of $\mathbb{Z}_p$. See MODULUS below.
- `names` – string or tuple (`None` is only allowed when $q = p$). The name of the generator, reducing to a generator of the residue field.
- `print_mode` – string (default: `None`). Valid modes are `'series'`, `'val-unit'`, `'terse'`, and `'bars'`. See PRINTING below.
- `halt` – currently irrelevant (to be used for lazy fields)
- `ram_name` – string (defaults to string representation of p if None). `ram_name` controls how the prime is printed. See PRINTING below.
- `res_name` – string (defaults to `None`, which corresponds to adding a `'0'` to the end of the name). Controls how elements of the reside field print.
- `print_pos` – bool (default `None`) Whether to only use positive integers in the representations of elements. See PRINTING below.
- `print_sep` – string (default `None`) The separator character used in the `'bars'` mode. See PRINTING below.

- print_max_ram_terms – integer (default None) The maximum number of powers of p shown. See PRINTING below.
- print_max_unram_terms – integer (default None) The maximum number of entries shown in a coefficient of p. See PRINTING below.
- print_max_terse_terms – integer (default None) The maximum number of terms in the polynomial representation of an element (using 'terse'). See PRINTING below.
- check – bool (default True) whether to check inputs.

OUTPUT:

- The corresponding unramified p-adic ring.

TYPES AND PRECISION:

There are two types of precision for a p-adic element. The first is relative precision, which gives the number of known p-adic digits:

```
sage: R.<a> = Zq(25, 20, 'capped-rel', print_mode='series'); b = 25*a; b
a*5^2 + O(5^22)
sage: b.precision_relative()
20
```

The second type of precision is absolute precision, which gives the power of p that this element is defined modulo:

```
sage: b.precision_absolute()
22
```

There are four types of unramified p-adic rings: capped relative rings, capped absolute rings, fixed modulus rings, and lazy rings.

In the capped relative case, the relative precision of an element is restricted to be at most a certain value, specified at the creation of the field. Individual elements also store their own precision, so the effect of various arithmetic operations on precision is tracked. When you cast an exact element into a capped relative field, it truncates it to the precision cap of the field.:

```
sage: R.<a> = Zq(9, 5, 'capped-rel', print_mode='series'); b = (1+2*a)^4; b
2 + (2*a + 2)*3 + (2*a + 1)*3^2 + O(3^5)
sage: c = R(3249); c
3^2 + 3^4 + 3^5 + 3^6 + O(3^7)
sage: b + c
2 + (2*a + 2)*3 + (2*a + 2)*3^2 + 3^4 + O(3^5)
```

One can invert non-units: the result is in the fraction field.:

```
sage: d = ~(3*b+c); d
2*3^-1 + (a + 1) + (a + 1)*3 + a*3^3 + O(3^4)
sage: d.parent()
Unramified Extension of 3-adic Field with capped relative precision 5 in a defined by (1 + O(3^5
```

The capped absolute case is the same as the capped relative case, except that the cap is on the absolute precision rather than the relative precision.:

```
sage: R.<a> = Zq(9, 5, 'capped-abs', print_mode='series'); b = 3*(1+2*a)^4; b
2*3 + (2*a + 2)*3^2 + (2*a + 1)*3^3 + O(3^5)
sage: c = R(3249); c
3^2 + 3^4 + O(3^5)
sage: b*c
```

```
2*3^3 + (2*a + 2)*3^4 + O(3^5)
sage: b*c >> 1
2*3^2 + (2*a + 2)*3^3 + O(3^4)
```

The fixed modulus case is like the capped absolute, except that individual elements don't track their precision.:

```
sage: R.<a> = Zq(9, 5, 'fixed-mod', print_mode='series'); b = 3*(1+2*a)^4; b
2*3 + (2*a + 2)*3^2 + (2*a + 1)*3^3 + O(3^5)
sage: c = R(3249); c
3^2 + 3^4 + O(3^5)
sage: b*c
2*3^3 + (2*a + 2)*3^4 + O(3^5)
sage: b*c >> 1
2*3^2 + (2*a + 2)*3^3 + O(3^5)
```

The lazy case will eventually support elements that can increase their precision upon request. It is not currently implemented.

MODULUS:

The modulus needs to define an unramified extension of $\mathbb{Z}_p$: when it is reduced to a polynomial over $\mathbb{F}_p$ it should be irreducible.

The modulus can be given in a number of forms.

1. A **polynomial**.

The base ring can be $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{Z}_p$, $\mathbb{F}_p$, or anything that can be converted to $\mathbb{Z}_p$.:

```
sage: P.<x> = ZZ[]
sage: R.<a> = Zq(27, modulus = x^3 + 2*x + 1); R.modulus()
(1 + O(3^20))*x^3 + (2 + O(3^20))*x + (1 + O(3^20))
sage: P.<x> = QQ[]
sage: S.<a> = Zq(27, modulus = x^3 + 2/7*x + 1)
sage: P.<x> = Zp(3)[]
sage: T.<a> = Zq(27, modulus = x^3 + 2*x + 1)
sage: P.<x> = Qp(3)[]
sage: U.<a> = Zq(27, modulus = x^3 + 2*x + 1)
sage: P.<x> = GF(3)[]
sage: V.<a> = Zq(27, modulus = x^3 + 2*x + 1)
```

Which form the modulus is given in has no effect on the unramified extension produced:

```
sage: R == S, R == T, T == U, U == V
(False, True, True, False)
```

unless the modulus is different, or the precision of the modulus differs. In the case of `V`, the modulus is only given to precision `1`, so the resulting field has a precision cap of `1`.:

```
sage: V.precision_cap()
1
sage: U.precision_cap()
20
sage: P.<x> = Zp(3)[]
sage: modulus = x^3 + (2 + O(3^7))*x + (1 + O(3^10))
sage: modulus
(1 + O(3^20))*x^3 + (2 + O(3^7))*x + (1 + O(3^10))
sage: W.<a> = Zq(27, modulus = modulus); W.precision_cap()
7
```

1. The modulus can also be given as a **symbolic expression**.:

```
sage: x = var('x')
sage: X.<a> = Zq(27, modulus = x^3 + 2*x + 1); X.modulus()
(1 + O(3^20))*x^3 + (2 + O(3^20))*x + (1 + O(3^20))
sage: X == R
True
```

By default, the polynomial chosen is the standard lift of the generator chosen for $\mathbb{F}_q$.:

```
sage: GF(125, 'a').modulus()
x^3 + 3*x + 3
sage: Y.<a> = Zq(125); Y.modulus()
(1 + O(5^20))*x^3 + (3 + O(5^20))*x + (3 + O(5^20))
```

However, you can choose another polynomial if desired (as long as the reduction to $\mathbb{F}_p[x]$ is irreducible).:

```
sage: P.<x> = ZZ[]
sage: Z.<a> = Zq(125, modulus = x^3 + 3*x^2 + x + 1); Z.modulus()
(1 + O(5^20))*x^3 + (3 + O(5^20))*x^2 + (1 + O(5^20))*x + (1 + O(5^20))
sage: Y == Z
False
```

PRINTING:

There are many different ways to print p-adic elements. The way elements of a given field print is controlled by options passed in at the creation of the field. There are four basic printing modes (`'series'`, `'val-unit'`, `'terse'` and `'bars'`; `'digits'` is not available), as well as various options that either hide some information in the print representation or sometimes make print representations more compact. Note that the printing options affect whether different p-adic fields are considered equal.

1. **series**: elements are displayed as series in p.:

```
sage: R.<a> = Zq(9, 20, 'capped-rel', print_mode='series'); (1+2*a)^4
2 + (2*a + 2)*3 + (2*a + 1)*3^2 + O(3^20)
sage: -3*(1+2*a)^4
3 + a*3^2 + 3^3 + (2*a + 2)*3^4 + (2*a + 2)*3^5 + (2*a + 2)*3^6 + (2*a + 2)*3^7 + (2*a + 2)*
sage: b = ~(3*a+18); b
(a + 2)*3^-1 + 1 + 2*3 + (a + 1)*3^2 + 3^3 + 2*3^4 + (a + 1)*3^5 + 3^6 + 2*3^7 + (a + 1)*3^8
sage: b.parent() is R.fraction_field()
True
```

print_pos controls whether negatives can be used in the coefficients of powers of p.:

```
sage: S.<b> = Zq(9, print_mode='series', print_pos=False); (1+2*b)^4
-1 - b*3 - 3^2 + (b + 1)*3^3 + O(3^20)
sage: -3*(1+2*b)^4
3 + b*3^2 + 3^3 + (-b - 1)*3^4 + O(3^20)
```

ram_name controls how the prime is printed.:

```
sage: T.<d> = Zq(9, print_mode='series', ram_name='p'); 3*(1+2*d)^4
2*p + (2*d + 2)*p^2 + (2*d + 1)*p^3 + O(p^20)
```

print_max_ram_terms limits the number of powers of p that appear.:

```
sage: U.<e> = Zq(9, print_mode='series', print_max_ram_terms=4); repr(-3*(1+2*e)^4)
'3 + e*3^2 + 3^3 + (2*e + 2)*3^4 + ... + O(3^20)'
```

print_max_unram_terms limits the number of terms that appear in a coefficient of a power of p.:

```
sage: V.<f> = Zq(128, prec = 8, print_mode='series'); repr((1+f)^9)
'(f^3 + 1) + (f^5 + f^4 + f^3 + f^2)*2 + (f^6 + f^5 + f^4 + f + 1)*2^2 + (f^5 + f^4 + f^2 + f +
sage: V.<f> = Zq(128, prec = 8, print_mode='series', print_max_unram_terms = 3); repr((1+f)^9)
'(f^3 + 1) + (f^5 + f^4 + ... + f^2)*2 + (f^6 + f^5 + ... + 1)*2^2 + (f^5 + f^4 + ... + 1)*2^3 +
sage: V.<f> = Zq(128, prec = 8, print_mode='series', print_max_unram_terms = 2); repr((1+f)^9)
'(f^3 + 1) + (f^5 + ... + f^2)*2 + (f^6 + ... + 1)*2^2 + (f^5 + ... + 1)*2^3 + (f^6 + ... + 1)*2
sage: V.<f> = Zq(128, prec = 8, print_mode='series', print_max_unram_terms = 1); repr((1+f)^9)
'(f^3 + ...) + (f^5 + ...)*2 + (f^6 + ...)*2^2 + (f^5 + ...)*2^3 + (f^6 + ...)*2^4 + (f^5 + ...)
sage: V.<f> = Zq(128, prec = 8, print_mode='series', print_max_unram_terms = 0); repr((1+f)^9 -
'(...)*2 + (...)*2^2 + (...)*2^3 + (...)*2^4 + (...)*2^5 + (...)*2^6 + (...)*2^7 + O(2^8)'
```

print_sep and *print_max_terse_terms* have no effect.

Note that print options affect equality:

```
sage: R == S, R == T, R == U, R == V, S == T, S == U, S == V, T == U, T == V, U == V
(False, False, False, False, False, False, False, False, False, False)
```

1. **val-unit**: elements are displayed as $p^k u$:

```
sage: R.<a> = Zq(9, 7, print_mode='val-unit'); b = (1+3*a)^9 - 1; b
3^3 * (15 + 64*a) + O(3^7)
sage: ~b
3^-3 * (41 + a) + O(3)
```

print_pos controls whether to use a balanced representation or not.:

```
sage: S.<a> = Zq(9, 7, print_mode='val-unit', print_pos=False); b = (1+3*a)^9 - 1; b
3^3 * (15 - 17*a) + O(3^7)
sage: ~b
3^-3 * (-40 + a) + O(3)
```

ram_name affects how the prime is printed.:

```
sage: A.<x> = Zp(next_prime(10^6), print_mode='val-unit')[]
sage: T.<a> = Zq(next_prime(10^6)^3, 4, print_mode='val-unit', ram_name='p', modulus=x^3+385831*
p^2 * (90732455187 + 713749771767*a + 579958835561*a^2) + O(p^4)
sage: b * (a^2 + a - 4)^-4
p^2 * 1 + O(p^4)
```

print_max_terse_terms controls how many terms of the polynomial appear in the unit part.:

```
sage: U.<a> = Zq(17^4, 6, print_mode='val-unit', print_max_terse_terms=3); b = (17*(a^3-a+14)^6)
17 * (772941 + 717522*a + 870707*a^2 + ...) + O(17^6)
```

print_sep, *print_max_ram_terms* and *print_max_unram_terms* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

1. **terse**: elements are displayed as a polynomial of degree less than the degree of the extension.:

```
sage: R.<a> = Zq(125, print_mode='terse')
sage: (a+5)^177
68210977979428 + 90313850704069*a + 73948093055069*a^2 + O(5^20)
sage: (a/5+1)^177
10990518995053/5^177 + 14019905391569/5^177*a + 16727634070694/5^177*a^2 + O(5^-158)
```

Note that in this last computation, you get one fewer p-adic digit than one might expect. This is because `R` is capped absolute, and thus 5 is cast in with relative precision 19.

As of version 3.3, if coefficients of the polynomial are non-integral, they are always printed with an explicit power of p in the denominator.:

```
sage: 5*a + a^2/25
5*a + 1/5^2*a^2 + O(5^16)
```

print_pos controls whether to use a balanced representation or not.:

```
sage: (a-5)^6
22864 + 95367431627998*a + 8349*a^2 + O(5^20)
sage: S.<a> = Zq(125, print_mode='terse', print_pos=False); b = (a-5)^6; b
22864 - 12627*a + 8349*a^2 + O(5^20)
sage: (a - 1/5)^6
-20624/5^6 + 18369/5^5*a + 1353/5^3*a^2 + O(5^14)
```

ram_name affects how the prime is printed.:

```
sage: T.<a> = Zq(125, print_mode='terse', ram_name='p'); (a - 1/5)^6
95367431620001/p^6 + 18369/p^5*a + 1353/p^3*a^2 + O(p^14)
```

print_max_terse_terms controls how many terms of the polynomial are shown.:

```
sage: U.<a> = Zq(625, print_mode='terse', print_max_terse_terms=2); (a-1/5)^6
106251/5^6 + 49994/5^5*a + ... + O(5^14)
```

print_sep, *print_max_ram_terms* and *print_max_unram_terms* have no effect.

Equality again depends on the printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

1. **digits**: This print mode is not available when the residue field is not prime. It might make sense to have a dictionary for small fields, but this isn't implemented.
2. **bars**: elements are displayed in a similar fashion to series, but more compactly.:

```
sage: R.<a> = Zq(125); (a+5)^6
(4*a^2 + 3*a + 4) + (3*a^2 + 2*a)*5 + (a^2 + a + 1)*5^2 + (3*a + 2)*5^3 + (3*a^2 + a + 3)*5^
sage: R.<a> = Zq(125, print_mode='bars', prec=8); repr((a+5)^6)
'...[2, 3, 2]|[3, 1, 3]|[2, 3]|[1, 1, 1]|[0, 2, 3]|[4, 3, 4]'
sage: repr((a-5)^6)
'...[0, 4]|[1, 4]|[2, 0, 2]|[1, 4, 3]|[2, 3, 1]|[4, 4, 3]|[2, 4, 4]|[4, 3, 4]'
```

Note that it's not possible to read of the precision from the representation in this mode.:

```
sage: b = a + 3; repr(b)
'...[3, 1]'
sage: c = a + R(3, 4); repr(c)
'...[3, 1]'
sage: b.precision_absolute()
8
sage: c.precision_absolute()
4
```

print_pos controls whether the digits can be negative.:

```
sage: S.<a> = Zq(125, print_mode='bars', print_pos=False); repr((a-5)^6)
'...[1, -1, 1]|[2, 1, -2]|[2, 0, -2]|[-2, -1, 2]|[0, 0, -1]|[-2]|[-1, -2, -1]'
sage: repr((a-1/5)^6)
'...[0, 1, 2]|[-1, 1, 1]|.|[-2, -1, -1]|[2, 2, 1]|[0, 0, -2]|[0, -1]|[0, -1]|[1]'
```

print_max_ram_terms controls the maximum number of "digits" shown. Note that this puts a cap on the relative precision, not the absolute precision.:

```
sage: T.<a> = Zq(125, print_mode='bars', print_max_ram_terms=3, print_pos=False); repr((a-5)^6)
'...[0, 0, -1]|[-2]|[-1, -2, -1]'
sage: repr(5*(a-5)^6+50)
'...[0, 0, -1]|[]|[-1, -2, -1]|[]'
```

However, if the element has negative valuation, digits are shown up to the decimal point.:

```
sage: repr((a-1/5)^6)
'...|.|[-2, -1, -1]|[2, 2, 1]|[0, 0, -2]|[0, -1]|[0, -1]|[1]'
```

print_sep controls the separating character (`'|'` by default).:

```
sage: U.<a> = Zq(625, print_mode='bars', print_sep=''); b = (a+5)^6; repr(b)
'...[0, 1][4, 0, 2][3, 2, 2, 3][4, 2, 2, 4][0, 3][1, 1, 3][3, 1, 4, 1]'
```

print_max_unram_terms controls how many terms are shown in each `'digit'`:

```
sage: with local_print_mode(U, {'max_unram_terms': 3}): repr(b)
'...[0, 1][4,..., 0, 2][3,..., 2, 3][4,..., 2, 4][0, 3][1,..., 1, 3][3,..., 4, 1]'
sage: with local_print_mode(U, {'max_unram_terms': 2}): repr(b)
'...[0, 1][4,..., 2][3,..., 3][4,..., 4][0, 3][1,..., 3][3,..., 1]'
sage: with local_print_mode(U, {'max_unram_terms': 1}): repr(b)
'...[..., 1][..., 2][..., 3][..., 4][..., 3][..., 3][..., 1]'
sage: with local_print_mode(U, {'max_unram_terms':0}): repr(b-75*a)
'...[...][...][...][...][][...][...]'
```

ram_name and *print_max_terse_terms* have no effect.

Equality depends on printing options:

```
sage: R == S, R == T, R == U, S == T, S == U, T == U
(False, False, False, False, False, False)
```

EXAMPLES

Unlike for `Zp`, you can't create `Zq(N)` when `N` is not a prime power.

However, you can use `check=False` to pass in a pair in order to not have to factor. If you do so, you need to use names explicitly rather than the `R.<a>` syntax.:

```
sage: p = next_prime(2^123)
sage: k = Zp(p)
sage: R.<x> = k[]
sage: K = Zq([(p, 5)], modulus=x^5+x+4, names='a', ram_name='p', print_pos=False, check=False)
sage: K.0^5
(-a - 4) + O(p^20)
```

In tests on sage.math, the creation of K as above took an average of 1.58ms, while:

```
sage: K = Zq(p^5, modulus=x^5+x+4, names='a', ram_name='p', print_pos=False, check=True)
```

took an average of 24.5ms. Of course, with smaller primes these savings disappear.

ZqCA(*q, prec=20, modulus=None, names=None, print_mode=None, halt=40, ram_name=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_ram_terms=None, print_max_unram_terms=None, print_max_terse_terms=None, check=True*)
A shorcut function to create capped absolute unramified p-adic rings.

See documentation for Zq for a description of the input paramters.

EXAMPLES:

```
sage: R.<a> = ZqCA(25, 40); R
Unramified Extension of 5-adic Ring with capped absolute precision 40 in a defined by (1 + O(5^4
```

ZqCR(*q, prec=20, modulus=None, names=None, print_mode=None, halt=40, ram_name=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_ram_terms=None, print_max_unram_terms=None, print_max_terse_terms=None, check=True*)
A shorcut function to create capped relative unramified p-adic rings.

Same functionality as Zq. See documentation for Zq for a description of the input paramters.

EXAMPLES:

```
sage: R.<a> = ZqCR(25, 40); R
Unramified Extension of 5-adic Ring with capped relative precision 40 in a defined by (1 + O(5^4
```

ZqFM(*q, prec=20, modulus=None, names=None, print_mode=None, halt=40, ram_name=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_ram_terms=None, print_max_unram_terms=None, print_max_terse_terms=None, check=True*)
A shorcut function to create fixed modulus unramified p-adic rings.

See documentation for Zq for a description of the input paramters.

EXAMPLES:

```
sage: R.<a> = ZqFM(25, 40); R
Unramified Extension of 5-adic Ring of fixed modulus 5^40 in a defined by (1 + O(5^40))*x^2 + (4
```

is_eisenstein(*poly*)
Returns True iff this monic polynomial is Eisenstein.

A polynomial is Eisenstein if it is monic, the constant term has valuation 1 and all other terms have positive valuation.

EXAMPLES:

```
sage: R = Zp(5)
sage: S.<x> = R[]
sage: from sage.rings.padics.factory import is_eisenstein
sage: f = x^4 - 75*x + 15
sage: is_eisenstein(f)
```

```
True
sage: g = x^4 + 75
sage: is_eisenstein(g)
False
sage: h = x^7 + 27*x -15
sage: is_eisenstein(h)
False
```

is_unramified(*poly*)

Returns true iff this monic polynomial is unramified.

A polynomial is unramified if its reduction modulo the maximal ideal is irreducible.

EXAMPLES:

```
sage: R = Zp(5)
sage: S.<x> = R[]
sage: from sage.rings.padics.factory import is_unramified
sage: f = x^4 + 14*x + 9
sage: is_unramified(f)
True
sage: g = x^6 + 17*x + 6
sage: is_unramified(g)
False
```

krasner_check(*poly, prec*)

Returns True iff poly determines a unique isomorphism class of extensions at precision prec.

Currently just returns True (thus allowing extensions that are not defined to high enough precision in order to specify them up to isomorphism). This will change in the future.

EXAMPLES:

```
sage: from sage.rings.padics.factory import krasner_check
sage: krasner_check(1,2) #this is a stupid example.
True
```

class **pAdicExtension_class**()

A class for creating extensions of p-adic rings and fields.

EXAMPLES:

```
sage: R = Zp(5,3)
sage: S.<x> = ZZ[]
sage: W.<w> = pAdicExtension(R, x^4-15)
sage: W
Eisenstein Extension of 5-adic Ring with capped relative precision 3 in w defined by (1 + O(5^3)
sage: W.precision_cap()
12
```

create_key_and_extra_args(*base, premodulus, prec=None, print_mode=None, halt=None, names=None, var_name=None, res_name=None, unram_name=None, ram_name=None, print_pos=None, print_sep=None, print_alphabet=None, print_max_ram_terms=None, print_max_unram_terms=None, print_max_terse_terms=None, check=True, unram=False*)

Creates a key from input parameters for pAdicExtension.

See the documentation for `Qq` for more information.

TESTS:

```
sage: R = Zp(5,3)
sage: S.<x> = ZZ[]
sage: pAdicExtension.create_key_and_extra_args(R, x^4-15,names='w')
(('e', 5-adic Ring with capped relative precision 3, x^4 - 15, (1 + O(5^3))*x^4 + (O(5^4))*x
```

create_object(*version, key, shift_seed*)

Creates an object using a given key.

See the documentation for pAdicExtension for more information.

TESTS:

```
sage: R = Zp(5,3)
sage: S.<x> = R[]
sage: pAdicExtension.create_object(version = (3,4,2), key = ('e', R, x^4 - 15, x^4 - 15, ('w
Eisenstein Extension of 5-adic Ring with capped relative precision 3 in w defined by (1 + O(
```

truncate_to_prec(*poly, absprec*)

Truncates the unused precision off of a polynomial.

EXAMPLES:

```
sage: R = Zp(5)
sage: S.<x> = R[]
sage: from sage.rings.padics.factory import truncate_to_prec
sage: f = x^4 + (3+O(5^6))*x^3 + O(5^4)
sage: truncate_to_prec(f, 5)
(1 + O(5^5))*x^4 + (3 + O(5^5))*x^3 + (O(5^5))*x^2 + (O(5^5))*x + (O(5^4))
```

27.3 Local Generic.

Superclass for p-adic and power series rings.

AUTHORS:

- David Roe

class LocalGeneric(*base, prec, names, element_class*)

defining_polynomial(*var='x'*)

Returns the defining polynomial of this local ring, i.e. just `x`.

INPUT:

- `self` – a local ring
- `var` – string (default: `'x'`) the name of the variable

OUTPUT:

```
- polynomial -- the defining polynomial of this ring as an extension over its ground ring
```

EXAMPLES:

```
sage: R = Zp(3, 3, 'fixed-mod'); R.defining_polynomial('foo')
(1 + O(3^3))*foo
```

degree()

Returns the degree of `self` over the ground ring, i.e. 1.

INPUT:

- `self` – a local ring

OUTPUT:

- integer – the degree of this ring, i.e., 1

EXAMPLES:

```
sage: R = Zp(3, 10, 'capped-rel'); R.degree()
1
```

e(*K=None*)

Returns the ramification index over the ground ring: 1 unless overridden.

INPUT:

- `self` – a local ring
- `K` – a subring of `self` (default `None`)

OUTPUT:

- integer – the ramification index of this ring: 1 unless overridden.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.e()
1
```

ext(**args, **kwds*)

Constructs an extension of self. See `extension` for more details.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.uniformiser()
t + O(t^21)
```

f(*K=None*)

Returns the inertia degree over the ground ring: 1 unless overridden.

INPUT:

- `self` – a local ring
- `K` – a subring (default `None`)

OUTPUT:

- integer – the inertia degree of this ring: 1 unless overridden.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.f()
1
```

ground_ring()

Returns `self`.

Will be overridden by extensions.

INPUT:

- `self` – a local ring

OUTPUT:

```
- the ground ring of ``self``, i.e., itself
```

EXAMPLES:

```
sage: R = Zp(3, 5, 'fixed-mod')
sage: S = Zp(3, 4, 'fixed-mod')
sage: R.ground_ring() is R
True
sage: S.ground_ring() is R
False
```

ground_ring_of_tower()

Returns `self`.

Will be overridden by extensions.

INPUT:

- `self` – a p-adic ring

OUTPUT:

- the ground ring of the tower for `self`, i.e., itself

EXAMPLES:

```
sage: R = Zp(5)
sage: R.ground_ring_of_tower()
5-adic Ring with capped relative precision 20
```

inertia_degree(*K=None*)

Returns the inertia degree over `K` (defaults to the ground ring): 1 unless overridden.

INPUT:

- `self` – a local ring
- `K` – a subring of `self` (default None)

OUTPUT:

- integer – the inertia degree of this ring: 1 unless overridden.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.inertia_degree()
1
```

inertia_subring()

Returns the inertia subring, i.e. `self`.

INPUT:

- `self` – a local ring

OUTPUT:

- the inertia subring of self, i.e., itself

EXAMPLES:

```
sage: R = Zp(5)
sage: R.inertia_subring()
5-adic Ring with capped relative precision 20
```

is_atomic_repr()

Return False, since we want p-adics to be printed with parentheses around them when they are coefficients, e.g., in a polynomial.

INPUT:

- `self` – a p-adic ring

OUTPUT:

- boolean – whether `self`'s representation is atomic, i.e., `False`

EXAMPLES:

```
sage: R = Zp(5, 5, 'fixed-mod'); R.is_atomic_repr()
False
```

is_capped_absolute()

Returns whether this p-adic ring bounds precision in a capped absolute fashion.

The absolute precision of an element is the power of p modulo which that element is defined. In a capped absolute ring, the absolute precision of elements are bounded by a constant depending on the ring.

EXAMPLES:

```
sage: R = ZpCA(5, 15)
sage: R.is_capped_absolute()
True
sage: R(5^7)
5^7 + O(5^15)
sage: S = Zp(5, 15)
sage: S.is_capped_absolute()
False
sage: S(5^7)
5^7 + O(5^22)
```

is_capped_relative()

Returns whether this p-adic ring bounds precision in a capped relative fashion.

The relative precision of an element is the power of p modulo which the unit part of that element is defined. In a capped relative ring, the relative precision of elements are bounded by a constant depending on the ring.

EXAMPLES:

```
sage: R = ZpCA(5, 15)
sage: R.is_capped_relative()
False
sage: R(5^7)
5^7 + O(5^15)
sage: S = Zp(5, 15)
sage: S.is_capped_relative()
True
sage: S(5^7)
5^7 + O(5^22)
```

is_exact()

Returns whether this p-adic ring is exact, i.e. False.

INPUT: self – a p-adic rinng

OUTPUT: boolean – whether self is exact, i.e. False.

EXAMPLES: #sage: R = Zp(5, 3, 'lazy'); R.is_exact() #False sage: R = Zp(5, 3, 'fixed-mod'); R.is_exact() False

is_finite()

Returns whether this ring is finite, i.e. `False`.

INPUT:

```
- ``self`` -- a `p`-adic ring
```

OUTPUT:

```
- boolean -- whether self is finite, i.e., ``False``
```

EXAMPLES:

```
sage: R = Zp(3, 10,'fixed-mod'); R.is_finite()
False
```

is_fixed_mod()

Returns whether this p-adic ring bounds precision in a fixed modulus fashion.

The absolute precision of an element is the power of p modulo which that element is defined. In a fixed modulus ring, the absolute precision of every element is defined to be the precision cap of the parent. This means that some operations, such as division by p, don't return a well defined answer.

EXAMPLES:

```
sage: R = ZpFM(5,15)
sage: R.is_fixed_mod()
True
sage: R(5^7,absprec=9)
5^7 + O(5^15)
sage: S = ZpCA(5, 15)
sage: S.is_fixed_mod()
False
sage: S(5^7,absprec=9)
5^7 + O(5^9)
```

is_lazy()

Returns whether this p-adic ring bounds precision in a lazy fashion.

In a lazy ring, elements have mechanisms for computing themselves to greater precision.

EXAMPLES:

```
sage: R = Zp(5)
sage: R.is_lazy()
False
```

maximal_unramified_subextension()

Returns the maximal unramified subextension.

INPUT:

- `self` – a local ring

OUTPUT:

- the maximal unramified subextension of `self`

EXAMPLES:

```
sage: R = Zp(5)
sage: R.maximal_unramified_subextension()
5-adic Ring with capped relative precision 20
```

precision_cap()

Returns the precision cap for `self`.

INPUT:

- `self` – a local ring

OUTPUT:

- integer – `self`'s precision cap

EXAMPLES:

```
sage: R = Zp(3, 10,'fixed-mod'); R.precision_cap()
10
sage: R = Zp(3, 10,'capped-rel'); R.precision_cap()
10
sage: R = Zp(3, 10,'capped-abs'); R.precision_cap()
10
```

NOTES:

```
This will have different meanings depending on the type of
local ring.  For fixed modulus rings, all elements are
considered modulo ``self.prime()^self.precision_cap()``.
For rings with an absolute cap (i.e. the class
``pAdicRingCappedAbsolute``), each element has a precision
that is tracked and is bounded above by
``self.precision_cap()``.  Rings with relative caps
(e.g. the class ``pAdicRingCappedRelative``) are the same
except that the precision is the precision of the unit
part of each element.  For lazy rings, this gives the
initial precision to which elements are computed.
```

ramification_index(*K=None*)

Returns the ramification index over the ground ring: 1 unless overridden.

INPUT:

- `self` – a local ring

OUTPUT:

- integer – the ramification index of this ring: 1 unless overridden.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.ramification_index()
1
```

residue_characteristic()

Returns the characteristic of `self`'s residue field.

INPUT:

- `self` – a p-adic ring.

OUTPUT:

- integer – the characteristic of the residue field.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.residue_characteristic()
3
```

residue_class_degree(*K=None*)

Returns the inertia degree over the ground ring: 1 unless overridden.

INPUT:

- `self` – a local ring
- `K` – a subring (default `None`)

OUTPUT:

- integer – the inertia degree of this ring: 1 unless overridden.

EXAMPLES:

```
sage: R = Zp(3, 5, 'capped-rel'); R.residue_class_degree()
1
```

uniformiser()

Returns a uniformiser for `self`, ie a generator for the unique maximal ideal.

EXAMPLES:

```
sage: R = Zp(5)
sage: R.uniformiser()
5 + O(5^21)
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.uniformiser()
t + O(t^21)
```

uniformiser_pow(*n*)

Returns the $n`thpowerofthe uniformiserof "self`$ (as an element of `self`).

EXAMPLES:

```
sage: R = Zp(5)
sage: R.uniformiser_pow(5)
5^5 + O(5^25)
```

27.4 p-Adic Generic.

A generic superclass for all p-adic parents.

AUTHORS:

- David Roe
- Genya Zaytman: documentation
- David Harvey: doctests

local_print_mode(*obj, print_options, pos=None, ram_name=None*)

Context manager for safely temporarily changing the print_mode of a p-adic ring/field.

EXAMPLES:

```
sage: R = Zp(5)
sage: R(45)
4*5 + 5^2 + O(5^21)
sage: with local_print_mode(R, 'val-unit'):
...       print R(45)
...
5 * 9 + O(5^21)
```

NOTES:

```
For more documentation see localvars in parent_gens.pyx
```

class pAdicGeneric(*base, p, prec, print_mode, names, element_class*)

characteristic()

Returns the characteristic of self, which is always 0.

INPUT:

```
self -- a p-adic parent
```

OUTPUT:

integer -- self's characteristic, i.e., 0

EXAMPLES:

```
sage: R = Zp(3, 10,'fixed-mod'); R.characteristic()
0
```

extension(*modulus, prec=None, names=None, print_mode=None, halt=None, **kwds*)

Create an extension of this p-adic ring.

EXAMPLES:

```
sage: k = Qp(5)
sage: R.<x> = k[]
sage: l.<w> = k.extension(x^2-5); l
Eisenstein Extension of 5-adic Field with capped relative precision 20 in w defined by (1 +

sage: F = list(Qp(19)['x'](cyclotomic_polynomial(5)).factor())[0][0]
sage: L = Qp(19).extension(F, names='a')
sage: L
Unramified Extension of 19-adic Field with capped relative precision 20 in a defined by (1 +
```

gens()

Returns a list of generators.

EXAMPLES:

```
sage: R = Zp(5); R.gens()
[5 + O(5^21)]
sage: Zq(25,names='a').gens()
[a + O(5^20)]
sage: S.<x> = ZZ[]; f = x^5 + 25*x -5; W.<w> = R.ext(f); W.gens()
[w + O(w^101)]
```

ngens()

Returns the number of generators of self.

We conventionally define this as 1: for base rings, we take a uniformizer as the generator; for extension rings, we take a root of the minimal polynomial defining the extension.

EXAMPLES:

```
sage: Zp(5).ngens()
1
sage: Zq(25,names='a').ngens()
1
```

prime()

Returns the prime, ie the characteristic of the residue field.

INPUT:

self -- a p-adic parent

OUTPUT:

integer -- the characteristic of the residue field

EXAMPLES:

```
sage: R = Zp(3,5,'fixed-mod')
sage: R.prime()
3
```

print_mode()

Returns the current print mode as a string.

INPUT:

```
self -- a p-adic field
```

OUTPUT:

```
string -- self's print mode
```

EXAMPLES:

```
sage: R = Qp(7,5, 'capped-rel')
sage: R.print_mode()
'series'
```

residue_characteristic()

Returns the prime, i.e., the characteristic of the residue field.

INPUT:

```
self -- a p-adic ring
```

OUTPUT:

```
integer -- the characteristic of the residue field
```

EXAMPLES:

```
sage: R = Zp(3,5,'fixed-mod')
sage: R.residue_characteristic()
3
```

residue_class_field()

Returns the residue class field.

INPUT:

```
self -- a p-adic ring
```

OUTPUT:

```
the residue field
```

EXAMPLES:

```
sage: R = Zp(3,5,'fixed-mod')
sage: k = R.residue_class_field()
sage: k
Finite Field of size 3
```

residue_field()

Returns the residue class field.

INPUT:

```
self -- a p-adic ring
```

OUTPUT:

```
the residue field
```

EXAMPLES:

```
sage: R = Zp(3,5,'fixed-mod')
sage: k = R.residue_field()
sage: k
Finite Field of size 3
```

residue_system()

Returns a list of elements representing all the residue classes.

INPUT:

self -- a p-adic ring

OUTPUT:

list of elementss -- a list of elements representing all the residue classes

EXAMPLES:

```
sage: R = Zp(3, 5,'fixed-mod')
sage: R.residue_system()
[O(3^5), 1 + O(3^5), 2 + O(3^5)]
```

teichmuller(*x*, *prec=None*)

Returns the teichmuller representative of x.

INPUT:

- self -- a p-adic ring
- x -- something that can be cast into self

OUTPUT:

- element -- the teichmuller lift of x

EXAMPLES:

```
sage: R = Zp(5, 10, 'capped-rel', 'series')
sage: R.teichmuller(2)
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
sage: R = Qp(5, 10,'capped-rel','series')
sage: R.teichmuller(2)
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
sage: R = Zp(5, 10, 'capped-abs', 'series')
sage: R.teichmuller(2)
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
sage: R = Zp(5, 10, 'fixed-mod', 'series')
sage: R.teichmuller(2)
2 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: y = W.teichmuller(3); y
3 + 3*w^5 + w^7 + 2*w^9 + 2*w^10 + 4*w^11 + w^12 + 2*w^13 + 3*w^15 + 2*w^16 + 3*w^17 + w^18
sage: y^5 == y
True
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: b = A.teichmuller(1 + 2*a - a^2); b
(4*a^2 + 2*a + 1) + 2*a*5 + (3*a^2 + 1)*5^2 + (a + 4)*5^3 + (a^2 + a + 1)*5^4 + O(5^5)
sage: b^125 == b
True
```

AUTHORS:

- Initial version: David Roe
- Quadratic time version: Kiran Kedlaya <kedlaya@math.mit.edu> (3/27/07)

teichmuller_system()

Returns a set of teichmuller representatives for the invertible elements of Z / pZ.

INPUT:

- self – a p-adic ring

OUTPUT:

•list of elements – a list of teichmuller representatives for the invertible elements of Z / pZ

EXAMPLES:

```
sage: R = Zp(3, 5,'fixed-mod', 'terse')
sage: R.teichmuller_system()
[1 + O(3^5), 242 + O(3^5)]
```

NOTES:

Should this return 0 as well?

uniformizer_pow(*n*)

Returns p^n, as an element of self.

If n is infinity, returns 0.

EXAMPLES:

```
sage: R = Zp(3, 5, 'fixed-mod')
sage: R.uniformizer_pow(3)
3^3 + O(3^5)
sage: R.uniformizer_pow(infinity)
O(3^5)
```

27.5 p-Adic Generic Nodes.

This file contains a bunch of intermediate classes for the p-adic parents, allowing a function to be implemented at the right level of generality.

AUTHORS:

- David Roe

class CappedAbsoluteGeneric(*base, prec, names, element_class*)

is_capped_absolute()

Returns whether this p-adic ring bounds precision in a capped absolute fashion.

The absolute precision of an element is the power of p modulo which that element is defined. In a capped absolute ring, the absolute precision of elements are bounded by a constant depending on the ring.

EXAMPLES:

```
sage: R = ZpCA(5, 15)
sage: R.is_capped_absolute()
True
sage: R(5^7)
5^7 + O(5^15)
sage: S = Zp(5, 15)
sage: S.is_capped_absolute()
False
sage: S(5^7)
5^7 + O(5^22)
```

class CappedRelativeFieldGeneric(*base, prec, names, element_class*)

class CappedRelativeGeneric(*base, prec, names, element_class*)

is_capped_relative()

Returns whether this p-adic ring bounds precision in a capped relative fashion.

The relative precision of an element is the power of p modulo which the unit part of that element is defined. In a capped relative ring, the relative precision of elements are bounded by a constant depending on the ring.

EXAMPLES:

```
sage: R = ZpCA(5, 15)
sage: R.is_capped_relative()
False
sage: R(5^7)
5^7 + O(5^15)
sage: S = Zp(5, 15)
sage: S.is_capped_relative()
True
sage: S(5^7)
5^7 + O(5^22)
```

class CappedRelativeRingGeneric(*base, prec, names, element_class*)

class FixedModGeneric(*base, prec, names, element_class*)

is_fixed_mod()

Returns whether this p-adic ring bounds precision in a fixed modulus fashion.

The absolute precision of an element is the power of p modulo which that element is defined. In a fixed modulus ring, the absolute precision of every element is defined to be the precision cap of the parent. This means that some operations, such as division by p, don't return a well defined answer.

EXAMPLES:

```
sage: R = ZpFM(5,15)
sage: R.is_fixed_mod()
True
sage: R(5^7,absprec=9)
5^7 + O(5^15)
sage: S = ZpCA(5, 15)
sage: S.is_fixed_mod()
False
sage: S(5^7,absprec=9)
5^7 + O(5^9)
```

is_pAdicField(*R*)

Returns `True` if and only if `R` is a p-adic field.

EXAMPLES:

```
sage: is_pAdicField(Zp(17))
False
sage: is_pAdicField(Qp(17))
True
```

is_pAdicRing(*R*)

Returns `True` if and only if `R` is a p-adic ring (not a field).

EXAMPLES:

```
sage: is_pAdicRing(Zp(5))
True
sage: is_pAdicRing(RR)
False
```

class **pAdicCappedAbsoluteRingGeneric**(*base, p, prec, print_mode, names, element_class*)

class **pAdicCappedRelativeFieldGeneric**(*base, p, prec, print_mode, names, element_class*)

class **pAdicCappedRelativeRingGeneric**(*base, p, prec, print_mode, names, element_class*)

class **pAdicFieldBaseGeneric**(*p, prec, print_mode, names, element_class*)

composite(*subfield1, subfield2*)

Returns the composite of two subfields of self, i.e., the largest subfield containing both

INPUT:

- `self` – a p-adic field
- `subfield1` – a subfield
- `subfield2` – a subfield

OUTPUT:

- the composite of subfield1 and subfield2

EXAMPLES:

```
sage: K = Qp(17); K.composite(K, K) is K
True
```

construction()

Returns the functorial construction of `self`, namely, completion of the rational numbers with respect a given prime.

Also preserves other information that makes this field unique (e.g. precision, rounding, print mode).

EXAMPLE:

```
sage: K = Qp(17, 8, print_mode='val-unit', print_sep='&')
sage: c, L = K.construction(); L
Rational Field
sage: c(L)
17-adic Field with capped relative precision 8
sage: K == c(L)
True
```

subfield(*list*)

Returns the subfield generated by the elements in list

INPUT:

- `self` – a p-adic field
- `list` – a list of elements of `self`

OUTPUT:

- the subfield of `self` generated by the elements of list

EXAMPLES:

```
sage: K = Qp(17); K.subfield([K(17), K(1827)]) is K
True
```

subfields_of_degree(*n*)

Returns the number of subfields of self of degree n

INPUT:

- `self` – a p-adic field
- `n` – an integer

OUTPUT:

- integer – the number of subfields of degree `n` over self.base_ring()

EXAMPLES:

```
sage: K = Qp(17)
sage: K.subfields_of_degree(1)
1
```

class pAdicFieldGeneric (*base, p, prec, print_mode, names, element_class*)

class pAdicFixedModRingGeneric (*base, p, prec, print_mode, names, element_class*)

class pAdicRingBaseGeneric (*p, prec, print_mode, names, element_class*)

construction()

Returns the functorial construction of self, namely, completion of the rational numbers with respect a given prime.

Also preserves other information that makes this field unique (e.g. precision, rounding, print mode).

EXAMPLE:

```
sage: K = Zp(17, 8, print_mode='val-unit', print_sep='&')
sage: c, L = K.construction(); L
Integer Ring
sage: c(L)
17-adic Ring with capped relative precision 8
sage: K == c(L)
True
```

random_element (*algorithm='default'*)

Returns a random element of self, optionally using the algorithm argument to decide how it generates the element. Algorithms currently implemented:

- default: Choose a_i, $i >= 0$, randomly between 0 and $p-1$ until a nonzero choice is made. Then continue choosing a_i randomly between 0 and $p-1$ until we reach precision_cap, and return $\sum a_i p^i$.

EXAMPLES:

```
sage: Zp(5,6).random_element()
3 + 3*5 + 2*5^2 + 3*5^3 + 2*5^4 + 5^5 + O(5^6)
sage: ZpCA(5,6).random_element()
4*5^2 + 5^3 + O(5^6)
sage: ZpFM(5,6).random_element()
2 + 4*5^2 + 2*5^4 + 5^5 + O(5^6)
```

class pAdicRingGeneric (*base, p, prec, print_mode, names, element_class*)

is_field()

Returns whether this ring is actually a field, ie `False`.

EXAMPLES:

```
sage: Zp(5).is_field()
False
```

krull_dimension()

Returns the Krull dimension of self, i.e. 1

INPUT:

- self – a p-adic ring

OUTPUT:

- the Krull dimension of self. Since self is a p-adic ring, this is 1.

EXAMPLES:

```
sage: Zp(5).krull_dimension()
1
```

27.6 p-Adic Base Generic.

A superclass for implementations of $\mathbb{Z}_p$ and $\mathbb{Q}_p$.

AUTHORS:

- David Roe

class pAdicBaseGeneric(*p, prec, print_mode, names, element_class*)

absolute_discriminant()
Returns the absolute discriminant of this p-adic ring
EXAMPLES:

```
sage: Zp(5).absolute_discriminant()
1
```

discriminant(*K=None*)
Returns the discriminant of this p-adic ring over `K`
INPUT:

- `self` – a p-adic ring
- `K` – a sub-ring of `self` or `None` (default: `None`)

OUTPUT:

- integer – the discriminant of this ring over `K` (or the absolute discriminant if `K` is `None`)

EXAMPLES:

```
sage: Zp(5).discriminant()
1
```

fraction_field(*print_mode=None*)
Returns the fraction field of `self`.
INPUT:

- `print_mode` - a dictionary containing print options. Defaults to the same options as this ring.

OUTPUT:

- the fraction field of `self`.

EXAMPLES:

```
sage: R = Zp(5, print_mode='digits')
sage: K = R.fraction_field(); repr(K(1/3))[3:]
'31313131313131313132'
sage: L = R.fraction_field({'max_ram_terms':4}); repr(L(1/3))[3:]
'3132'
```

gen(*n=0*)
Returns the `n`th generator of this extension. For base rings/fields, we consider the generator to be the prime.
EXAMPLES:

```
sage: R = Zp(5); R.gen()
5 + O(5^21)
```

has_pth_root()

Returns whether or not $\mathbb{Z}_p$ has a primitive p^{th} root of unity.

EXAMPLES:

```
sage: Zp(2).has_pth_root()
True
sage: Zp(17).has_pth_root()
False
```

has_root_of_unity(*n*)

Returns whether or not $\mathbb{Z}_p$ has a primitive n^{th} root of unity.

INPUT:

- `self` – a p-adic ring
- n – an integer

OUTPUT:

- `boolean` – whether `self` has primitive n^{th} root of unity

EXAMPLES:

```
sage: R=Zp(37)
sage: R.has_root_of_unity(12)
True
sage: R.has_root_of_unity(11)
False
```

integer_ring(*print_mode=None*)

Returns the integer ring of `self`, possibly with `print_mode` changed.

INPUT:

- `print_mode` - a dictionary containing print options. Defaults to the same options as this ring.

OUTPUT:

- The ring of integral elements in `self`.

EXAMPLES:

```
sage: K = Qp(5, print_mode='digits')
sage: R = K.integer_ring(); repr(R(1/3))[3:]
'31313131313131313132'
sage: S = K.integer_ring({'max_ram_terms':4}); repr(S(1/3))[3:]
'3132'
```

is_abelian()

Returns whether the Galois group is abelian, i.e. `True`. #should this be automorphism group?

EXAMPLES:

```
sage: R = Zp(3, 10,'fixed-mod'); R.is_abelian()
True
```

is_isomorphic(*ring*)

Returns whether `self` and `ring` are isomorphic, i.e. whether `ring` is an implementation of $\mathbb{Z}_p$ for the same prime as `self`.

INPUT:

- `self` – a p-adic ring
- `ring` – a ring

OUTPUT:

•boolean – whether ring is an implementation of mathbb{Z}_p‘ for the same prime as self.

EXAMPLES:

```
sage: R = Zp(5, 15, print_mode='digits'); S = Zp(5, 44, print_max_terms=4); R.is_isomorphic(
True
```

is_normal()

Returns whether or not this is a normal extension, i.e. True.

EXAMPLES:

```
sage: R = Zp(3, 10,'fixed-mod'); R.is_normal()
True
```

plot(*max_points=2500, **args*)

Creates a visualization of this p-adic ring as a fractal similar as a generalization of the the Sierpi'nski triangle. The resulting image attempts to capture the algebraic and topological characteristics of $\mathbb{Z}_p$.

INPUT:

•point_count – the maximum number or points to plot, which controls the depth of recursion (default 2500)

•**args – color, size, etc. that are passed to the underlying point graphics objects

REFERENCES:

•Cuoco, A. ''Visualizing the p-adic Integers'', The American Mathematical Monthly, Vol. 98, No. 4 (Apr., 1991), pp. 355-364

EXAMPLES:

```
sage: Zp(3).plot()
sage: Zp(5).plot(max_points=625)
sage: Zp(23).plot(rgbcolor=(1,0,0))
```

uniformizer()

Returns a uniformizer for this ring.

EXAMPLES:

```
sage: R = Zp(3,5,'fixed-mod', 'series')
sage: R.uniformizer()
3 + O(3^5)
```

uniformizer_pow(*n*)

Returns the nth power of the uniformizer of self (as an element of self).

EXAMPLES:

```
sage: R = Zp(5)
sage: R.uniformizer_pow(5)
5^5 + O(5^25)
sage: R.uniformizer_pow(infinity)
0
```

zeta(*n=None*)

Returns a generator of the group of roots of unity.

INPUT:

•self – a p-adic ring

•n – an integer or None (default: None)

OUTPUT:

•element – a generator of the n^{th} roots of unity, or a generator of the full group of roots of unity if n is None

EXAMPLES:

```
sage: R = Zp(37,5)
sage: R.zeta(12)
8 + 24*37 + 37^2 + 29*37^3 + 23*37^4 + O(37^5)
```

zeta_order()
Returns the order of the group of roots of unity.
EXAMPLES:

```
sage: R = Zp(37); R.zeta_order()
36
sage: Zp(2).zeta_order()
2
```

27.7 p-Adic Extension Generic.

A common superclass for all extensions of Qp and Zp.

AUTHORS:

- David Roe

class pAdicExtensionGeneric(*poly, prec, print_mode, names, element_class*)

defining_polynomial()
Returns the polynomial defining this extension.
EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: W.defining_polynomial()
(1 + O(5^5))*x^5 + (O(5^6))*x^4 + (3*5^2 + O(5^6))*x^3 + (2*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^
```

degree()
Returns the degree of this extension.
EXAMPLES:

```
sage: R.<a> = Zq(125); R.degree()
3
sage: R = Zp(5); S.<x> = ZZ[]; f = x^5 - 25*x^3 + 5; W.<w> = R.ext(f)
sage: W.degree()
5
```

fraction_field(*print_mode=None*)
Returns the fraction field of this extension, which is just the extension of base.fraction_field() determined by the same polynomial.
INPUT:

•print_mode – a dictionary containing print options. Defaults to the same options as this ring.

OUTPUT:

•the fraction field of self.

EXAMPLES:

```
sage: U.<a> = Zq(17^4, 6, print_mode='val-unit', print_max_terse_terms=3)
sage: U.fraction_field()
Unramified Extension of 17-adic Field with capped relative precision 6 in a defined by (1 +
sage: U.fraction_field({"pos":False}) == U.fraction_field()
False
```

ground_ring()

Returns the ring of which this ring is an extension.

EXAMPLE:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: W.ground_ring()
5-adic Ring with capped relative precision 5
```

ground_ring_of_tower()

Returns the p-adic base ring of which this is ultimately an extension.

Currently this function is identical to ground_ring(), since relative extensions have not yet been implemented.

EXAMPLES:

```
sage: Qq(27,30,names='a').ground_ring_of_tower()
3-adic Field with capped relative precision 30
```

integer_ring(*print_mode=None*)

Returns the ring of integers of self, which is just the extension of base.integer_ring() determined by the same polynomial.

INPUT:

```
- print_mode -- a dictionary containing print options.
  Defaults to the same options as this ring.
```

OUTPUT:

```
- the ring of elements of self with nonnegative valuation.
```

EXAMPLES:

```
sage: U.<a> = Qq(17^4, 6, print_mode='val-unit', print_max_terse_terms=3)
sage: U.integer_ring()
Unramified Extension of 17-adic Ring with capped relative precision 6 in a defined by (1 + O
sage: U.fraction_field({"pos":False}) == U.fraction_field()
False
```

modulus()

Returns the polynomial defining this extension.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: W.modulus()
(1 + O(5^5))*x^5 + (O(5^6))*x^4 + (3*5^2 + O(5^6))*x^3 + (2*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^
```

polynomial_ring()

Returns the polynomial ring of which this is a quotient.

EXAMPLES:

```
sage: Qq(27,30,names='a').polynomial_ring()
Univariate Polynomial Ring in x over 3-adic Field with capped relative precision 30
```

random_element()

Returns a random element of self.

This is done by picking a random element of the ground ring self.degree() times, then treating those elements as coefficients of a polynomial in self.gen().

EXAMPLES:

```
sage: R.<a> = Zq(125, 5); R.random_element()
3*a + (2*a + 1)*5 + (4*a^2 + 3*a + 4)*5^2 + (a^2 + 2*a)*5^3 + (a + 2)*5^4 + O(5^5)
sage: R = Zp(5,3); S.<x> = ZZ[]; f = x^5 + 25*x^2 - 5; W.<w> = R.ext(f)
sage: W.random_element()
3 + 4*w + 3*w^2 + w^3 + 4*w^4 + w^5 + w^6 + 3*w^7 + w^8 + 2*w^10 + 4*w^11 + w^12 + 2*w^13 +
```

27.8 Eisenstein Extension Generic.

This file implements the shared functionality for Eisenstein extentions.

AUTHORS:

- David Roe

class EisensteinExtensionGeneric(*poly, prec, print_mode, names, element_class*)

gen(*n=0*)

Returns a generator for self as an extension of its ground ring.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.gen()
t + O(t^21)
```

inertia_degree(*K=None*)

Returns the inertia degree of self over K, or the ground ring if K is None.

The inertia degree is the degree of the extension of residue fields induced by this extensions. Since Eisenstein extensions are totally ramified, this will be 1 for K=None.

INPUTS:

•self – an eisenstein extension

•K – a subring of self (default None -> self.ground_ring())

OUTPUTS:

•The degree of the induced extensions of residue fields.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.inertia_degree()
1
```

inertia_subring()

Returns the inertia subring.

Since an eisenstein extension is totally ramified, this is just the ground field.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.inertia_subring()
7-adic Ring with capped relative precision 10
```

ramification_index(*K=None*)

Returns the ramification index of self over K, or over the ground ring if K is None.

The ramification index is the index of the image of the valuation map on K in the image of the valuation map on self (both normalized so that the valuation of p is 1).

INPUTS:

- self – an eisenstein extension
- K – a subring of self (default None -> self.ground_ring())

OUTPUTS:

- The ramification index of the extension self/K

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.ramification_index()
2
```

residue_class_field()

Returns the residue class field.

INPUT:

- self – a p-adic ring

OUTPUT:

- the residue field

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.residue_class_field()
Finite Field of size 7
```

uniformizer()

Returns the uniformizer of self, ie a generator for the unique maximal ideal.

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.uniformizer()
t + O(t^21)
```

uniformizer_pow(*n*)

Returns the nth power of the uniformizer of self (as an element of self).

EXAMPLES:

```
sage: A = Zp(7,10)
sage: S.<x> = A[]
sage: B.<t> = A.ext(x^2+7)
sage: B.uniformizer_pow(5)
t^5 + O(t^25)
```

27.9 Unramified Extension Generic.

This file implements the shared functionality for unramified extentions.

AUTHORS:

- David Roe

class **UnramifiedExtensionGeneric**(*poly, prec, print_mode, names, element_class*)

An unramified extension of Qp or Zp.

discriminant(*K=None*)

Returns the discriminant of self over the subring K.

INPUTS:

```
- K -- a subring/subfield (defaults to the base ring).
```

EXAMPLES:

```
sage: R.<a> = Zq(125)
sage: R.discriminant()
...
NotImplementedError
```

gen(*n=0*)

Returns a generator for this unramified extension.

This is an element that satisfies the polynomial defining this extension. Such an element will reduce to a generator of the corresponding residue field extension.

EXAMPLES:

```
sage: R.<a> = Zq(125); R.gen()
a + O(5^20)
```

has_pth_root()

Returns whether or not Z_p has a primitive $p^{mbox{th}}$ root of unity.

Since adjoining a $p^{mbox{th}}$ root of unity yields a totally ramified extension, self will contain one if and only if the ground ring does.

INPUT:

```
- self -- a p-adic ring
```

OUTPUT:

```
- boolean -- whether self has primitive $p^{\mbox{th}}$
  root of unity.
```

EXAMPLES:

```
sage: R.<a> = Zq(1024); R.has_pth_root()
True
sage: R.<a> = Zq(17^5); R.has_pth_root()
False
```

has_root_of_unity(*n*)

Returns whether or not Z_p has a primitive $n^{\mbox{th}}$ root of unity.

INPUT:

```
- self -- a p-adic ring
- n -- an integer
```

OUTPUT:

```
- boolean -- whether self has primitive $n^{\mbox{th}}$
  root of unity
```

EXAMPLES:

```
sage: R.<a> = Zq(37^8)
sage: R.has_root_of_unity(144)
True
sage: R.has_root_of_unity(89)
True
sage: R.has_root_of_unity(11)
False
```

inertia_degree(*K=None*)

Returns the inertia degree of self over the subring K.

INPUTS:

```
- K -- a subring (or subfield) of self.  Defaults to the
  base.
```

EXAMPLES:

```
sage: R.<a> = Zq(125); R.inertia_degree()
3
```

is_galois(*K=None*)

Returns True if this extension is Galois.

Every unramified extension is Galois.

INPUTS:

```
- K -- a subring/subfield (defaults to the base ring).
```

EXAMPLES:

```
sage: R.<a> = Zq(125); R.is_galois()
True
```

ramification_index(*K=None*)

Returns the ramification index of self over the subring K.

INPUTS:

```
- K -- a subring (or subfield) of self.  Defaults to the
  base.
```

EXAMPLES:

```
sage: R.<a> = Zq(125); R.ramification_index()
1
```

residue_class_field()

Returns the residue class field.

EXAMPLES:

```
sage: R.<a> = Zq(125); R.residue_class_field()
Finite Field in a0 of size 5^3
```

uniformizer()

Returns a uniformizer for this extension.

Since this extension is unramified, a uniformizer for the ground ring will also be a uniformizer for this extension.

EXAMPLES:

```
sage: R.<a> = ZqCR(125)
sage: R.uniformizer()
5 + O(5^21)
```

uniformizer_pow(n)

Returns the nth power of the uniformizer of self (as an element of self).

EXAMPLES:

```
sage: R.<a> = ZqCR(125)
sage: R.uniformizer_pow(5)
5^5 + O(5^25)
```

27.10 p-Adic Base Leaves.

Implementations of $\mathbb{Z}_p$ and $\mathbb{Q}_p$

AUTHORS:

- David Roe
- Genya Zaytman: documentation
- David Harvey: doctests
- William Stein: doctest updates

EXAMPLES:

p-Adic rings and fields are examples of inexact structures, as the reals are. That means that elements cannot generally be stored exactly: to do so would take an infinite amount of storage. Instead, we store an approximation to the elements with varying precision.

There are two types of precision for a p-adic element. The first is relative precision, which gives the number of known p-adic digits:

```
sage: R = Qp(5, 20, 'capped-rel', 'series'); a = R(675); a
2*5^2 + 5^4 + O(5^22)
sage: a.precision_relative()
20
```

The second type of precision is absolute precision, which gives the power of p that this element is stored modulo:

```
sage: a.precision_absolute()
22
```

The number of times that p divides the element is called the valuation, and can be accessed with the functions `valuation()` and `ordp()`:

sage: a.valuation() 2

The following relationship holds:

`self.valuation() + self.precision_relative() == self.precision_absolute()`.

sage: a.valuation() + a.precision_relative() == a.precision_absolute() True

In the capped relative case, the relative precision of an element is restricted to be at most a certain value, specified at the creation of the field. Individual elements also store their own precision, so the effect of various arithmetic operations on precision is tracked. When you cast an exact element into a capped relative field, it truncates it to the precision cap of the field.:

```
sage: R = Qp(5, 5); a = R(4006); a
1 + 5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(17/3); b
4 + 2*5 + 3*5^2 + 5^3 + 3*5^4 + O(5^5)
sage: c = R(4025); c
5^2 + 2*5^3 + 5^4 + 5^5 + O(5^7)
sage: a + b
4*5 + 3*5^2 + 3*5^3 + 4*5^4 + O(5^5)
sage: a + b + c
4*5 + 4*5^2 + 5^4 + O(5^5)
```

```
sage: R = Zp(5, 5, 'capped-rel', 'series'); a = R(4006); a
1 + 5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(17/3); b
4 + 2*5 + 3*5^2 + 5^3 + 3*5^4 + O(5^5)
sage: c = R(4025); c
5^2 + 2*5^3 + 5^4 + 5^5 + O(5^7)
sage: a + b
4*5 + 3*5^2 + 3*5^3 + 4*5^4 + O(5^5)
sage: a + b + c
4*5 + 4*5^2 + 5^4 + O(5^5)
```

In the capped absolute type, instead of having a cap on the relative precision of an element there is instead a cap on the absolute precision. Elements still store their own precisions, and as with the capped relative case, exact elements are truncated when cast into the ring.:

```
sage: R = ZpCA(5, 5); a = R(4005); a
5 + 2*5^3 + 5^4 + O(5^5)
sage: b = R(4025); b
5^2 + 2*5^3 + 5^4 + O(5^5)
sage: a * b
5^3 + 2*5^4 + O(5^5)
sage: (a * b) // 5^3
1 + 2*5 + O(5^2)
sage: type((a * b) // 5^3)
<type 'sage.rings.padics.padic_capped_absolute_element.pAdicCappedAbsoluteElement'>
sage: (a * b) / 5^3
1 + 2*5 + O(5^2)
sage: type((a * b) / 5^3)
<type 'sage.rings.padics.padic_capped_relative_element.pAdicCappedRelativeElement'>
```

The fixed modulus type is the leanest of the p-adic rings: it is basically just a wrapper around $\mathbb{Z}/p^n\mathbb{Z}$ providing a unified interface with the rest of the p-adics. This is the type you should use if your primary interest is in speed (though it's not all that much faster than other p-adic types). It does not track precision of elements.:

```
sage: R = ZpFM(5, 5); a = R(4005); a
5 + 2*5^3 + 5^4 + O(5^5)
sage: a // 5
1 + 2*5^2 + 5^3 + O(5^5)
```

p-Adic rings and fields should be created using the creation functions `Zp` and `Qp` as above. This will ensure that there is only one instance of $\mathbb{Z}_p$ and $\mathbb{Q}_p$ of a given type, p, print mode and precision. It also saves typing very long class names.:

```
sage: Qp(17,10)
17-adic Field with capped relative precision 10
sage: R = Qp(7, prec = 20, print_mode = 'val-unit'); S = Qp(7, prec = 20, print_mode = 'val-unit'); R
True
sage: Qp(2)
2-adic Field with capped relative precision 20
```

Once one has a p-Adic ring or field, one can cast elements into it in the standard way. Integers, ints, longs, Rationals, other p-Adic types, pari p-adics and elements of $\mathbb{Z}/p^n\mathbb{Z}$ can all be cast into a p-Adic field.:

```
sage: R = Qp(5, 5, 'capped-rel','series'); a = R(16); a
1 + 3*5 + O(5^5)
sage: b = R(23/15); b
5^-1 + 3 + 3*5 + 5^2 + 3*5^3 + O(5^4)
sage: S = Zp(5, 5, 'fixed-mod','val-unit'); c = S(Mod(75,125)); c
5^2 * 3 + O(5^5)
sage: R(c)
3*5^2 + O(5^5)
```

In the previous example, since fixed-mod elements don't keep track of their precision, we assume that it has the full precision of the ring. This is why you have to cast manually here.

While you can cast explicitly as above, the chains of automatic coercion are more restricted. As always in SAGE, the following arrows are transitive and the diagram is commutative.:

```
int -> long -> Integer -> Zp capped-rel -> Zp capped_abs -> IntegerMod
Integer -> Zp fixed-mod -> IntegerMod
Integer -> Zp capped-abs -> Qp capped-rel
```

In addition, there are arrows within each type. For capped relative and capped absolute rings and fields, these arrows go from lower precision cap to higher precision cap. This works since elements track their own precision: choosing the parent with higher precision cap means that precision is less likely to be truncated unnecessarily. For fixed modulus parents, the arrow goes from higher precision cap to lower. The fact that elements do not track precision necessitates this choice in order to not produce incorrect results.

TESTS:

```
sage: R = Qp(5, 15, print_mode='bars', print_sep='&'); S = loads(dumps(R))
sage: R == S
True
sage: repr(S(2777))[3:]
'4&2&1&0&2'

sage: R = Zp(5, 15, print_mode='bars', print_sep='&'); S = loads(dumps(R))
sage: R == S
True
sage: repr(S(2777))[3:]
```

```
'4&2&1&0&2'

sage: R = ZpCA(5, 15, print_mode='bars', print_sep='&'); S = loads(dumps(R))
sage: R == S
True
sage: repr(S(2777))[3:]
'4&2&1&0&2'
```

class pAdicFieldCappedRelative (*p, prec, print_mode, names*)

An implementation of p-adic fields with capped relative precision.

EXAMPLES:

```
sage: K = Qp(17, 1000000)
sage: K = Qp(next_prime(10^60))
```

random_element (*algorithm='default'*)

Returns a random element of `self`, optionally using the `algorithm` argument to decide how it generates the element. Algorithms currently implemented:

- default: Choose an integer k using the standard distribution on the integers. Then choose an integer a uniformly in the range $0 \leq a < p^N$ where N is the precision cap of `self`. Return `self(p^k * a, absprec = k + self.precision_cap())`.

EXAMPLES:

```
sage: Qp(17,6).random_element()
15*17^-8 + 10*17^-7 + 3*17^-6 + 2*17^-5 + 11*17^-4 + 6*17^-3 + O(17^-2)
```

class pAdicRingCappedAbsolute (*p, prec, print_mode, names*)

An implementation of the p-adic integers with capped absolute precision.

class pAdicRingCappedRelative (*p, prec, print_mode, names*)

An implementation of the p-adic integers with capped relative precision.

class pAdicRingFixedMod (*p, prec, print_mode, names*)

An implementation of the p-adic integers using fixed modulus.

fraction_field (*print_mode=None*)

Would normally return $\mathbb{Q}_p$, but there is no implementation of $\mathbb{Q}_p$ matching this ring so this raises an error

If you want to be able to divide with elements of a fixed modulus p-adic ring, you must cast explicitly.

EXAMPLES:

```
sage: ZpFM(5).fraction_field()
...
TypeError: This implementation of the p-adic ring does not support fields of fractions.

sage: a = ZpFM(5)(4); b = ZpFM(5)(5)
```

27.11 p-Adic Extension Leaves.

The final classes for extensions of Zp and Qp (ie classes that are not just designed to be inherited from).

AUTHORS:

- David Roe

class EisensteinExtensionFieldCappedRelative(*prepoly, poly, prec, halt, print_mode, shift_seed, names*)

TESTS:

```
sage: R = Qp(3, 10000, print_pos=False); S.<x> = ZZ[]; f = x^3 + 9*x - 3
sage: W.<w> = R.ext(f); W == loads(dumps(W))
True
```

class EisensteinExtensionRingCappedAbsolute(*prepoly, poly, prec, halt, print_mode, shift_seed, names*)

TESTS:

```
sage: R = ZpCA(3, 10000, print_pos=False); S.<x> = ZZ[]; f = x^3 + 9*x - 3
sage: W.<w> = R.ext(f); W == loads(dumps(W))
True
```

class EisensteinExtensionRingCappedRelative(*prepoly, poly, prec, halt, print_mode, shift_seed, names*)

TESTS:

```
sage: R = Zp(3, 10000, print_pos=False); S.<x> = ZZ[]; f = x^3 + 9*x - 3
sage: W.<w> = R.ext(f); W == loads(dumps(W))
True
```

class EisensteinExtensionRingFixedMod(*prepoly, poly, prec, halt, print_mode, shift_seed, names*)

TESTS:

```
sage: R = ZpFM(3, 10000, print_pos=False); S.<x> = ZZ[]; f = x^3 + 9*x - 3
sage: W.<w> = R.ext(f); W == loads(dumps(W))
True
```

class UnramifiedExtensionFieldCappedRelative(*prepoly, poly, prec, halt, print_mode, shift_seed, names*)

TESTS:

```
sage: R.<a> = QqCR(27,10000); R == loads(dumps(R))
True
```

class UnramifiedExtensionRingCappedAbsolute(*prepoly, poly, prec, halt, print_mode, shift_seed, names*)

TESTS:

```
sage: R.<a> = ZqCA(27,10000); R == loads(dumps(R))
True
```

class UnramifiedExtensionRingCappedRelative(*prepoly, poly, prec, halt, print_mode, shift_seed, names*)

TESTS:

```
sage: R.<a> = ZqCR(27,10000); R == loads(dumps(R))
True
```

class UnramifiedExtensionRingFixedMod(*prepoly, poly, prec, halt, print_mode, shift_seed, names*)

TESTS:

```
sage: R.<a> = ZqFM(27,10000); R == loads(dumps(R))
True
```

27.12 Local Generic Element.

This file contains a common superclass for p-adic elements and power series elements.

AUTHORS:

- David Roe

class LocalGenericElement()

is_integral()

Returns whether self is an integral element.

INPUT:

•`self` – a local ring element

OUTPUT:

•boolean – whether `self` is an integral element.

EXAMPLES:

```
sage: R = Qp(3,20)
sage: a = R(7/3); a.is_integral()
False
sage: b = R(7/5); b.is_integral()
True
```

is_unit()

Returns whether self is a unit

INPUT:

•`self` – a local ring element

OUTPUT:

•boolean – whether `self` is a unit

EXAMPLES:

```
sage: R = Zp(3,20,'capped-rel'); K = Qp(3,20,'capped-rel')
sage: R(0).is_unit()
False
sage: R(1).is_unit()
True
sage: R(2).is_unit()
True
sage: R(3).is_unit()
False
```

TESTS:

```
sage: R(4).is_unit()
True
sage: R(6).is_unit()
False
sage: R(9).is_unit()
False
sage: K(0).is_unit()
False
sage: K(1).is_unit()
True
sage: K(2).is_unit()
```

```
True
sage: K(3).is_unit()
False
sage: K(4).is_unit()
True
sage: K(6).is_unit()
False
sage: K(9).is_unit()
False
sage: K(1/3).is_unit()
False
sage: K(1/9).is_unit()
False
```

normalized_valuation()

Returns the normalized valuation of this local ring element, i.e., the valuation divided by the absolute ramification index.

INPUT:

`self` – a local ring element.

OUTPUT:

rational – the normalized valuation of `self`.

EXAMPLES:

```
sage: Q7 = Qp(7)
sage: R.<x> = Q7[]
sage: F.<z> = Q7.ext(x^3+7*x+7)
sage: z.normalized_valuation()
1/3
```

slice()

Returns the sum of the $p^{i+FOO*k}$ terms of the series expansion of self, for $i + FOO * k$ between i and $j - 1$ inclusive, and FOO an arbitrary integer. Behaves analogously to the slice function for lists.

INPUT:

- `self` – a p-adic element
- `i` – an integer
- `j` – an integer
- `k` – a positive integer, default value 1

EXAMPLES: sage: R = Zp(5, 6, 'capped-rel') sage: a = R(1/2); a 3 + 2*5 + 2*5^2 + 2*5^3 + 2*5^4 + 2*5^5 + O(5^6) sage: a.slice(2, 4) 2*5^2 + 2*5^3 + O(5^4) sage: a.slice(1, 6, 2) 2*5 + 2*5^3 + 2*5^5 + O(5^6) sage: a.slice(5, 4) O(5^4)

sqrt()

TODO: document what "extend" and "all" do

INPUT:

- `self` – a local ring element

OUTPUT:

- local ring element – the square root of `self`

EXAMPLES:

```
sage: R = Zp(13, 10, 'capped-rel', 'series')
sage: a = sqrt(R(-1)); a * a
12 + 12*13 + 12*13^2 + 12*13^3 + 12*13^4 + 12*13^5 + 12*13^6 + 12*13^7 + 12*13^8 + 12*13^9 +
sage: sqrt(R(4))
2 + O(13^10)
```

```
sage: sqrt(R(4/9)) * 3
2 + O(13^10)
```

27.13 p-Adic Generic Element.

Elements of p-Adic Rings and Fields

AUTHOR:

- David Roe
- Genya Zaytman: documentation
- David Harvey: doctests

clear_mpz_globals()

gmp_randrange()

init_mpz_globals()

class **pAdicGenericElement**()

abs()

Returns the p-adic absolute value of self.

This is normalized so that the absolute value of p is 1/p.

INPUT:

```
- prec -- Integer.  The precision of the real field in
  which the answer is returned.  If None, returns a
  rational for absolutely unramified fields, or a real
  with 53 bits of precision if ramified.
```

EXAMPLES:

```
sage: a = Qp(5)(15); a.abs()
1/5
sage: a.abs(53)
0.200000000000000
```

additive_order()

Returns the additive order of self, where self is considered to be zero if it is zero modulo $p^{mbox{prec}}$.

INPUT:

```
self -- a p-adic element
prec -- an integer
```

OUTPUT:

```
integer -- the additive order of self
```

EXAMPLES:

```
sage: R = Zp(7, 4, 'capped-rel', 'series'); a = R(7^3); a.additive_order(3)
1
sage: a.additive_order(4)
+Infinity
```

```
sage: R = Zp(7, 4, 'fixed-mod', 'series'); a = R(7^5); a.additive_order(6)
1
```

algdep()

Returns a polynomial of degree at most n which is approximately satisfied by this number. Note that the returned polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n.

ALGORITHM:

```
Uses the PARI C-library algdep command.
```

INPUT:

```
- self -- a p-adic element
- n -- an integer
```

OUTPUT:

```
polynomial -- degree n polynomial aproximately satisfied by self
```

EXAMPLES:

```
sage: K = Qp(3,20,'capped-rel','series'); R = Zp(3,20,'capped-rel','series')
sage: a = K(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12 + 2*3^15 + 2*3^16 + 3^17 + 2
sage: a.algdep(1)
19*x - 7
sage: K2 = Qp(7,20,'capped-rel')
sage: b = K2.zeta(); b.algdep(2)
x^2 - x + 1
sage: K2 = Qp(11,20,'capped-rel')
sage: b = K2.zeta(); b.algdep(4)
x^4 - x^3 + x^2 - x + 1
sage: a = R(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12 + 2*3^15 + 2*3^16 + 3^17 + 2
sage: a.algdep(1)
19*x - 7
sage: R2 = Zp(7,20,'capped-rel')
sage: b = R2.zeta(); b.algdep(2)
x^2 - x + 1
sage: R2 = Zp(11,20,'capped-rel')
sage: b = R2.zeta(); b.algdep(4)
x^4 - x^3 + x^2 - x + 1
```

algebraic_dependency()

Returns a polynomial of degree at most n which is approximately satisfied by this number. Note that the returned polynomial need not be irreducible, and indeed usually won't be if this number is a good approximation to an algebraic number of degree less than n.

ALGORITHM:

```
Uses the PARI C-library algdep command.
```

INPUT:

```
- self -- a p-adic element
- n -- an integer
```

OUTPUT:

```
polynomial -- degree n polynomial aproximately satisfied by self
```

EXAMPLES:

```
sage: K = Qp(3,20,'capped-rel','series'); R = Zp(3,20,'capped-rel','series')
sage: a = K(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12 + 2*3^15 + 2*3^16 + 3^17 + 2
sage: a.algebraic_dependency(1)
19*x - 7
sage: K2 = Qp(7,20,'capped-rel')
sage: b = K2.zeta(); b.algebraic_dependency(2)
x^2 - x + 1
sage: K2 = Qp(11,20,'capped-rel')
sage: b = K2.zeta(); b.algebraic_dependency(4)
x^4 - x^3 + x^2 - x + 1
sage: a = R(7/19); a
1 + 2*3 + 3^2 + 3^3 + 2*3^4 + 2*3^5 + 3^8 + 2*3^9 + 3^11 + 3^12 + 2*3^15 + 2*3^16 + 3^17 + 2
sage: a.algebraic_dependency(1)
19*x - 7
sage: R2 = Zp(7,20,'capped-rel')
sage: b = R2.zeta(); b.algebraic_dependency(2)
x^2 - x + 1
sage: R2 = Zp(11,20,'capped-rel')
sage: b = R2.zeta(); b.algebraic_dependency(4)
x^4 - x^3 + x^2 - x + 1
```

is_square()

Returns whether self is a square

INPUT:

```
self -- a p-adic element
```

OUTPUT:

```
boolean -- whether self is a square
```

EXAMPLES:

```
sage: R = Zp(3,20,'capped-rel')
sage: R(0).is_square()
True
sage: R(1).is_square()
True
sage: R(2).is_square()
False
```

TESTS:

```
sage: R(3).is_square()
False
sage: R(4).is_square()
True
sage: R(6).is_square()
False
sage: R(9).is_square()
True

sage: R2 = Zp(2,20,'capped-rel')
sage: R2(0).is_square()
True
sage: R2(1).is_square()
True
sage: R2(2).is_square()
False
```

```
sage: R2(3).is_square()
False
sage: R2(4).is_square()
True
sage: R2(5).is_square()
False
sage: R2(6).is_square()
False
sage: R2(7).is_square()
False
sage: R2(8).is_square()
False
sage: R2(9).is_square()
True

sage: K = Qp(3,20,'capped-rel')
sage: K(0).is_square()
True
sage: K(1).is_square()
True
sage: K(2).is_square()
False
sage: K(3).is_square()
False
sage: K(4).is_square()
True
sage: K(6).is_square()
False
sage: K(9).is_square()
True
sage: K(1/3).is_square()
False
sage: K(1/9).is_square()
True

sage: K2 = Qp(2,20,'capped-rel')
sage: K2(0).is_square()
True
sage: K2(1).is_square()
True
sage: K2(2).is_square()
False
sage: K2(3).is_square()
False
sage: K2(4).is_square()
True
sage: K2(5).is_square()
False
sage: K2(6).is_square()
False
sage: K2(7).is_square()
False
sage: K2(8).is_square()
False
sage: K2(9).is_square()
True
sage: K2(1/2).is_square()
False
```

```
sage: K2(1/4).is_square()
True
```

multiplicative_order()

Returns the multiplicative order of self, where self is considered to be one if it is one modulo $p^{mbox{prec}}$.

INPUT:

```
self -- a p-adic element
prec -- an integer
```

OUTPUT:

```
integer -- the multiplicative order of self
```

EXAMPLES:

```
sage: K = Qp(5,20,'capped-rel')
sage: K(-1).multiplicative_order(20)
2
sage: K(1).multiplicative_order(20)
1
sage: K(2).multiplicative_order(20)
+Infinity
sage: K(3).multiplicative_order(20)
+Infinity
sage: K(4).multiplicative_order(20)
+Infinity
sage: K(5).multiplicative_order(20)
+Infinity
sage: K(25).multiplicative_order(20)
+Infinity
sage: K(1/5).multiplicative_order(20)
+Infinity
sage: K(1/25).multiplicative_order(20)
+Infinity
sage: K.zeta().multiplicative_order(20)
4

sage: R = Zp(5,20,'capped-rel')
sage: R(-1).multiplicative_order(20)
2
sage: R(1).multiplicative_order(20)
1
sage: R(2).multiplicative_order(20)
+Infinity
sage: R(3).multiplicative_order(20)
+Infinity
sage: R(4).multiplicative_order(20)
+Infinity
sage: R(5).multiplicative_order(20)
+Infinity
sage: R(25).multiplicative_order(20)
+Infinity
sage: R.zeta().multiplicative_order(20)
4
```

ordp()

Returns the valuation of self, normalized so that the valuation of p is 1

INPUT:

self -- a p-adic element

OUTPUT:

integer -- the valuation of self, normalized so that the valuation of p is 1

EXAMPLES:

```
sage: R = Zp(5,20,'capped-rel')
sage: R(0).ordp()
+Infinity
sage: R(1).ordp()
0
sage: R(2).ordp()
0
sage: R(5).ordp()
1
sage: R(10).ordp()
1
sage: R(25).ordp()
2
sage: R(50).ordp()
2
sage: R(1/2).ordp()
0
```

rational_reconstruction()

Returns a rational approximation to this p-adic number

INPUT:

self -- a p-adic element

OUTPUT:

rational -- an approximation to self

EXAMPLES:

```
sage: R = Zp(5,20,'capped-rel')
sage: for i in range(11):
...       for j in range(1,10):
...           if j == 5:
...               continue
...           assert i/j == R(i/j).rational_reconstruction()
```

square_root()

Returns the square root of this p-adic number

INPUT:

- self -- a p-adic element
- extend -- bool (default: True); if True, return a square root in an extension if necessary; if False and no root exists in the given ring or field, raise a ValueError
- all -- bool (default: False); if True, return a list of all square roots

OUTPUT:

p-adic element -- the square root of this p-adic number

If all = False, the square root chosen is the one whose reduction mod p is in the range [0, p/2).

EXAMPLES:

```
sage: R = Zp(3,20,'capped-rel', 'val-unit')
sage: R(0).square_root()
0
sage: R(1).square_root()
1 + O(3^20)
sage: R(2).square_root(extend = False)
...
ValueError: element is not a square
sage: R(4).square_root() == R(-2)
True
sage: R(9).square_root()
3 * 1 + O(3^21)

When p = 2, the precision of the square root is one less
than the input:

sage: R2 = Zp(2,20,'capped-rel')
sage: R2(0).square_root()
0
sage: R2(1).square_root()
1 + O(2^19)
sage: R2(4).square_root()
2 + O(2^20)

sage: R2(9).square_root() == R2(3, 19) or R2(9).square_root() == R2(-3, 19)
True

sage: R2(17).square_root()
1 + 2^3 + 2^5 + 2^6 + 2^7 + 2^9 + 2^10 + 2^13 + 2^16 + 2^17 + O(2^19)

sage: R3 = Zp(5,20,'capped-rel')
sage: R3(0).square_root()
0
sage: R3(1).square_root()
1 + O(5^20)
sage: R3(-1).square_root() == R3.teichmuller(2) or R3(-1).square_root() == R3.teichmuller(3)
True
```

TESTS:

```
sage: R = Qp(3,20,'capped-rel')
sage: R(0).square_root()
0
sage: R(1).square_root()
1 + O(3^20)
sage: R(4).square_root() == R(-2)
True
sage: R(9).square_root()
3 + O(3^21)
sage: R(1/9).square_root()
3^-1 + O(3^19)

sage: R2 = Qp(2,20,'capped-rel')
sage: R2(0).square_root()
0
sage: R2(1).square_root()
1 + O(2^19)
sage: R2(4).square_root()
```

```
2 + O(2^20)
sage: R2(9).square_root() == R2(3,19) or R2(9).square_root() == R2(-3,19)
True
sage: R2(17).square_root()
1 + 2^3 + 2^5 + 2^6 + 2^7 + 2^9 + 2^10 + 2^13 + 2^16 + 2^17 + O(2^19)

sage: R3 = Qp(5,20,'capped-rel')
sage: R3(0).square_root()
0
sage: R3(1).square_root()
1 + O(5^20)
sage: R3(-1).square_root() == R3.teichmuller(2) or R3(-1).square_root() == R3.teichmuller(3)
True
```

str()

Returns a string representation of self.

EXAMPLES:

```
sage: Zp(5,5,print_mode='bars')(1/3).str()[3:]
'1|3|1|3|2'
```

val_unit()

Returns (self.valuation(), self.unit_part()).

EXAMPLES:

```
sage: Zp(5,5)(5).val_unit()
(1, 1 + O(5^5))
```

valuation()

Returns the valuation of self.

INPUT:

```
- self -- a p-adic element
```

OUTPUT:

```
- integer -- the valuation of self
```

EXAMPLES:

```
sage: R = Zp(17, 4,'capped-rel')
sage: a = R(2*17^2)
sage: a.valuation()
2
sage: R = Zp(5, 4,'capped-rel')
sage: R(0).valuation()
+Infinity
```

TESTS:

```
sage: R(1).valuation()
0
sage: R(2).valuation()
0
sage: R(5).valuation()
1
sage: R(10).valuation()
1
sage: R(25).valuation()
2
sage: R(50).valuation()
```

```
2
sage: R = Qp(17, 4)
sage: a = R(2*17^2)
sage: a.valuation()
2
sage: R = Qp(5, 4)
sage: R(0).valuation()
+Infinity
sage: R(1).valuation()
0
sage: R(2).valuation()
0
sage: R(5).valuation()
1
sage: R(10).valuation()
1
sage: R(25).valuation()
2
sage: R(50).valuation()
2
sage: R(1/2).valuation()
0
sage: R(1/5).valuation()
-1
sage: R(1/10).valuation()
-1
sage: R(1/25).valuation()
-2
sage: R(1/50).valuation()
-2
```

27.14 p-Adic Base Generic Element.

A common superclass for features shared among all elements of $\mathbb{Z}_p$ and $\mathbb{Q}_p$ (regardless of implementation).

AUTHORS:

- David Roe

class pAdicBaseGenericElement()

abs()

Returns the p-adic absolute value of `self`.

This is normalized so that the absolute value of p is $1/p$.

INPUT:

- `prec` – Integer. The precision of the real field in which the answer is returned. If `None`, returns a rational for absolutely unramified fields, or a real with 53 bits of precision if ramified.

EXAMPLES: sage: a = Qp(5)(15); a.abs() 1/5 sage: a.abs(53) 0.200000000000000

exp()

Compute the p-adic exponential of any element of $\mathbb{Z}_p$ where the series converges.

EXAMPLES:

Borrowed from log.:

```
sage: Z13 = Zp(13, 10, print_mode='series')
sage: a = Z13(13 + 6*13**2 + 2*13**3 + 5*13**4 + 10*13**6 + 13**7 + 11*13**8 + 8*13**9).add_
13 + 6*13^2 + 2*13^3 + 5*13^4 + 10*13^6 + 13^7 + 11*13^8 + 8*13^9 + O(13^10)
sage: a.exp()
1 + 13 + O(13^10)
sage: Q13 = Qp(13, 10, print_mode='series')
sage: a = Q13(13 + 6*13**2 + 2*13**3 + 5*13**4 + 10*13**6 + 13**7 + 11*13**8 + 8*13**9).add_
13 + 6*13^2 + 2*13^3 + 5*13^4 + 10*13^6 + 13^7 + 11*13^8 + 8*13^9 + O(13^10)
sage: a.exp()
1 + 13 + O(13^10)
```

The next few examples illustrate precision when computing p-adic exps. First we create a field with emph{default} precision 10.:

```
sage: R = Zp(5,10, print_mode='series')
sage: e = R(2*5 + 2*5**2 + 4*5**3 + 3*5**4 + 5**5 + 3*5**7 + 2*5**8 + 4*5**9).add_bigoh(10);
2*5 + 2*5^2 + 4*5^3 + 3*5^4 + 5^5 + 3*5^7 + 2*5^8 + 4*5^9 + O(5^10)
sage: e.exp()*R.teichmuller(4)
4 + 2*5 + 3*5^3 + O(5^10)

sage: K = Qp(5,10, print_mode='series')
sage: e = K(2*5 + 2*5**2 + 4*5**3 + 3*5**4 + 5**5 + 3*5**7 + 2*5**8 + 4*5**9).add_bigoh(10);
2*5 + 2*5^2 + 4*5^3 + 3*5^4 + 5^5 + 3*5^7 + 2*5^8 + 4*5^9 + O(5^10)
sage: e.exp()*K.teichmuller(4)
4 + 2*5 + 3*5^3 + O(5^10)
```

TESTS:

Check that results are consistent over a range of precision:

```
sage: max_prec = 40
sage: p = 3
sage: K = Zp(p, max_prec)
sage: full_exp = (K(p)).exp()
sage: for prec in range(2, max_prec):
...       ll = (K(p).add_bigoh(prec)).exp()
...       assert ll == full_exp
...       assert ll.precision_absolute() == prec
sage: K = Qp(p, max_prec)
sage: full_exp = (K(p)).exp()
sage: for prec in range(2, max_prec):
...       ll = (K(p).add_bigoh(prec)).exp()
...       assert ll == full_exp
...       assert ll.precision_absolute() == prec
```

AUTHORS:

- Genya Zaytman (2007-02-15)

log()

Compute the p-adic logarithm of any unit in $\mathbb{Z}_p$. (See below for normalization.)

The usual power series for log with values in the additive group of $\mathbb{Z}_p$ only converges for 1-units (units congruent to 1 modulo p). However, there is a unique extension of log to a homomorphism defined on all the units. If $u = a \cdot v$ is a unit with $v \equiv 1 \pmod{p}$ and a a Teichmuller representative, then we define $log(u) = log(v)$. This is the correct extension because the units U of $\mathbb{Z}_p$ split as a product $U = V \times \langle w \rangle$, where V is the subgroup of 1-units and w is a $(p-1)$'$strootofunity.The$'$\langle w \rangle$ factor is torsion, so must go to 0 under any homomorphism to the torsion free group $(\mathbb{Z}_p, +)$.

INPUTS:

- `self` – a p-adic element.
- `branch` – A choice of branch, ie a choice of logarithm of the uniformizer. This choice can be made arbitrarily.

NOTES:

What some other systems do:

•PARI: Seems to define log the same way as we do.

•MAGMA: Gives an error when unit is not a 1-unit.

ALGORITHM:

Input: Some p-adic unit u (or non-unit if `branch` is specified).

1.Check that the input p-adic number is really a unit (i.e., valuation 0), or take the unit part and multiply by `branch * self.valuation()` if not and `branch` specified.

2.Let $1 - x = u^{p-1}$, which is a 1-unit.

3.Use the series expansion

..math

```
\log(1-x) = F(x) = -x - 1/2*x^2 - 1/3*x^3 - 1/4*x^4 - 1/5*x^5 - \cdots
```

to compute the logarithm $log(u^{p-1})$. Use enough terms so that terms added on are zero

1.Then

..math

```
\log(u) = \log(u^{p-1})/(p-1) = F(1-u^{p-1})/(p-1).
```

EXAMPLES:

```
sage: Z13 = Zp(13, 10, print_mode='series')
sage: a = Z13(14); a
1 + 13 + O(13^10)
```

Note that the relative precision decreases when we take log: it is the absolute precision that is preserved.:

```
sage: a.log()
13 + 6*13^2 + 2*13^3 + 5*13^4 + 10*13^6 + 13^7 + 11*13^8 + 8*13^9 + O(13^10)
sage: Q13 = Qp(13, 10, print_mode='series')
sage: a = Q13(14); a
1 + 13 + O(13^10)
sage: a.log()
13 + 6*13^2 + 2*13^3 + 5*13^4 + 10*13^6 + 13^7 + 11*13^8 + 8*13^9 + O(13^10)
```

The next few examples illustrate precision when computing p-adic logs. First we create a field with emph{default} precision 10.:

```
sage: R = Zp(5,10, print_mode='series')
sage: e = R(389); e
4 + 2*5 + 3*5^3 + O(5^10)
sage: e.log()
2*5 + 2*5^2 + 4*5^3 + 3*5^4 + 5^5 + 3*5^7 + 2*5^8 + 4*5^9 + O(5^10)
sage: K = Qp(5,10, print_mode='series')
sage: e = K(389); e
4 + 2*5 + 3*5^3 + O(5^10)
sage: e.log()
2*5 + 2*5^2 + 4*5^3 + 3*5^4 + 5^5 + 3*5^7 + 2*5^8 + 4*5^9 + O(5^10)
```

Check that results are consistent over a range of precision:

```
sage: max_prec = 40
sage: p = 3
sage: K = Zp(p, max_prec)
sage: full_log = (K(1 + p)).log()
sage: for prec in range(2, max_prec):
...       ll = (K(1 + p).add_bigoh(prec)).log()
```

```
...       assert ll == full_log
...       assert ll.precision_absolute() == prec
```

AUTHORS:

- William Stein: initial version
- David Harvey (2006-09-13): corrected subtle precision bug (need to take denominators into account! – see trac #53)
- Genya Zaytman (2007-02-14): adapted to new p-adic class

TODO:

- Currently implemented as $O(N^2)$. This can be improved to soft-$O(N)$ using algorithm described by Dan Bernstein: `http://cr.yp.to/lineartime/multapps-20041007.pdf`

minimal_polynomial()

Returns a minimal polynomial of this p-adic element, i.e., `x - self`

INPUT:

- `self` – a p-adic element
- `name` – string: the name of the variable

OUTPUT:

- `polynomial` – a minimal polynomial of this p-adic element, i.e., `x - self`

EXAMPLES:

```
sage: Zp(5,5)(1/3).minimal_polynomial('x')
(1 + O(5^5))*x + (3 + 5 + 3*5^2 + 5^3 + 3*5^4 + O(5^5))
```

norm()

Returns the norm of this p-adic element over the ground ring.

NOTE! This is not the p-adic absolute value. This is a field theoretic norm down to a ground ring. If you want the p-adic absolute value, use the `abs()` function instead.

INPUT:

- `self` – a p-adic element
- `ground` – a subring of the ground ring (default: base ring)

OUTPUT:

- element – the norm of this p-adic element over the ground ring

EXAMPLES:

```
sage: Zp(5)(5).norm()
5 + O(5^21)
```

trace()

Returns the trace of this p-adic element over the ground ring

INPUT:

- `self` – a p-adic element
- `ground` – a subring of the ground ring (default: base ring)

OUTPUT:

- `element` – the trace of this p-adic element over the ground ring

EXAMPLES:

```
sage: Zp(5,5)(5).trace()
5 + O(5^6)
```

27.15 p-Adic Capped Relative Element.

Elements of p-Adic Rings with Capped Relative Precision

AUTHORS:

- David Roe
- Genya Zaytman: documentation
- David Harvey: doctests

TESTS:

```
sage: M = MatrixSpace(pAdicField(3,100),2)
sage: (M([1,0,0,90]) - (1+O(3^100)) * M(1)).left_kernel()
Vector space of degree 2 and dimension 1 over 3-adic Field with capped relative precision 100
Basis matrix:
[1 + O(3^100)            0]
```

clear_mpz_globals()

gmp_randrange()

init_mpz_globals()

class pAdicCappedRelativeElement()

add_bigoh()

Returns a new element with absolute precision decreased to absprec.

INPUT:

```
self -- a p-adic element
absprec -- an integer
```

OUTPUT:

```
element -- self with precision set to the minimum of
           self's precision and absprec
```

EXAMPLE:

```
sage: R = Zp(7,4,'capped-rel','series'); a = R(8); a.add_bigoh(1)
1 + O(7)
sage: b = R(0); b.add_bigoh(3)
O(7^3)
sage: R = Qp(7,4); a = R(8); a.add_bigoh(1)
1 + O(7)
sage: b = R(0); b.add_bigoh(3)
O(7^3)

The precision never increases::

sage: R(4).add_bigoh(2).add_bigoh(4)
4 + O(7^2)

Another example that illustrates that the precision does
not increase::
```

```
sage: k = Qp(3,5)
sage: a = k(1234123412/3^70); a
2*3^-70 + 3^-69 + 3^-68 + 3^-67 + O(3^-65)
sage: a.add_bigoh(2)
2*3^-70 + 3^-69 + 3^-68 + 3^-67 + O(3^-65)

sage: k = Qp(5,10)
sage: a = k(1/5^3 + 5^2); a
5^-3 + 5^2 + O(5^7)
sage: a.add_bigoh(2)
5^-3 + O(5^2)
sage: a.add_bigoh(-1)
5^-3 + O(5^-1)
```

copy()

Returns a copy of self.

EXAMPLES:

```
sage: a = Zp(5,6)(17); b = a.copy()
sage: a == b
True
sage: a is b
False
```

is_equal_to()

Returns whether self is equal to right modulo $p^{mbox{absprec}}$.

if absprec is None, returns True if self and right are equal to the minimum of their precisions.

INPUT:

```
- self -- a p-adic element
- right -- a p-addic element
- absprec -- an integer or None
```

OUTPUT:

```
- boolean -- whether self is equal to right (modulo $p^{\mbox{absprec}}$)
```

EXAMPLES:

```
sage: R = Zp(5, 10); a = R(0); b = R(0, 3); c = R(75, 5)
sage: aa = a + 625; bb = b + 625; cc = c + 625
sage: a.is_equal_to(aa), a.is_equal_to(aa, 4), a.is_equal_to(aa, 5)
(False, True, False)
sage: a.is_equal_to(aa, 15)
...
PrecisionError: Elements not known to enough precision

sage: a.is_equal_to(a, 50000)
True

sage: a.is_equal_to(b), a.is_equal_to(b, 2)
(True, True)
sage: a.is_equal_to(b, 5)
...
PrecisionError: Elements not known to enough precision

sage: b.is_equal_to(b, 5)
...
PrecisionError: Elements not known to enough precision
```

```
sage: b.is_equal_to(bb, 3)
True
sage: b.is_equal_to(bb, 4)
...
PrecisionError: Elements not known to enough precision

sage: c.is_equal_to(b, 2), c.is_equal_to(b, 3)
(True, False)
sage: c.is_equal_to(b, 4)
...
PrecisionError: Elements not known to enough precision

sage: c.is_equal_to(cc, 2), c.is_equal_to(cc, 4), c.is_equal_to(cc, 5)
(True, True, False)
```

TESTS:

```
sage: aa.is_equal_to(a), aa.is_equal_to(a, 4), aa.is_equal_to(a, 5)
(False, True, False)
sage: aa.is_equal_to(a, 15)
...
PrecisionError: Elements not known to enough precision

sage: b.is_equal_to(a), b.is_equal_to(a, 2)
(True, True)
sage: b.is_equal_to(a, 5)
...
PrecisionError: Elements not known to enough precision

sage: bb.is_equal_to(b, 3)
True
sage: bb.is_equal_to(b, 4)
...
PrecisionError: Elements not known to enough precision

sage: b.is_equal_to(c, 2), b.is_equal_to(c, 3)
(True, False)
sage: b.is_equal_to(c, 4)
...
PrecisionError: Elements not known to enough precision

sage: cc.is_equal_to(c, 2), cc.is_equal_to(c, 4), cc.is_equal_to(c, 5)
(True, True, False)
```

is_zero()

Returns whether self is zero modulo $p^{mbox{absprec}}$.

If absprec is None, returns True if this element is indistinguishable from zero.

INPUT:

```
- self -- a p-adic element
- absprec -- an integer or None
```

OUTPUT:

```
- boolean -- whether self is zero
```

EXAMPLES:

```
sage: R = Zp(5); a = R(0); b = R(0,5); c = R(75)
sage: a.is_zero(), a.is_zero(6)
```

```
(True, True)
sage: b.is_zero(), b.is_zero(5)
(True, True)
sage: c.is_zero(), c.is_zero(2), c.is_zero(3)
(False, True, False)
sage: b.is_zero(6)
...
PrecisionError: Not enough precision to determine if element is zero
```

lift()

Return an integer or rational congruent to self modulo self's precision. If a rational is returned, its denominator will eqaul p^ordp(self).

INPUT:

```
- self -- a p-adic element
```

OUTPUT:

```
- integer -- a integer congruent to self mod $p^{\mbox{prec}}$
```

EXAMPLES:

```
sage: R = Zp(7,4,'capped-rel'); a = R(8); a.lift()
8
sage: R = Qp(7,4); a = R(8); a.lift()
8
sage: R = Qp(7,4); a = R(8/7); a.lift()
8/7
```

lift_to_precision()

Returns another element of the same parent, with absolute precision at least absprec, congruent to self modulo self's precision.

If such lifting would yield an element with precision greater than allowed by the precision cap of self's parent, an error is raised.

EXAMPLES:

```
sage: R = Zp(5); a = R(0); b = R(0,5); c = R(17,3)
sage: a.lift_to_precision(5)
0
sage: b.lift_to_precision(4)
O(5^5)
sage: b.lift_to_precision(8)
O(5^8)
sage: b.lift_to_precision(40)
O(5^40)
sage: c.lift_to_precision(1)
2 + 3*5 + O(5^3)
sage: c.lift_to_precision(8)
2 + 3*5 + O(5^8)
sage: c.lift_to_precision(40)
...
PrecisionError: Precision higher than allowed by the precision cap.
```

list()

Returns a list of coefficients in a power series expansion of self in terms of p. If self is a field element, they start at p^valuation, if a ring element at p^0.

INPUT:

```
- self -- a p-adic element
- lift_mode - 'simple', 'smallest' or 'teichmuller'
```

OUTPUT:

```
- list -- the list of coeficients of self.  These will be integers if lift_mode is 'simple'
```

EXAMPLES:

```
sage: R = Zp(7,6); a = R(12837162817); a
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
sage: L = a.list(); L
[3, 4, 4, 0, 4]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('smallest'); L
[3, -3, -2, 1, -3, 1]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('teichmuller'); L
[3 + 4*7 + 6*7^2 + 3*7^3 + 2*7^5 + O(7^6),
0,
5 + 2*7 + 3*7^3 + O(7^4),
1 + O(7^3),
3 + 4*7 + O(7^2),
5 + O(7)]
sage: sum([L[i] * 7^i for i in range(len(L))])
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)

sage: R = Qp(7,4); a = R(6*7+7**2); a.list()
[6, 1]
sage: a.list('smallest')
[-1, 2]
sage: a.list('teichmuller')
[6 + 6*7 + 6*7^2 + 6*7^3 + O(7^4),
2 + 4*7 + 6*7^2 + O(7^3),
3 + 4*7 + O(7^2),
3 + O(7)]
```

NOTE:

```
use slice operators to get a particular range
```

padded_list()

Returns a list of coeficiants of p starting with p^0 up to p^n exclusive (padded with zeros if needed). If a field element, starts at p^val instead.

INPUT:

```
self -- a p-adic element
n - an integer
lift_mode - 'simple', 'smallest' or 'teichmuller'
```

OUTPUT:

```
list -- the list of coeficients of self
```

EXAMPLES:

```
sage: R = Zp(7,3,'capped-rel'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
sage: R = Qp(7,3); a = R(2*7+7**2); a.padded_list(5)
[2, 1, 0, 0]
```

```
sage: a.padded_list(3)
[2, 1]
```

NOTE:

```
the slice operators throw an error if asked for a slice above the precision
```

precision_absolute()

Returns the absolute precision of self.

This is the power of the maximal ideal modulo which this element is defined.

INPUT:

```
self -- a p-adic element
```

OUTPUT:

```
integer -- the absolute precision of self
```

EXAMPLES:

```
sage: R = Zp(7,3,'capped-rel'); a = R(7); a.precision_absolute()
4
sage: R = Qp(7,3); a = R(7); a.precision_absolute()
4
sage: R(7^-3).precision_absolute()
0
```

precision_relative()

Returns the relative precision of self.

This is the power of the maximal ideal modulo which the unit part of self is defined.

INPUT: self – a p-adic element

OUTPUT: integer – the relative precision of self

EXAMPLES: sage: R = Zp(7,3,'capped-rel'); a = R(7); a.precision_relative() 3 sage: R = Qp(7,3); a = R(7); a.precision_relative() 3 sage: a = R(7^-2, -1); a.precision_relative() 1 sage: a 7^-2 + O(7^-1)

residue()

Reduces this element modulo $p^absprec$.

INPUT:

```
- self -- a p-adic element
- absprec - an integer (defaults to 1)
```

OUTPUT:

```
- element of $Z/(p^{absprec} Z)$ -- self reduced mod p^absprec
```

EXAMPLES:

```
sage: R = Zp(7,4,'capped-rel'); a = R(8); a.residue(1)
1
sage: R = Qp(7,4,'capped-rel'); a = R(8); a.residue(1)
1
sage: a.residue(6)
...
PrecisionError: Not enough precision known in order to compute residue.
sage: b = a/7
sage: b.residue(1)
...
ValueError: Element must have non-negative valuation in order to compute residue.
```

unit_part()

Returns the unit part of self.

INPUT:

- self -- a p-adic element

OUTPUT:

- p-adic element -- the unit part of self

EXAMPLES:

```
sage: R = Zp(17,4,'capped-rel')
sage: a = R(18*17)
sage: a.unit_part()
1 + 17 + O(17^4)
sage: type(a)
<type 'sage.rings.padics.padic_capped_relative_element.pAdicCappedRelativeElement'>
sage: R = Qp(17,4,'capped-rel')
sage: a = R(18*17)
sage: a.unit_part()
1 + 17 + O(17^4)
sage: type(a)
<type 'sage.rings.padics.padic_capped_relative_element.pAdicCappedRelativeElement'>
sage: a = R(2*17^2); a
2*17^2 + O(17^6)
sage: a.unit_part()
2 + O(17^4)
sage: b=1/a; b
9*17^-2 + 8*17^-1 + 8 + 8*17 + O(17^2)
sage: b.unit_part()
9 + 8*17 + 8*17^2 + 8*17^3 + O(17^4)
sage: Zp(5)(75).unit_part()
3 + O(5^20)
```

val_unit()

Returns a pair (self.valuation(), self.unit_part()).

EXAMPLES:

```
sage: R = Zp(5); a = R(75, 20); a
3*5^2 + O(5^20)
sage: a.val_unit()
(2, 3 + O(5^18))
sage: R(0).val_unit()
(+Infinity, O(5^0))
sage: R(0, 10).val_unit()
(10, O(5^0))
```

unpickle_pcre_v1()

Unpickles a capped relative element.

EXAMPLES:

```
sage: from sage.rings.padics.padic_capped_relative_element import unpickle_pcre_v1
sage: R = Zp(5)
sage: a = unpickle_pcre_v1(R, 17, 2, 5); a
2*5^2 + 3*5^3 + O(5^7)
```

27.16 p-Adic Capped Absolute Element.

Elements of p-Adic Rings with Absolute Precision Cap

AUTHORS:

- David Roe
- Genya Zaytman: documentation
- David Harvey: doctests

clear_mpz_globals()

gmp_randrange()

init_mpz_globals()

make_pAdicCappedAbsoluteElement()

Unpickles a capped absolute element.

EXAMPLES:

```
sage: from sage.rings.padics.padic_capped_absolute_element import make_pAdicCappedAbsoluteElemen
sage: R = ZpCA(5)
sage: a = make_pAdicCappedAbsoluteElement(R, 17*25, 5); a
2*5^2 + 3*5^3 + O(5^5)
```

class **pAdicCappedAbsoluteElement**()

add_bigoh()

Returns a new element with absolute precision decreased to `prec`. The precision never increases.

INPUT:

- `self` – a p-adic element
- `prec` – an integer

OUTPUT:

- `element` – `self` with precision set to the minimum of `self`'s precision and `prec`

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs','series'); a = R(8); a.add_bigoh(1)
1 + O(7)

sage: k = ZpCA(3,5)
sage: a = k(41); a
2 + 3 + 3^2 + 3^3 + O(3^5)
sage: a.add_bigoh(7)
2 + 3 + 3^2 + 3^3 + O(3^5)
sage: a.add_bigoh(3)
2 + 3 + 3^2 + O(3^3)
```

copy()

Returns a copy of `self`.

EXAMPLES:

```
sage: a = ZpCA(5,6)(17); b = a.copy()
sage: a == b
True
sage: a is b
False
```

is_equal_to()

Returns whether `self` is equal to `right` modulo `p^absprec`.

INPUT:

- self – a p-adic element
- right – a p-adic element with the same parent
- absprec – an integer

OUTPUT:

- boolean – whether self is equal to right

EXAMPLES:

```
sage: R = ZpCA(2, 6)
sage: R(13).is_equal_to(R(13))
True
sage: R(13).is_equal_to(R(13+2^10))
True
sage: R(13).is_equal_to(R(17), 2)
True
sage: R(13).is_equal_to(R(17), 5)
False
```

is_zero()

Returns whether self is zero modulo p^absprec.

INPUT:

- self – a p-adic element
- prec – an integer

OUTPUT:

- boolean – whether self is zero

EXAMPLES:

```
sage: R = ZpCA(17, 6)
sage: R(0).is_zero()
True
sage: R(17^6).is_zero()
True
sage: R(17^2).is_zero(absprec=2)
True
```

lift()

Returns an integer congruent to this p-adic element modulo p^self.absprec().

EXAMPLES:

```
sage: R = ZpCA(3)
sage: R(10).lift()
10
sage: R(-1).lift()
3486784400
```

lift_to_precision()

Returns a p-adic integer congruent to this p-adic element modulo p^absprec with precision at least absprec.

If such lifting would yield an element with precision greater than allowed by the precision cap of self's parent, an error is raised.

EXAMPLES:

```
sage: R = ZpCA(17)
sage: R(-1,2).lift_to_precision(10)
16 + 16*17 + O(17^10)
sage: R(1,15).lift_to_precision(10)
```

```
1 + O(17^15)
sage: R(1,15).lift_to_precision(30)
...
PrecisionError: Precision higher than allowed by the precision cap.
```

list()

Returns a list of coefficients of p starting with p^0

INPUT:

- `self` – a p-adic element
- `lift_mode` – `'simple'`, `'smallest'` or `'teichmuller'` (default `'simple'`)

OUTPUT:

- `list` – the list of coeficients of `self`

NOTES:

- Returns a list $[a_0, a_1, \ldots, a_n]$ so that:
  - If `lift_mode = 'simple'`, a_i is an integer with $0 \leq a_i < p$.
  - If `lift_mode = 'smallest'`, a_i is an integer with $-p/2 < a_i \leq p/2$.
  - If `lift_mode = 'teichmuller'`, a_i has the same parent as $self$ and $a_i^p \equiv a_i$ modulo `p^(self.precision_absolute() - i)`
- $\sum_{i=0}^{n} a_i \cdot p^i =$ `self`, modulo the precision of `self`.

EXAMPLES:

```
sage: R = ZpCA(7,6); a = R(12837162817); a
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
sage: L = a.list(); L
[3, 4, 4, 0, 4]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('smallest'); L
[3, -3, -2, 1, -3, 1]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('teichmuller'); L
[3 + 4*7 + 6*7^2 + 3*7^3 + 2*7^5 + O(7^6),
O(7^5),
5 + 2*7 + 3*7^3 + O(7^4),
1 + O(7^3),
3 + 4*7 + O(7^2),
5 + O(7)]
sage: sum([L[i] * 7^i for i in range(len(L))])
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
```

multiplicative_order()

Returns the minimum possible multiplicative order of `self`.

INPUT:

- `self` – a p-adic element

OUTPUT:

- the multiplicative order of self. This is the minimum multiplicative order of all elements of $\mathbb{Z}_p$ lifting `self` to infinite precision.

EXAMPLES:

```
sage: R = ZpCA(7, 6)
sage: R(1/3)
5 + 4*7 + 4*7^2 + 4*7^3 + 4*7^4 + 4*7^5 + O(7^6)
```

```
sage: R(1/3).multiplicative_order()
+Infinity
sage: R(7).multiplicative_order()
+Infinity
sage: R(1).multiplicative_order()
1
sage: R(-1).multiplicative_order()
2
sage: R.teichmuller(3).multiplicative_order()
6
```

padded_list()

Returns a list of coefficients of p starting with p^0 up to p^n exclusive (padded with zeros if needed)

INPUT:

- self – a p-adic element
- n - an integer

OUTPUT:

- list – the list of coeficients of self

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
```

NOTE:

this differs from the padded_list method of padic_field_element

the slice operators throw an error if asked for a slice above the precision, while this function works

precision_absolute()

Returns the absolute precision of self.

This is the power of the maximal ideal modulo which this element is defined.

INPUT:

- self – a p-adic element

OUTPUT:

- integer – the absolute precision of self

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs'); a = R(7); a.precision_absolute()
4
```

precision_relative()

Returns the relative precision of self.

This is the power of the maximal ideal modulo which the unit part of self is defined.

INPUT:

- self – a p-adic element

OUTPUT:

- integer – the relative precision of self

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs'); a = R(7); a.precision_relative()
3
```

residue()

Reduces self modulo p^absprec

INPUT:

- `self` – a p-adic element
- `absprec` - an integer

OUTPUT:

- element of $\mathbb{Z}/p^{\text{absprec}}\mathbb{Z}$ – `self` reduced modulo `p^absprec`.

EXAMPLES:

```
sage: R = Zp(7,4,'capped-abs'); a = R(8); a.residue(1)
1
```

unit_part()

Returns the unit part of `self`.

INPUT:

- `self` – a p-adic element

OUTPUT:

- p-adic element – the unit part of `self`

EXAMPLES:

```
sage: R = Zp(17,4,'capped-abs', 'val-unit')
sage: a = R(18*17)
sage: a.unit_part()
18 + O(17^3)
sage: type(a)
<type 'sage.rings.padics.padic_capped_absolute_element.pAdicCappedAbsoluteElement'>
```

val_unit()

Returns a 2-tuple, the first element set to the valuation of `self`, and the second to the unit part of `self`.

If `self = 0`, then the unit part is `O(p^0)`.

EXAMPLES:

```
sage: R = ZpCA(5)
sage: a = R(75, 6); b = a - a
sage: a.val_unit()
(2, 3 + O(5^4))
sage: b.val_unit()
(6, O(5^0))
```

valuation()

Returns the valuation of `self`, ie the largest power of p dividing `self`.

EXAMPLES:

```
sage: R = ZpCA(5)
sage: R(5^5*1827).valuation()
5
```

TESTS:

```
sage: R(1).valuation()
0
sage: R(2).valuation()
0
sage: R(5).valuation()
1
sage: R(10).valuation()
1
sage: R(25).valuation()
2
sage: R(50).valuation()
```

```
2
sage: R(0).valuation()
20
```

27.17 p-Adic Fixed-Mod Element.

Elements of p-Adic Rings with Fixed Modulus

AUTHORS:

- David Roe
- Genya Zaytman: documentation
- David Harvey: doctests

clear_mpz_globals()

gmp_randrange()

init_mpz_globals()

make_pAdicFixedModElement()

Unpickles a capped relative element.

EXAMPLES:

```
sage: from sage.rings.padics.padic_fixed_mod_element import make_pAdicFixedModElement
sage: R = ZpFM(5)
sage: a = make_pAdicFixedModElement(R, 17*25); a
2*5^2 + 3*5^3 + O(5^20)
```

class **pAdicFixedModElement**()

add_bigoh()

Returns a new element truncated modulo p^absprec.

INPUT:

```
- self -- a p-adic element
- absprec -- an integer
```

OUTPUT:

```
- element -- a new element truncated modulo p^absprec.
```

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod','series'); a = R(8); a.add_bigoh(1)
1 + O(7^4)
```

copy()

Returns a copy of self.

EXAMPLES:

```
sage: a = ZpFM(5,6)(17); b = a.copy()
sage: a == b
True
sage: a is b
False
```

is_equal_to()

Returns whether self is equal to right modulo $p^{mbox{absprec}}$.

If absprec is None, returns if self == 0.

INPUT:

```
- self -- a p-adic element
- right -- a p-addic element with the same parent
- absprec -- a positive integer (or None)
```

OUTPUT:

```
boolean -- whether self is equal to right
```

EXAMPLES:

```
sage: R = ZpFM(2, 6)
sage: R(13).is_equal_to(R(13))
True
sage: R(13).is_equal_to(R(13+2^10))
True
sage: R(13).is_equal_to(R(17), 2)
True
sage: R(13).is_equal_to(R(17), 5)
False
```

is_zero()

Returns whether self is zero modulo $p^{mbox{absprec}}$.

INPUT:

```
- self -- a p-adic element
- absprec -- an integer
```

OUTPUT:

```
boolean -- whether self is zero
```

EXAMPLES:

```
sage: R = ZpFM(17, 6)
sage: R(0).is_zero()
True
sage: R(17^6).is_zero()
True
sage: R(17^2).is_zero(absprec=2)
True
```

lift()

Return an integer congruent to self modulo self's precision.

INPUT:

```
- self -- a p-adic element
```

OUTPUT:

```
- integer -- a integer congruent to self mod $p^{\mbox{prec}}$
```

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod'); a = R(8); a.lift()
8
sage: type(a.lift())
<type 'sage.rings.integer.Integer'>
```

lift_to_precision()

Returns self.

For compatibility with other p-adic types.

EXAMPLES:

```
sage: R = ZpFM(5); a = R(5); a.lift_to_precision(7)
5 + O(5^20)
```

list()

Returns a list of coefficients of p starting with p^0.

INPUT:

```
- self -- a p-adic element
- lift_mode -- 'simple', 'smallest' or 'teichmuller' (default 'simple')
```

OUTPUT:

```
- list -- the list of coeficients of self
```

NOTES:

```
Returns a list [a_0, a_1, \ldots, a_n] so that each a_i is an integer
and \sum_{i = 0}^n a_i * p^i = self, modulo the precision cap.
If lift_mode = 'simple', 0 <= a_i < p.
If lift_mode = 'smallest', -p/2 < a_i <= p/2.
If lift_mode = 'teichmuller', a_i^p = a_i, modulo the precision cap.
```

EXAMPLES:

```
sage: R = ZpFM(7,6); a = R(12837162817); a
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
sage: L = a.list(); L
[3, 4, 4, 0, 4]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('smallest'); L
[3, -3, -2, 1, -3, 1]
sage: sum([L[i] * 7^i for i in range(len(L))]) == a
True
sage: L = a.list('teichmuller'); L
[3 + 4*7 + 6*7^2 + 3*7^3 + 2*7^5 + O(7^6),
O(7^6),
5 + 2*7 + 3*7^3 + 6*7^4 + 4*7^5 + O(7^6),
1 + O(7^6),
3 + 4*7 + 6*7^2 + 3*7^3 + 2*7^5 + O(7^6),
5 + 2*7 + 3*7^3 + 6*7^4 + 4*7^5 + O(7^6)]
sage: sum([L[i] * 7^i for i in range(len(L))])
3 + 4*7 + 4*7^2 + 4*7^4 + O(7^6)
```

multiplicative_order()

Returns the minimum possible multiplicative order of self.

INPUT:

```
- self -- a p-adic element
```

OUTPUT:

```
- integer -- the multiplicative order of self.  This is
  the minimum multiplicative order of all elements of Z_p
  lifting self to infinite precision.
```

EXAMPLES:

```
sage: R = ZpFM(7, 6)
sage: R(1/3)
5 + 4*7 + 4*7^2 + 4*7^3 + 4*7^4 + 4*7^5 + O(7^6)
sage: R(1/3).multiplicative_order()
+Infinity
sage: R(7).multiplicative_order()
+Infinity
sage: R(1).multiplicative_order()
1
sage: R(-1).multiplicative_order()
2
sage: R.teichmuller(3).multiplicative_order()
6
```

padded_list()

Returns a list of coefficients of p starting with p^0 up to p^n exclusive (padded with zeros if needed)

INPUT:

```
- self -- a p-adic element
- n -- an integer
```

OUTPUT:

```
- list -- the list of coeficients of self
```

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod'); a = R(2*7+7**2); a.padded_list(5)
[0, 2, 1, 0, 0]
```

NOTE:

```
For elements with positive valuation, this function will
return a list with leading 0s, unlike for field elements.

The slice operators throw an error if asked for a slice
above the precision, while this function works
```

precision_absolute()

Returns the absolute precision of self.

INPUT:

```
- self -- a p-adic element
```

OUTPUT:

```
- integer -- the absolute precision of self
```

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod'); a = R(7); a.precision_absolute()
4
```

precision_relative()

Returns the relative precision of self

INPUT:

```
- self -- a p-adic element
```

OUTPUT:

```
- integer -- the relative precision of self
```

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod'); a = R(7); a.precision_relative()
3
sage: a = R(0); a.precision_relative()
0
```

residue()

Reduces this mod p^prec

INPUT:

```
- self -- a p-adic element
- absprec - an integer (default 1)
```

OUTPUT:

```
- element of Z/(p^prec Z) -- self reduced mod p^prec
```

EXAMPLES:

```
sage: R = Zp(7,4,'fixed-mod'); a = R(8); a.residue(1)
1
```

unit_part()

Returns the unit part of self.

If the valuation of self is positive, then the high digits of the result will be zero.

INPUT:

```
- self -- a p-adic element
```

OUTPUT:

```
- p-adic element -- the unit part of self
```

EXAMPLES:

```
sage: R = Zp(17, 4, 'fixed-mod')
sage: R(5).unit_part()
5 + O(17^4)
sage: R(18*17).unit_part()
1 + 17 + O(17^4)
sage: R(0).unit_part()
O(17^4)
sage: type(R(5).unit_part())
<type 'sage.rings.padics.padic_fixed_mod_element.pAdicFixedModElement'>
sage: R = ZpFM(5, 5); a = R(75); a.unit_part()
3 + O(5^5)
```

val_unit()

Returns a 2-tuple, the first element set to the valuation of self, and the second to the unit part of self.

If self == 0, then the unit part is O(p^self.parent().precision_cap()).

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: a = R(75); b = a - a
sage: a.val_unit()
(2, 3 + O(5^5))
sage: b.val_unit()
(5, O(5^5))
```

valuation()

Returns the valuation of self.

If self is zero, the valuation returned is the precision of the ring.

INPUT:

- self -- a p-adic element

OUTPUT:

- integer -- the valuation of self.

EXAMPLES:

```
sage: R = Zp(17, 4,'fixed-mod')
sage: a = R(2*17^2)
sage: a.valuation()
2
sage: R = Zp(5, 4,'fixed-mod')
sage: R(0).valuation()
4
sage: R(1).valuation()
0
sage: R(2).valuation()
0
sage: R(5).valuation()
1
sage: R(10).valuation()
1
sage: R(25).valuation()
2
sage: R(50).valuation()
2
```

27.18 p-Adic Extension Element.

A common superclass for all elements of extension rings and field of Zp and Qp.

AUTHORS:

- David Roe

class pAdicExtElement()

abs()

Returns the p-adic absolute value of self.

This is normalized so that the absolute value of p is 1/p.

INPUT – prec - Integer. The precision of the real field in which the answer is returned. If None, returns a rational for absolutely unramified fields, or a real with 53 bits of precision if ramified.

EXAMPLES: sage: R = Zp(5,5) sage: S.<x> = ZZ[] sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5 sage: W.<w> = R.ext(f) sage: w.abs() 0.724779663677696

27.19 p-Adic ZZ_pX Element.

A common superclass implementing features shared by all elements that use NTL's ZZ_pX as the fundamental data type.

AUTHORS:

- David Roe

class pAdicZZpXElement()

norm()

Return the absolute or relative norm of this element.

NOTE! This is not the p-adic absolute value. This is a field theoretic norm down to a ground ring. If you want the p-adic absolute value, use the `abs()` function instead.

If `base` is given then `base` must be a subfield of the parent L of `self`, in which case the norm is the relative norm from L to `base`.

In all other cases, the norm is the absolute norm down to $\mathbb{Q}_p$ or $\mathbb{Z}_p$.

EXAMPLES:

```
sage: R = ZpCR(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: ((1+2*w)^5).norm()
1 + 5^2 + O(5^5)
sage: ((1+2*w)).norm()^5
1 + 5^2 + O(5^5)
```

TESTS:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: ((1+2*w)^5).norm()
1 + 5^2 + O(5^5)
sage: ((1+2*w)).norm()^5
1 + 5^2 + O(5^5)
sage: R = ZpFM(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: ((1+2*w)^5).norm()
1 + 5^2 + O(5^5)
sage: ((1+2*w)).norm()^5
1 + 5^2 + O(5^5)
```

trace()

Return the absolute or relative trace of this element.

If `base` is given then `base` must be a subfield of the parent L of `self`, in which case the norm is the relative norm from L to `base`.

In all other cases, the norm is the absolute norm down to $\mathbb{Q}_p$ or $\mathbb{Z}_p$.

EXAMPLES:

```
sage: R = ZpCR(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = (2+3*w)^7
sage: b = (6+w^3)^5
sage: a.trace()
3*5 + 2*5^2 + 3*5^3 + 2*5^4 + O(5^5)
sage: a.trace() + b.trace()
```

```
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
sage: (a+b).trace()
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
```

TESTS:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = (2+3*w)^7
sage: b = (6+w^3)^5
sage: a.trace()
3*5 + 2*5^2 + 3*5^3 + 2*5^4 + O(5^5)
sage: a.trace() + b.trace()
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
sage: (a+b).trace()
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = (2+3*w)^7
sage: b = (6+w^3)^5
sage: a.trace()
3*5 + 2*5^2 + 3*5^3 + 2*5^4 + O(5^5)
sage: a.trace() + b.trace()
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
sage: (a+b).trace()
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
```

27.20 p-Adic `ZZ_pX` CR Element.

This file implements elements of eisenstein and unramified extensions of $\mathbb{Z}_p$ and $\mathbb{Q}_p$ with capped relative precision.

For the parent class see padic_extension_leaves.pyx.

The underlying implementation is through NTL's `ZZ_pX` class. Each element contains the following data:

- `ordp` (`long`) – A power of the uniformizer to scale the unit by. For unramified extensions this uniformizer is p, for eisenstein extensions it is not. A value equal to the maximum value of a long indicates that the element is an exact zero.

- `relprec` (`long`) – A signed integer giving the precision to which this element is defined. For nonzero `relprec`, the absolute value gives the power of the uniformizer modulo which the unit is defined. A positive value indicates that the element is normalized (ie `unit` is actually a unit: in the case of eisenstein extensions the constant term is not divisible by p, in the case of unramified extensions that there is at least one coefficient that is not divisible by p). A negative value indicates that the element may or may not be normalized. A zero value indicates that the element is zero to some precision. If so, `ordp` gives the absolute precision of the element. If `ordp` is greater than `maxordp`, then the element is an exact zero.

- `unit` (`ZZ_pX_c`) – An ntl `ZZ_pX` storing the unit part. The varible x is the uniformizer in the case of eisenstein extensions. If the element is not normalized, the `unit` may or may not actually be a unit. This `ZZ_pX` is created with global ntl modulus determined by the absolute value of `relprec`. If `relprec` is 0, `unit` **is not initialized**, or destructed if normalized and found to be zero. Otherwise, let r be relprec and e be the ramification index over $\mathbb{Q}_p$ or $\mathbb{Z}_p$. Then the modulus of unit is given by $p^{ceil(r/e)}$. Note that all kinds of problems arise if you

try to mix moduli. ZZ_pX_conv_modulus gives a semi-safe way to convert between different moduli without having to pass through ZZX (see sage/libs/ntl/decl.pxi and c_lib/src/ntl_wrap.cpp)

- prime_pow (some subclass of PowComputer_ZZ_pX) – a class, identical among all elements with the same parent, holding common data.
  - prime_pow.deg – The degree of the extension
  - prime_pow.e – The ramification index
  - prime_pow.f – The inertia degree
  - prime_pow.prec_cap – the unramified precision cap. For eisenstein extensions this is the smallest power of p that is zero.
  - prime_pow.ram_prec_cap – the ramified precision cap. For eisenstein extensions this will be the smallest power of x that is indistinugishable from zero.
  - prime_pow.pow_ZZ_tmp, prime_pow.pow_mpz_t_tmp‘‘, prime_pow.pow_Integer – functions for accessing powers of p. The first two return pointers. See sage/rings/padics/pow_computer_ext for examples and important warnings.
  - prime_pow.get_context, prime_pow.get_context_capdiv, prime_pow.get_top_context – obtain an ntl_ZZ_pContext_class corresponding to p^n. The capdiv version divides by prime_pow.e as appropriate. top_context corresponds to p^{prec_cap}.
  - prime_pow.restore_context, prime_pow.restore_context_capdiv, prime_pow.restore_top_context – restores the given context.
  - prime_pow.get_modulus, get_modulus_capdiv, get_top_modulus – Returns a ZZ_pX_Modulus_c* pointing to a polynomial modulus defined modulo p^n (appropriately divided by prime_pow.e in the capdiv case).

EXAMPLES:

An eisenstein extension:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f); W
Eisenstein Extension of 5-adic Ring with capped relative precision 5 in w defined by (1 + O(5^5))*x^5
sage: z = (1+w)^5; z
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 + 4*w^17 + 4*w^20 +
sage: y = z >> 1; y
w^4 + w^5 + 2*w^6 + 4*w^7 + 3*w^9 + w^11 + 4*w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 + 4*w^19 + w^20
sage: y.valuation()
4
sage: y.precision_relative()
20
sage: y.precision_absolute()
24
sage: z - (y << 1)
1 + O(w^25)
sage: (1/w)^12+w
w^-12 + w + O(w^13)
sage: (1/w).parent()
Eisenstein Extension of 5-adic Field with capped relative precision 5 in w defined by (1 + O(5^5))*x^
```

An unramified extension:

```
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: z = (1+a)^5; z
(2*a^2 + 4*a) + (3*a^2 + 3*a + 1)*5 + (4*a^2 + 3*a + 4)*5^2 + (4*a^2 + 4*a + 4)*5^3 + (4*a^2 + 4*a +
sage: z - 1 - 5*a - 10*a^2 - 10*a^3 - 5*a^4 - a^5
O(5^5)
sage: y = z >> 1; y
(3*a^2 + 3*a + 1) + (4*a^2 + 3*a + 4)*5 + (4*a^2 + 4*a + 4)*5^2 + (4*a^2 + 4*a + 4)*5^3 + O(5^4)
sage: 1/a
(3*a^2 + 4) + (a^2 + 4)*5 + (3*a^2 + 4)*5^2 + (a^2 + 4)*5^3 + (3*a^2 + 4)*5^4 + O(5^5)
```

Different printing modes:

```
sage: R = Zp(5, print_mode='digits'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5; W.<w> = R.ex
sage: z = (1+w)^5; repr(z)
'...411040311321031044222131124200011101120110200202330321433201121440323201314400140044441030421100
sage: R = Zp(5, print_mode='bars'); S.<x> = R[]; g = x^3 + 3*x + 3; A.<a> = R.ext(g)
sage: z = (1+a)^5; repr(z)
'...[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4,
sage: R = Zp(5, print_mode='terse'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5; W.<w> = R.ext
sage: z = (1+w)^5; z
6 + 95367431640505*w + 25*w^2 + 95367431640560*w^3 + 5*w^4 + O(w^100)
sage: R = Zp(5, print_mode='val-unit'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5; W.<w> = R
sage: y = (1+w)^5 - 1; y
w^5 * (2090041 + 19073486126901*w + 1258902*w^2 + 674*w^3 + 16785*w^4) + O(w^100)
```

You can get at the underlying ntl unit:

```
sage: z._ntl_rep()
[6 95367431640505 25 95367431640560 5]
sage: y._ntl_rep()
[2090041 19073486126901 1258902 674 16785]
sage: y._ntl_rep_abs()
([5 95367431640505 25 95367431640560 5], 0)
```

NOTES:

```
If you get an error ``internal error: can't grow this
_ntl_gbigint,`` it indicates that moduli are being mixed
inappropriately somewhere.  For example, when calling a function
with a ``ZZ_pX_c`` as an argument, it copies.  If the modulus is not
set to the modulus of the ``ZZ_pX_c``, you can get errors.
```

AUTHORS:

- David Roe (2008-01-01) initial version

make_ZZpXCRElement()

Unpickling.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
```

```
sage: y = W(775, 19); y
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)
sage: loads(dumps(y)) #indirect doctest
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)
```

class pAdicZZpXCRElement()

copy()

Returns a copy of `self`.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: b = W(45, 17); b
4*w^5 + 3*w^7 + w^9 + w^10 + 2*w^11 + w^12 + w^13 + 3*w^14 + w^16 + O(w^17)
sage: c = b.copy(); c
4*w^5 + 3*w^7 + w^9 + w^10 + 2*w^11 + w^12 + w^13 + 3*w^14 + w^16 + O(w^17)
sage: c is b
False
```

is_equal_to()

Returns whether `self` is equal to `right` modulo `self.uniformizer()^absprec`.

If `absprec` is `None`, returns if `self` is equal to `right` modulo the lower of their two precisions.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(47); b = W(47 + 25)
sage: a.is_equal_to(b)
False
sage: a.is_equal_to(b, 7)
True
```

is_zero()

Returns whether the valuation of `self` is at least `absprec`. If `absprec` is `None`, returns if `self` is indistinugishable from zero.

If `self` is an inexact zero of valuation less than `absprec`, raises a PrecisionError.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: O(w^189).is_zero()
True
sage: W(0).is_zero()
True
sage: a = W(675)
sage: a.is_zero()
False
sage: a.is_zero(7)
True
sage: a.is_zero(21)
```

```
False
```

lift_to_precision()

Returns a pAdicZZpXCRElement congruent to self but with absolute precision at least absprec. If setting absprec that high would violate the precision cap, raises a precision error. If self is an inexact zero and absprec is greater than the maximum allowed valuation, raises an error.

Note that the new digits will not necessarily be zero.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(345, 17); a
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15 + O(w^17)
sage: b = a.lift_to_precision(19); b
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15 + w^17 + 2*w^18 + O(
sage: c = a.lift_to_precision(24); c
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15 + w^17 + 2*w^18 + 4*
sage: a._ntl_rep()
[19 35 118 60 121]
sage: b._ntl_rep()
[19 35 118 60 121]
sage: c._ntl_rep()
[19 35 118 60 121]
```

list()

Returns a list giving a series representation of self.

- If lift_mode == 'simple' or 'smallest', the returned list will consist of integers (in the eisenstein case) or a list of lists of integers (in the unramified case). self can be reconstructed as a sum of elements of the list times powers of the uniformiser (in the eisenstein case), or as a sum of powers of the p times polynomials in the generator (in the unramified case).
  - If lift_mode == 'simple', all integers will be in the interval $[0, p-1]$.
  - If lift_mode == 'smallest' they will be in the interval $[(1-p)/2, p/2]$.
- If lift_mode == 'teichmuller', returns a list of pAdicZZpXCRElements, all of which are Teichmuller representatives and such that self is the sum of that list times powers of the uniformizer.

Note that zeros are truncated from the returned list if self.parent() is a field, so you must use the valuation function to fully reconstruct self.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: y = W(775, 19); y
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)
sage: (y>>9).list()
[0, 1, 0, 4, 0, 2, 1, 2, 4, 1]
sage: (y>>9).list('smallest')
[0, 1, 0, -1, 0, 2, 1, 2, 0, 1]
sage: w^10 - w^12 + 2*w^14 + w^15 + 2*w^16 + w^18 + O(w^19)
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: y = 75 + 45*a + 1200*a^2; y
```

```
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^6)
sage: y.list()
[[], [0, 4], [3, 1, 3], [0, 0, 4], [0, 0, 1]]
sage: y.list('smallest')
[[], [0, -1], [-2, 2, -2], [1], [0, 0, 2]]
sage: 5*((-2*5 + 25) + (-1 + 2*5)*a + (-2*5 + 2*125)*a^2)
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^6)
sage: W(0).list()
[]
sage: W(0,4).list()
[0]
sage: A(0,4).list()
[[]]
```

log()

Compute the p-adic logarithm of any unit. (See below for normalization.)

INPUT:

- branch – pAdicZZpXFMElement (default None). The log of the uniformizer. If None, then an error is raised if self is not a unit.
- same_ring – bool or pAdicGeneric (default True). When $e > p$ it is possible (even common) that the image of the log map is not contained in the ring of integers. If same_ring is True, then this function will return a value in self.parent() or raise an error if the answer would have negative valuation. If same_ring is False, then this function will raise an error (this behavior is for consistency with other p-adic types). If same_ring is a p-adic field into which this fixed mod ring can be successfully cast, then self is cast into that field and the log is taken there. Note that this casting will assume that self has the full precision possible.

OUTPUT:

- The p-adic log of self.

Let K be the parent of self, π be a uniformizer of K and w be a generator for the group of roots of unity in K. The usual power series for log with values in the additive group of K only converges for 1-units (units congruent to 1 modulo π). However, there is a unique extension of log to a homomorphism defined on all the units. If $u = av$ is a unit with $v \equiv 1 \pmod{p}$, then we define $log(u) = log(v)$. This is the correct extension because the units U of K split as a product $U = V \times < w >$, where V is the subgroup of 1-units. The $< w >$ factor is torsion, so must go to 0 under any homomorphism to the torsion free group $(K, +)$.

NOTES:

What some other systems do with regard to non-1-units:

- PARI: Seems to define log the same way as we do.
- MAGMA: Gives an error when unit is not a 1-unit.

In addition, if branch is specified, then the log map will work on non-units.

..math

```
log(pi^k \cdot u) = k \cdot branch + log(u)
```

ALGORITHM:

Input: Some unit u.

1. Check that the input is really a unit (i.e., valuation 0), or that branch is specified.
2. Let $1 - x = u^{q-1}$, which is a 1-unit, where q is the order of the residue field of K.
3. Use the series expansion

..math

```
\log(1-x) = F(x) = -x - 1/2*x^2 - 1/3*x^3 - 1/4*x^4 - 1/5*x^5 - \cdots
```

to compute the logarithm $log(u^{q-1})$.
Add on terms until x^k is zero modulo the precision cap, and then determine if there are further terms that contribute to the sum (those where k is slightly above the precision cap but divisible by p).

1.Then

..math

```
\log(u) = log(u^{q-1})/(q-1) = F(1-u^{q-1})/(q-1).``
```

EXAMPLES:
First, the Eisenstein case.:

```
sage: R = ZpCR(5,5)
sage: S.<x> = R[]
sage: f = x^4 + 15*x^2 + 625*x - 5
sage: W.<w> = R.ext(f)
sage: z = 1 + w^2 + 4*w^7; z
1 + w^2 + 4*w^7 + O(w^20)
```

Log is actually not implemented completely yet in this case!:

```
sage: z.log()   # not tested -- what we wish would happen
4*w^2 + 3*w^4 + w^6 + w^7 + w^8 + 4*w^9 + 3*w^10 + w^12 + w^13 + 3*w^14 + w^15 + 4*w^16 + 4*
sage: z.log()   # what does happen
...
NotImplementedError: log is not quite working yet
```

Check that log is multiplicative:

```
sage: y = 1 + 3*w^4 + w^5
sage: y.log() + z.log() - (y*z).log()  # not tested -- what we wish would happen
O(w^20)
sage: y.log() + z.log() - (y*z).log()  # what does happen
...
NotImplementedError: log is not quite working yet
```

Now an unramified example.:

```
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: b = 1 + 5*(1 + a^2) + 5^3*(3 + 2*a)
sage: b.log()          # not tested -- what should happen
(4*a^2 + 4)*5 + (a^2 + a + 2)*5^2 + (a^2 + 2*a + 4)*5^3 + (a^2 + 2*a + 2)*5^4 + O(5^5)
sage: b.log()          # what does happen
...
NotImplementedError: log is not quite working yet
```

Check that log is multiplicative:

```
sage: c = 3 + 5^2*(2 + 4*a)
sage: b.log() + c.log() - (b*c).log()   # not tested -- what should happen
O(5^5)
sage: b.log() + c.log() - (b*c).log()   # what *actually* does happen
...
NotImplementedError: log is not quite working yet
```

AUTHORS:

•David Roe: initial version

TODO:

•Currently implemented as $O(N^2)$. This can be improved to soft-$O(N)$ using algorithm described by Dan Bernstein: http://cr.yp.to/lineartime/multapps-20041007.pdf

matrix_mod_pn()

Returns the matrix of right multiplication by the element on the power basis $1, x, x^2, \ldots, x^{d-1}$ for this extension field. Thus the *rows* of this matrix give the images of each of the x^i. The entries of the matrices are IntegerMod elements, defined modulo p^(self.absprec() / e).

Raises an error if self has negative valuation.

EXAMPLES:

```
sage: R = ZpCR(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = (3+w)^7
sage: a.matrix_mod_pn()
[2757  333 1068  725 2510]
[  50 1507  483  318  725]
[ 500   50 3007 2358  318]
[1590 1375 1695 1032 2358]
[2415  590 2370 2970 1032]
```

precision_absolute()

Returns the absolute precision of self, ie the power of the uniformizer modulo which this element is defined.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
sage: (a.unit_part() - 3).precision_absolute()
9
```

precision_relative()

Returns the relative precision of self, ie the power of the uniformizer modulo which the unit part of self is defined.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
```

```
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
```

unit_part()
Returns the unit part of `self`, ie `self / uniformizer^(self.valuation())`
EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
```

27.21 p-Adic `ZZ_pX` CA Element.

This file implements elements of eisenstein and unramified extensions of `Zp` with capped absolute precision.

For the parent class see padic_extension_leaves.pyx.

The underlying implementation is through NTL's `ZZ_pX` class. Each element contains the following data:

- `absprec` (long) – An integer giving the precision to which this element is defined. This is the power of the uniformizer modulo which the element is well defined.
- `value` (`ZZ_pX_c`) – An ntl `ZZ_pX` storing the value. The varible x is the uniformizer in the case of eisenstein extensions. This ZZ_pX is created with global ntl modulus determined by absprec. Let a be absprec and e be the ramification index over $\mathbb{Q}_p$ or $\mathbb{Z}_p$. Then the modulus is given by $p^{ceil(a/e)}$. Note that all kinds of problems arise if you try to mix moduli. `ZZ_pX_conv_modulus` gives a semi-safe way to convert between different moduli without having to pass through ZZX (see `sage/libs/ntl/decl.pxi` and `c_lib/src/ntl_wrap.cpp`)
- `prime_pow` (some subclass of `PowComputer_ZZ_pX`) – a class, identical among all elements with the same parent, holding common data.
  - `prime_pow.deg` – The degree of the extension
  - `prime_pow.e` – The ramification index
  - `prime_pow.f` – The inertia degree
  - `prime_pow.prec_cap` – the unramified precision cap. For eisenstein extensions this is the smallest power of p that is zero.
  - `prime_pow.ram_prec_cap` – the ramified precision cap. For eisenstein extensions this will be the smallest power of x that is indistinugishable from zero.
  - `prime_pow.pow_ZZ_tmp`, prime_pow.pow_mpz_t_tmp‘‘, `prime_pow.pow_Integer` – functions for accessing powers of p. The first two return pointers. See `sage/rings/padics/pow_computer_ext` for examples and important warnings.

- prime_pow.get_context, prime_pow.get_context_capdiv, prime_pow.get_top_context – obtain an ntl_ZZ_pContext_class corresponding to p^n. The capdiv version divides by prime_pow.e as appropriate. top_context corresponds to p^{prec_cap}.
- prime_pow.restore_context, prime_pow.restore_context_capdiv, prime_pow.restore_top_context – restores the given context.
- prime_pow.get_modulus, get_modulus_capdiv, get_top_modulus – Returns a ZZ_pX_Modulus_c* pointing to a polynomial modulus defined modulo p^n (appropriately divided by prime_pow.e in the capdiv case).

EXAMPLES:

An eisenstein extension:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f); W
Eisenstein Extension of 5-adic Ring with capped absolute precision 5 in w defined by (1 + O(5^5))*x^5
sage: z = (1+w)^5; z
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 + 4*w^17 + 4*w^20 +
sage: y = z >> 1; y
w^4 + w^5 + 2*w^6 + 4*w^7 + 3*w^9 + w^11 + 4*w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 + 4*w^19 + w^20
sage: y.valuation()
4
sage: y.precision_relative()
20
sage: y.precision_absolute()
24
sage: z - (y << 1)
1 + O(w^25)
sage: (1/w)^12+w
w^-12 + w + O(w^12)
sage: (1/w).parent()
Eisenstein Extension of 5-adic Field with capped relative precision 5 in w defined by (1 + O(5^5))*x^
```

An unramified extension:

```
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: z = (1+a)^5; z
(2*a^2 + 4*a) + (3*a^2 + 3*a + 1)*5 + (4*a^2 + 3*a + 4)*5^2 + (4*a^2 + 4*a + 4)*5^3 + (4*a^2 + 4*a +
sage: z - 1 - 5*a - 10*a^2 - 10*a^3 - 5*a^4 - a^5
O(5^5)
sage: y = z >> 1; y
(3*a^2 + 3*a + 1) + (4*a^2 + 3*a + 4)*5 + (4*a^2 + 4*a + 4)*5^2 + (4*a^2 + 4*a + 4)*5^3 + O(5^4)
sage: 1/a
(3*a^2 + 4) + (a^2 + 4)*5 + (3*a^2 + 4)*5^2 + (a^2 + 4)*5^3 + (3*a^2 + 4)*5^4 + O(5^5)
```

Different printing modes:

```
sage: R = ZpCA(5, print_mode='digits'); S.<x> = ZZ[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5; W.<w> = R
sage: z = (1+w)^5; repr(z)
'...411040311321031044222131124200011101120110200202330321433201121440323201314400140044441030421100
sage: R = ZpCA(5, print_mode='bars'); S.<x> = ZZ[]; g = x^3 + 3*x + 3; A.<a> = R.ext(g)
sage: z = (1+a)^5; repr(z)
```

```
'...[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4,
sage: R = ZpCA(5, print_mode='terse'); S.<x> = ZZ[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5; W.<w> = R
sage: z = (1+w)^5; z
6 + 95367431640505*w + 25*w^2 + 95367431640560*w^3 + 5*w^4 + O(w^100)
sage: R = ZpCA(5, print_mode='val-unit'); S.<x> = ZZ[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5; W.<w> =
sage: y = (1+w)^5 - 1; y
w^5 * (2090041 + 19073486126901*w + 1258902*w^2 + 674*w^3 + 16785*w^4) + O(w^100)
```

You can get at the underlying ntl representation:

```
sage: z._ntl_rep()
[6 95367431640505 25 95367431640560 5]
sage: y._ntl_rep()
[5 95367431640505 25 95367431640560 5]
sage: y._ntl_rep_abs()
([5 95367431640505 25 95367431640560 5], 0)
```

NOTES:

```
If you get an error 'internal error: can't grow this
_ntl_gbigint,' it indicates that moduli are being mixed
inappropriately somewhere.

For example, when calling a function with a ZZ_pX_c as an
argument, it copies.  If the modulus is not set to the modulus of
the ZZ_pX_c, you can get errors.
```

AUTHORS:

- David Roe (2008-01-01) initial version

make_ZZpXCAElement()

For pickling. Makes a `pAdicZZpXCAElement` with given `parent`, `value`, `absprec`.

EXAMPLES:

```
sage: from sage.rings.padics.padic_ZZ_pX_CA_element import make_ZZpXCAElement
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: make_ZZpXCAElement(W, ntl.ZZ_pX([3,2,4],5^3),13,0)
3 + 2*w + 4*w^2 + O(w^13)
```

class pAdicZZpXCAElement()

copy()

Returns a copy of `self`.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: b = W(45, 17); b
4*w^5 + 3*w^7 + w^9 + w^10 + 2*w^11 + w^12 + w^13 + 3*w^14 + w^16 + O(w^17)
```

```
sage: c = b.copy(); c
4*w^5 + 3*w^7 + w^9 + w^10 + 2*w^11 + w^12 + w^13 + 3*w^14 + w^16 + O(w^17)
sage: c is b
False
```

is_equal_to()

Returns whether `self` is equal to `right` modulo `self.uniformizer()^absprec`.

If `absprec` is `None`, returns if `self` is equal to `right` modulo the lower of their two precisions.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(47); b = W(47 + 25)
sage: a.is_equal_to(b)
False
sage: a.is_equal_to(b, 7)
True
```

is_zero()

Returns whether the valuation of `self` is at least `absprec`. If `absprec` is `None`, returns if `self` is indistinugishable from zero.

If `self` is an inexact zero of valuation less than `absprec`, raises a PrecisionError.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: O(w^189).is_zero()
True
sage: W(0).is_zero()
True
sage: a = W(675)
sage: a.is_zero()
False
sage: a.is_zero(7)
True
sage: a.is_zero(21)
False
```

lift_to_precision()

Returns a `pAdicZZpXCAElement` congruent to `self` but with absolute precision at least `absprec`. If setting `absprec` that high would violate the precision cap, raises a precision error.

Note that the new digits will not necessarily be zero.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(345, 17); a
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15 + O(w^17)
sage: b = a.lift_to_precision(19); b # indirect doctest
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15 + w^17 + 2*w^18 + O(
sage: c = a.lift_to_precision(24); c
4*w^5 + 3*w^7 + w^9 + 3*w^10 + 2*w^11 + 4*w^12 + w^13 + 2*w^14 + 2*w^15 + w^17 + 2*w^18 + 4*
```

```
sage: a._ntl_rep()
[345]
sage: b._ntl_rep()
[345]
sage: c._ntl_rep()
[345]
```

list()

Returns a list giving a series representation of `self`.

- If `lift_mode == 'simple'` or `'smallest'`, the returned list will consist of integers (in the eisenstein case) or a list of lists of integers (in the unramified case). `self` can be reconstructed as a sum of elements of the list times powers of the uniformiser (in the eisenstein case), or as a sum of powers of p times polynomials in the generator (in the unramified case).
  - If `lift_mode == 'simple'`, all integers will be in the interval $[0, p-1]$
  - If `lift_mod == 'smallest'` they will be in the interval $[(1-p)/2, p/2]$.
- If `lift_mode == 'teichmuller'`, returns a list of `pAdicZZpXCAElements`, all of which are Teichmuller representatives and such that `self` is the sum of that list times powers of the uniformizer.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: y = W(775, 19); y
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)
sage: (y>>9).list()
[0, 1, 0, 4, 0, 2, 1, 2, 4, 1]
sage: (y>>9).list('smallest')
[0, 1, 0, -1, 0, 2, 1, 2, 0, 1]
sage: w^10 - w^12 + 2*w^14 + w^15 + 2*w^16 + w^18 + O(w^19)
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + O(w^19)
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: y = 75 + 45*a + 1200*a^2; y
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^5)
sage: y.list()
[[], [0, 4], [3, 1, 3], [0, 0, 4], [0, 0, 1]]
sage: y.list('smallest')
[[], [0, -1], [-2, 2, -2], [1], [0, 0, 2]]
sage: 5*((-2*5 + 25) + (-1 + 2*5)*a + (-2*5 + 2*125)*a^2)
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^5)
sage: W(0).list()
[0]
sage: A(0,4).list()
[[]]
```

matrix_mod_pn()

Returns the matrix of right multiplication by the element on the power basis $1, x, x^2, \ldots, x^{d-1}$ for this extension field. Thus the *rows* of this matrix give the images of each of the x^i. The entries of the matrices are `IntegerMod` elements, defined modulo `p^(self.absprec() / e)`.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
```

```
sage: W.<w> = R.ext(f)
sage: a = (3+w)^7
sage: a.matrix_mod_pn()
[2757  333 1068  725 2510]
[  50 1507  483  318  725]
[ 500   50 3007 2358  318]
[1590 1375 1695 1032 2358]
[2415  590 2370 2970 1032]
```

precision_absolute()

Returns the absolute precision of `self`, ie the power of the uniformizer modulo which this element is defined.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
```

precision_relative()

Returns the relative precision of `self`, ie the power of the uniformizer modulo which the unit part of `self` is defined.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
```

to_fraction_field()

Returns `self` cast into the fraction field of `self.parent()`.

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: z = (1 + w)^5; z
```

```
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 + 4*w^17 +
sage: y = z.to_fraction_field(); y #indirect doctest
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 + 4*w^17 +
sage: y.parent()
Eisenstein Extension of 5-adic Field with capped relative precision 5 in w defined by (1 + C
```

unit_part()

Returns the unit part of `self`, ie `self / uniformizer^(self.valuation())`

EXAMPLES:

```
sage: R = ZpCA(5,5)
sage: S.<x> = ZZ[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75, 19); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + O(w^19)
sage: a.valuation()
10
sage: a.precision_absolute()
19
sage: a.precision_relative()
9
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + O(w^9)
```

27.22 p-Adic `ZZ_pX` FM Element.

This file implements elements of eisenstein and unramified extensions of $\mathbb{Z}_p$ with fixed modulus precision.

For the parent class see `padic_extension_leaves.pyx`.

The underlying implementation is through NTL's `ZZ_pX` class. Each element contains the following data:

- `value` (`ZZ_pX_c`) – An ntl `ZZ_pX` storing the value. The varible x is the uniformizer in the case of eisenstein extensions. This `ZZ_pX` is created with global ntl modulus determined by the parent's precision cap and shared among all elements.
- `prime_pow` (some subclass of `PowComputer_ZZ_pX`) – a class, identical among all elements with the same parent, holding common data.
  - `prime_pow.deg` – The degree of the extension
  - `prime_pow.e` – The ramification index
  - `prime_pow.f` – The inertia degree
  - `prime_pow.prec_cap` – the unramified precision cap. For eisenstein extensions this is the smallest power of p that is zero.
  - `prime_pow.ram_prec_cap` – the ramified precision cap. For eisenstein extensions this will be the smallest power of x that is indistinugishable from zero.
  - `prime_pow.pow_ZZ_tmp`, prime_pow.pow_mpz_t_tmp'', `prime_pow.pow_Integer` – functions for accessing powers of p. The first two return pointers. See `sage/rings/padics/pow_computer_ext` for examples and important warnings.
  - `prime_pow.get_context`, `prime_pow.get_context_capdiv`, `prime_pow.get_top_context` – obtain an `ntl_ZZ_pContext_class` corresponding to p^n. The capdiv version divides by `prime_pow.e` as appropriate. `top_context` corresponds to p^{prec_cap}.

- prime_pow.restore_context, prime_pow.restore_context_capdiv, prime_pow.restore_top_context – restores the given context.
- prime_pow.get_modulus, get_modulus_capdiv, get_top_modulus – Returns a ZZ_pX_Modulus_c* pointing to a polynomial modulus defined modulo p^n (appropriately divided by prime_pow.e in the capdiv case).

EXAMPLES:

An eisenstein extension:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f); W
Eisenstein Extension of 5-adic Ring of fixed modulus 5^5 in w defined by (1 + O(5^5))*x^5 + (3*5^2 +
sage: z = (1+w)^5; z
1 + w^5 + w^6 + 2*w^7 + 4*w^8 + 3*w^10 + w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 + 4*w^17 + 4*w^20 +
sage: y = z >> 1; y
w^4 + w^5 + 2*w^6 + 4*w^7 + 3*w^9 + w^11 + 4*w^12 + 4*w^13 + 4*w^14 + 4*w^15 + 4*w^16 + 4*w^19 + w^20
sage: y.valuation()
4
sage: y.precision_relative()
21
sage: y.precision_absolute()
25
sage: z - (y << 1)
1 + O(w^25)
```

An unramified extension:

```
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: z = (1+a)^5; z
(2*a^2 + 4*a) + (3*a^2 + 3*a + 1)*5 + (4*a^2 + 3*a + 4)*5^2 + (4*a^2 + 4*a + 4)*5^3 + (4*a^2 + 4*a +
sage: z - 1 - 5*a - 10*a^2 - 10*a^3 - 5*a^4 - a^5
O(5^5)
sage: y = z >> 1; y
(3*a^2 + 3*a + 1) + (4*a^2 + 3*a + 4)*5 + (4*a^2 + 4*a + 4)*5^2 + (4*a^2 + 4*a + 4)*5^3 + O(5^5)
sage: 1/a
(3*a^2 + 4) + (a^2 + 4)*5 + (3*a^2 + 4)*5^2 + (a^2 + 4)*5^3 + (3*a^2 + 4)*5^4 + O(5^5)
```

Different printing modes:

```
sage: R = ZpFM(5, print_mode='digits'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5; W.<w> = R
sage: z = (1+w)^5; repr(z)
'...411040311321031044222131124200011101120110200202330321433201121440323201314400140044441030421100
sage: R = ZpFM(5, print_mode='bars'); S.<x> = R[]; g = x^3 + 3*x + 3; A.<a> = R.ext(g)
sage: z = (1+a)^5; repr(z)
'...[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4, 4]|[4, 4,
sage: R = ZpFM(5, print_mode='terse'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5; W.<w> = R.e
sage: z = (1+w)^5; z
6 + 95367431640505*w + 25*w^2 + 95367431640560*w^3 + 5*w^4 + O(w^100)
sage: R = ZpFM(5, print_mode='val-unit'); S.<x> = R[]; f = x^5 + 75*x^3 - 15*x^2 + 125*x -5; W.<w> =
sage: y = (1+w)^5 - 1; y
w^5 * (2090041 + 95367431439401*w + 76293946571402*w^2 + 57220458985049*w^3 + 57220459001160*w^4) + O
```

AUTHORS:

- David Roe (2008-01-01) initial version

make_ZZpXFMElement()

Creates a new pAdicZZpXFMElement out of an ntl_ZZ_pX f, with parent parent. For use with pickling.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: z = (1 + w)^5 - 1
sage: loads(dumps(z)) == z # indirect doctest
True
```

class **pAdicZZpXFMElement**()

copy()

Returns a copy of self.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: b = W(45); b
4*w^5 + 3*w^7 + w^9 + w^10 + 2*w^11 + w^12 + w^13 + 3*w^14 + w^16 + 2*w^17 + w^19 + 4*w^20 +
sage: c = b.copy(); c
4*w^5 + 3*w^7 + w^9 + w^10 + 2*w^11 + w^12 + w^13 + 3*w^14 + w^16 + 2*w^17 + w^19 + 4*w^20 +
sage: c is b
False
```

is_equal_to()

Returns whether self is equal to right modulo self.uniformizer()^absprec.

If absprec is None, returns if self is equal to right modulo the precision cap.

EXAMPLES:

```
sage: R = Zp(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(47); b = W(47 + 25)
sage: a.is_equal_to(b)
False
sage: a.is_equal_to(b, 7)
True
```

is_zero()

Returns whether the valuation of self is at least absprec. If absprec is None, returns whether self is indistinugishable from zero.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: O(w^189).is_zero()
True
```

```
sage: W(0).is_zero()
True
sage: a = W(675)
sage: a.is_zero()
False
sage: a.is_zero(7)
True
sage: a.is_zero(21)
False
```

lift_to_precision()

Returns `self`.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: w.lift_to_precision(10000)
w + O(w^25)
```

list()

Returns a list giving a series representation of `self`.

- If `lift_mode == 'simple' or 'smallest'`, the returned list will consist of
  - integers (in the eisenstein case) or
  - lists of integers (in the unramified case).
- `self` can be reconstructed as
  - a sum of elements of the list times powers of the uniformiser (in the eisenstein case), or
  - as a sum of powers of the p times polynomials in the generator (in the unramified case).
- If `lift_mode == 'simple'`, all integers will be in the range $[0, p-1]$,
- If `lift_mode == 'smallest'` they will be in the range $[(1-p)/2, p/2]$.
- If `lift_mode == 'teichmuller'`, returns a list of `pAdicZZpXCRElements`, all of which are Teichmuller representatives and such that `self` is the sum of that list times powers of the uniformizer.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: y = W(775); y
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + w^20 + 2*w^21 + 3*w^22 + w^23 + w^2
sage: (y>>9).list()
[0, 1, 0, 4, 0, 2, 1, 2, 4, 1, 0, 1, 2, 3, 1, 1, 4, 1, 2, 4, 1, 0, 4, 3]
sage: (y>>9).list('smallest')
[0, 1, 0, -1, 0, 2, 1, 2, 0, 1, 2, 1, 1, -1, -1, 2, -2, 0, -2, -2, -2, 0, 2, -2, 2]
sage: w^10 - w^12 + 2*w^14 + w^15 + 2*w^16 + w^18 + 2*w^19 + w^20 + w^21 - w^22 - w^23 + 2*w
w^10 + 4*w^12 + 2*w^14 + w^15 + 2*w^16 + 4*w^17 + w^18 + w^20 + 2*w^21 + 3*w^22 + w^23 + w^2
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: y = 75 + 45*a + 1200*a^2; y
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^5)
sage: y.list()
[[], [0, 4], [3, 1, 3], [0, 0, 4], [0, 0, 1]]
sage: y.list('smallest')
[[], [0, -1], [-2, 2, -2], [1], [0, 0, 2]]
```

```
sage: 5*((-2*5 + 25) + (-1 + 2*5)*a + (-2*5 + 2*125)*a^2)
4*a*5 + (3*a^2 + a + 3)*5^2 + 4*a^2*5^3 + a^2*5^4 + O(5^5)
sage: W(0).list()
[0]
sage: A(0,4).list()
[[]]
```

log()

Compute the p-adic logarithm of any unit.

See below for normalization.

INPUT:

- branch – pAdicZZpXFMElement (default None). The log of the uniformizer. If None, then an error is raised if self is not a unit.
- same_ring – bool or pAdicGeneric (default True). When $e > p$ it is possible (even common) that the image of the log map is not contained in the ring of integers. If same_ring is True, then this function will return a value in self.parent() or raise an error if the answer would have negative valuation. If same_ring is False, then this function will raise an error (this behavior is for consistency with other p-adic types). If same_ring is a p-adic field into which this fixed mod ring can be successfully cast, then self is cast into that field and the log is taken there. Note that this casting will assume that self has the full precision possible.

OUTPUT:

The p-adic log of self.

Let K be the parent of self, π be a uniformizer of K and w be a generator for the group of roots of unity in K. The usual power series for log with values in the additive group of K only converges for 1-units (units congruent to 1 modulo π). However, there is a unique extension of log to a homomorphism defined on all the units. If $u = a \cdot v$ is a unit with $v = 1 \pmod{p}$, then we define $\log(u) = \log(v)$. This is the correct extension because the units U of K split as a product $U = V \times \langle w \rangle$, where V is the subgroup of 1-units. The $\langle w \rangle$ factor is torsion, so must go to 0 under any homomorphism to the torsion free group $(K, +)$.

NOTES:

What some other systems do with regard to non-1-units:

- PARI: Seems to define log the same way as we do.
- MAGMA: Gives an error when unit is not a 1-unit.

In addition, if branch is specified, then the log map will work on non-units

..math

```
\log(\pi^k \cdot u) = k \cdot branch + \log(u)
```

ALGORITHM:

Input: Some unit u.

1. Check that the input is really a unit (i.e., valuation 0), or that branch is specified.
2. Let $1 - x = u^{q-1}$, which is a 1-unit, where q is the order of the residue field of K.
3. Use the series expansion

..math

```
\log(1-x) = F(x) = -x - 1/2*x^2 - 1/3*x^3 - 1/4*x^4 - 1/5*x^5 - ...
```

to compute the logarithm $log(u^{q-1})$.

Add on terms until x^k is zero modulo the precision cap, and then determine if there are further terms that contribute to the sum (those where k is slightly above the precision cap but divisible by p).

1. Then

..math

```
\log(u) = \log(u^{q-1})/(q-1) = F(1-u^{q-1})/(q-1).
```

EXAMPLES:

First, the Eisenstein case.:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^4 + 15*x^2 + 625*x - 5
sage: W.<w> = R.ext(f)
sage: z = 1 + w^2 + 4*w^7; z
1 + w^2 + 4*w^7 + O(w^20)
sage: z.log()
4*w^2 + 3*w^4 + w^6 + w^7 + w^8 + 4*w^9 + 3*w^10 + w^12 + w^13 + 3*w^14 + w^15 + 4*w^16 + 4*
```

Check that log is multiplicative:

```
sage: y = 1 + 3*w^4 + w^5
sage: y.log() + z.log() - (y*z).log()
O(w^20)
```

Now an unramified example.:

```
sage: g = x^3 + 3*x + 3
sage: A.<a> = R.ext(g)
sage: b = 1 + 5*(1 + a^2) + 5^3*(3 + 2*a)
sage: b.log()
(4*a^2 + 4)*5 + (a^2 + a + 2)*5^2 + (a^2 + 2*a + 4)*5^3 + (a^2 + 2*a + 2)*5^4 + O(5^5)
```

Check that log is multiplicative:

```
sage: c = 3 + 5^2*(2 + 4*a)
sage: b.log() + c.log() - (b*c).log()
O(5^5)
```

AUTHORS:

- David Roe: initial version

TODO:

- Currently implemented as $O(N^2)$. This can be improved to soft-$O(N)$ using algorithm described by Dan Bernstein: http://cr.yp.to/lineartime/multapps-20041007.pdf

matrix_mod_pn()

Returns the matrix of right multiplication by the element on the power basis $1, x, x^2, \ldots, x^{d-1}$ for this extension field. Thus the emph{rows} of this matrix give the images of each of the x^i. The entries of the matrices are `IntegerMod` elements, defined modulo `p^(self.absprec() / e)`.

Raises an error if self has negative valuation.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = (3+w)^7
sage: a.matrix_mod_pn()
[2757  333 1068  725 2510]
[  50 1507  483  318  725]
[ 500   50 3007 2358  318]
[1590 1375 1695 1032 2358]
[2415  590 2370 2970 1032]
```

norm()

Return the absolute or relative norm of this element.

NOTE! This is not the p-adic absolute value. This is a field theoretic norm down to a ground ring.

If you want the p-adic absolute value, use the `abs()` function instead.

If K is given then K must be a subfield of the parent L of `self`, in which case the norm is the relative norm from L to K. In all other cases, the norm is the absolute norm down to $\mathbb{Q}_p$ or $\mathbb{Z}_p$.

EXAMPLES:

```
sage: R = ZpCR(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: ((1+2*w)^5).norm()
1 + 5^2 + O(5^5)
sage: ((1+2*w)).norm()^5
1 + 5^2 + O(5^5)
```

precision_absolute()

Returns the absolute precision of `self`, ie the precision cap of `self.parent()`.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + 3*w^19 + 2*w^21 + 3*w^22 + 3*w^23 + O(w^25)
sage: a.valuation()
10
sage: a.precision_absolute()
25
sage: a.precision_relative()
15
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + 3*w^9 + 2*w^11 + 3*w^12 + 3*w^13 + w^15 + 4*w^16 + 2*w
```

precision_relative()

Returns the relative precision of `self`, ie the precision cap of `self.parent()` minus the `valuation of self`.

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + 3*w^19 + 2*w^21 + 3*w^22 + 3*w^23 + O(w^25)
sage: a.valuation()
10
sage: a.precision_absolute()
25
sage: a.precision_relative()
15
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + 3*w^9 + 2*w^11 + 3*w^12 + 3*w^13 + w^15 + 4*w^16 + 2*w
```

trace()

Return the absolute or relative trace of this element.

If K is given then K must be a subfield of the parent L of `self`, in which case the norm is the relative norm from L to K. In all other cases, the norm is the absolute norm down to $\mathbb{Q}_p$ or $\mathbb{Z}_p$.

EXAMPLES:

```
sage: R = ZpCR(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = (2+3*w)^7
sage: b = (6+w^3)^5
sage: a.trace()
3*5 + 2*5^2 + 3*5^3 + 2*5^4 + O(5^5)
sage: a.trace() + b.trace()
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
sage: (a+b).trace()
4*5 + 5^2 + 5^3 + 2*5^4 + O(5^5)
```

unit_part()

Returns the unit part of `self`, ie `self / uniformizer^(self.valuation())`

EXAMPLES:

```
sage: R = ZpFM(5,5)
sage: S.<x> = R[]
sage: f = x^5 + 75*x^3 - 15*x^2 +125*x - 5
sage: W.<w> = R.ext(f)
sage: a = W(75); a
3*w^10 + 2*w^12 + w^14 + w^16 + w^17 + 3*w^18 + 3*w^19 + 2*w^21 + 3*w^22 + 3*w^23 + O(w^25)
sage: a.valuation()
10
sage: a.precision_absolute()
25
sage: a.precision_relative()
15
sage: a.unit_part()
3 + 2*w^2 + w^4 + w^6 + w^7 + 3*w^8 + 3*w^9 + 2*w^11 + 3*w^12 + 3*w^13 + w^15 + 4*w^16 + 2*w
```

27.23 PowComputer.

A class for computing and caching powers of the same integer.

This class is designed to be used as a field of p-adic rings and fields. Since elements of p-adic rings and fields need to use powers of p over and over, this class precomputes and stores powers of p. There is no reason that the base has to be prime however.

EXAMPLES:

```
sage: X = PowComputer(3, 4, 10)
sage: X(3)
27
sage: X(10) == 3^10
True
```

AUTHORS:

- David Roe

PowComputer()

Returns a PowComputer that caches the values $1, m, m^2, ldots, m^cache_limit$.

Once you create a PowComputer, merely call it to get values out.

You can input any integer, even if it's outside of the precomputed range.

INPUT:

```
* m -- An integer, the base that you want to exponentiate.
* cache_limit -- A positive integer that you want to cache powers up to.
```

EXAMPLES:

```
sage: PC = PowComputer(3, 5, 10)
sage: PC
PowComputer for 3
sage: PC(4)
81
sage: PC(6)
729
sage: PC(-1)
1/3
```

class **PowComputer_base**()

class **PowComputer_class**()

pow_Integer_Integer()

Tests the pow_Integer function.

EXAMPLES:

```
sage: PC = PowComputer(3, 5, 10)
sage: PC.pow_Integer_Integer(4)
81
sage: PC.pow_Integer_Integer(6)
729
sage: PC.pow_Integer_Integer(0)
1
sage: PC.pow_Integer_Integer(10)
59049
sage: PC = PowComputer_ext_maker(3, 5, 10, 20, False, ntl.ZZ_pX([-3,0,1], 3^10), 'big','e',r
sage: PC.pow_Integer_Integer(4)
81
sage: PC.pow_Integer_Integer(6)
729
sage: PC.pow_Integer_Integer(0)
1
sage: PC.pow_Integer_Integer(10)
59049
```

clear_mpz_globals()

gmp_randrange()

init_mpz_globals()

27.24 PowComputer_ext.

The classes in this file are designed to be attached to p-adic parents and elements for cython access to properties of the parent.

In addition to storing the defining polynomial (as an NTL polynomial) at different precisions, they also cache powers of p and data to speed right shifting of elements.

The heirarchy of PowComputers splits first at whether it's for a base ring (Qp or Zp) or an extension.

Among the extension classes (those in this file), they are first split by the type of NTL polynomial (ntl_ZZ_pX or ntl_ZZ_pEX), then by the amount and style of caching (see below). Finally, there are subclasses of the ntl_ZZ_pX PowComputers that cache additional information for Eisenstein extensions.

There are three styles of caching:

- FM: caches powers of p up to the cache_limit, only caches the polynomial modulus and the ntl_ZZ_pContext of precision prec_cap.
- small: Requires cache_limit = prec_cap. Caches p^k for every k up to the cache_limit and caches a polynomial modulus and a ntl_ZZ_pContext for each such power of p.
- big: Caches as the small does up to cache_limit and caches prec_cap. Also has a dictionary that caches values above the cache_limit when they are computed (rather than at ring creation time).

AUTHORS:

- David Roe (2008-01-01) initial version

class PowComputer_ZZ_pX()

polynomial()

Returns the polynomial (with coefficient precision prec_cap) associated to this PowComputer.

The polynomial is output as an ntl_ZZ_pX.

EXAMPLES:

```
sage: PC = PowComputer_ext_maker(5, 5, 10, 20, False, ntl.ZZ_pX([-5,0,1],5^10), 'FM', 'e',nt
sage: PC.polynomial()
[9765620 0 1]
```

speed_test()

Runs a speed test.

INPUT:

```
n -- input to a function to be tested (the function needs to be set in the source code).
runs -- The number of runs of that function
```

OUTPUT:

```
The time in seconds that it takes to call the function on n, runs times.

sage: PC = PowComputer_ext_maker(5, 10, 10, 20, False, ntl.ZZ_pX([-5, 0, 1], 5^10), 'small',
sage: PC.speed_test(10, 10^6) # random
0.0090679999999991878
```

class PowComputer_ZZ_pX_FM()

This class only caches a context and modulus for p^prec_cap.

Designed for use with fixed modulus p-adic rings, in Eisenstein and unramified extensions of $mathbb{Z}_p$.

class PowComputer_ZZ_pX_FM_Eis()

This class computes and stores low_shifter and high_shifter, which aid in right shifting elements.

class PowComputer_ZZ_pX_big()

This class caches all contexts and moduli between 1 and cache_limit, and also caches for prec_cap. In addition, it stores a dictionary of contexts and moduli of

reset_dictionaries()

Resets the dictionaries. Note that if there are elements lying around that need access to these dictionaries, calling this function and then doing arithmetic with those elements could cause trouble (if the context object gets garbage collected for example. The bugs introduced could be very subtle, because NTL will generate a new context object and use it, but there's the potential for the object to be incompatible with the different context object).

EXAMPLES:

```
sage: A = PowComputer_ext_maker(5, 6, 10, 20, False, ntl.ZZ_pX([-5,0,1],5^10), 'big','e',ntl
sage: P = A._get_context_test(8)
sage: A._context_dict()
{8: NTL modulus 390625}
sage: A.reset_dictionaries()
sage: A._context_dict()
{}
```

class PowComputer_ZZ_pX_big_Eis()

This class computes and stores low_shifter and high_shifter, which aid in right shifting elements. These are only stored at maximal precision: in order to get lower precision versions just reduce mod p^n.

class PowComputer_ZZ_pX_small()

This class caches contexts and moduli densely between 1 and cache_limit. It requires cache_limit == prec_cap.

It is intended for use with capped relative and capped absolute rings and fields, in Eisenstein and unramified extensions of the base p-adic fields.

class PowComputer_ZZ_pX_small_Eis()

This class computes and stores low_shifter and high_shifter, which aid in right shifting elements. These are only stored at maximal precision: in order to get lower precision versions just reduce mod p^n.

class PowComputer_ext()

PowComputer_ext_maker()

Returns a PowComputer that caches the values $1, prime, prime^2, ldots, prime^cache_limit$.

Once you create a PowComputer, merely call it to get values out. You can input any integer, even if it's outside of the precomputed range.

INPUT:

```
- prime -- An integer, the base that you want to exponentiate.

- cache_limit -- A positive integer that you want to cache
  powers up to.

- prec_cap -- The cap on precisions of elements.  For ramified
  extensions, p^((prec_cap - 1) // e) will be the largest
  power of p distinguishable from zero

- in_field -- Boolean indicating whether this PowComputer is
  attached to a field or not.

- poly -- An ntl_ZZ_pX or ntl_ZZ_pEX defining the extension.
  It should be defined modulo p^((prec_cap - 1) // e + 1)

- prec_type -- 'FM', 'small', or 'big', defining how caching
  is done.
```

```
- ext_type -- 'u' = unramified, 'e' = eisenstein, 't' =
  two-step

- shift_seed -- (required only for eisenstein and two-step)
  For eisenstein and two-step extensions, if f = a_n x^n - p
  a_{n-1} x^{n-1} - ... - p a_0 with a_n a unit, then
  shift_seed should be 1/a_n (a_{n-1} x^{n-1} + ... + a_0)
```

EXAMPLES:

```
sage: PC = PowComputer_ext_maker(5, 10, 10, 20, False, ntl.ZZ_pX([-5, 0, 1], 5^10), 'small','e',
sage: PC
PowComputer_ext for 5, with polynomial [9765620 0 1]
```

ZZ_pX_eis_shift_test()

Shifts _a right _n x-adic digits, where x is considered modulo the polynomial in _shifter.

EXAMPLES:

```
sage: from sage.rings.padics.pow_computer_ext import ZZ_pX_eis_shift_test
sage: A = PowComputer_ext_maker(5, 3, 10, 40, False, ntl.ZZ_pX([-5,75,15,0,1],5^10), 'big', 'e',
sage: ZZ_pX_eis_shift_test(A, [0, 1], 1, 5)
[1]
sage: ZZ_pX_eis_shift_test(A, [0, 0, 1], 1, 5)
[0 1]
sage: ZZ_pX_eis_shift_test(A, [5], 1, 5)
[75 15 0 1]
sage: ZZ_pX_eis_shift_test(A, [1], 1, 5)
[]
sage: ZZ_pX_eis_shift_test(A, [17, 91, 8, -2], 1, 5)
[316 53 3123 3]
sage: ZZ_pX_eis_shift_test(A, [316, 53, 3123, 3], -1, 5)
[15 91 8 3123]
sage: ZZ_pX_eis_shift_test(A, [15, 91, 8, 3123], 1, 5)
[316 53 3123 3]
```

clear_mpz_globals()

gmp_randrange()

init_mpz_globals()

27.25 p-Adic Printing.

This file contains code for printing p-adic elements.

It has been moved here to prevent code duplication and make finding the relevant code easier.

AUTHORS:

- David Roe

clear_mpz_globals()

gmp_randrange()

init_mpz_globals()

pAdicPrinter()
Creates a pAdicPrinter.

INPUT:

```
- ring -- a p-adic ring or field.

- options -- a dictionary, with keys in 'mode', 'pos',
  'ram_name', 'unram_name', 'var_name', 'max_ram_terms',
  'max_unram_terms', 'max_terse_terms', 'sep', 'alphabet'; see
  pAdicPrinter_class for the meanings of these keywords.
```

EXAMPLES:

```
sage: from sage.rings.padics.padic_printing import pAdicPrinter
sage: R = Zp(5)
sage: pAdicPrinter(R, {'sep': '&'})
series printer for 5-adic Ring with capped relative precision 20
```

class **pAdicPrinterDefaults**()
This class stores global defaults for p-adic printing.

allow_negatives()
Controls whether or not to display a balanced representation.

neg=None returns the current value.

EXAMPLES:

```
sage: padic_printing.allow_negatives(True)
sage: padic_printing.allow_negatives()
True
sage: Qp(29)(-1)
-1 + O(29^20)
sage: Qp(29)(-1000)
-14 - 5*29 - 29^2 + O(29^20)
sage: padic_printing.allow_negatives(False)
```

alphabet()
Controls the alphbet used to translate p-adic digits into strings (so that no separator need be used in 'digits' mode).

alphabet should be passed in as a list or tuple.

alphabet=None returns the current value.

EXAMPLES:

```
sage: padic_printing.alphabet("abc")
sage: padic_printing.mode('digits')
sage: repr(Qp(3)(1234))
'...bcaacab'

sage: padic_printing.mode('series')
sage: padic_printing.alphabet(('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','
```

max_poly_terms()
Controls the number of terms appearing when printing polynomial representations in 'terse' or 'val-unit' modes.

max=None returns the current value.

max=-1 encodes 'no limit.'

EXAMPLES:

```
sage: padic_printing.max_poly_terms(3)
sage: padic_printing.max_poly_terms()
3
sage: padic_printing.mode('terse')
sage: Zq(7^5, 5, names='a')([2,3,4])^8
2570 + 15808*a + 9018*a^2 + ... + O(7^5)

sage: padic_printing.max_poly_terms(-1)
sage: padic_printing.mode('series')
```

max_series_terms()

Controls the maximum number of terms shown when printing in 'series', 'digits' or 'bars' mode.

max=None returns the current value.

max=-1 encodes 'no limit.'

EXAMPLES:

```
sage: padic_printing.max_series_terms(2)
sage: padic_printing.max_series_terms()
2
sage: Qp(31)(1000)
8 + 31 + ... + O(31^20)
sage: padic_printing.max_series_terms(-1)
sage: Qp(37)(100000)
26 + 37 + 36*37^2 + 37^3 + O(37^20)
```

max_unram_terms()

For rings with non-prime residue fields, controls how many terms appear in the coefficient of each pi^n when printing in 'series' or 'bar' modes.

max=None returns the current value.

max=-1 encodes 'no limit.'

EXAMPLES:

```
sage: padic_printing.max_unram_terms(2)
sage: padic_printing.max_unram_terms()
2
sage: Zq(5^6, 5, names='a')([1,2,3,-1])^17
(3*a^4 + ... + 3) + (a^5 + ... + a)*5 + (3*a^3 + ... + 2)*5^2 + (3*a^5 + ... + 2)*5^3 + (4*a

sage: padic_printing.max_unram_terms(-1)
```

mode()

Set the default printing mode.

mode=None returns the current value.

The allowed values for mode are: 'val-unit', 'series', 'terse', 'digits' and 'bars'.

EXAMPLES:

```
sage: padic_printing.mode('terse')
sage: padic_printing.mode()
'terse'
sage: Qp(7)(100)
100 + O(7^20)
sage: padic_printing.mode('series')
sage: Qp(11)(100)
1 + 9*11 + O(11^20)
sage: padic_printing.mode('val-unit')
sage: Qp(13)(130)
13 * 10 + O(13^21)
```

```
sage: padic_printing.mode('digits')
sage: repr(Qp(17)(100))
'...5F'
sage: repr(Qp(17)(1000))
'...37E'
sage: padic_printing.mode('bars')
sage: repr(Qp(19)(1000))
'...2|14|12'

sage: padic_printing.mode('series')
```

sep()

Controls the separator used in 'bars' mode.

sep=None returns the current value.

EXAMPLES:

```
sage: padic_printing.sep('][')
sage: padic_printing.sep()
']['
sage: padic_printing.mode('bars')
sage: repr(Qp(61)(-1))
'...60][60][60][60][60][60][60][60][60][60][60][60][60][60][60][60][60][60][60][60'

sage: padic_printing.sep('|')
sage: padic_printing.mode('series')
```

class **pAdicPrinter_class**()

This class stores the printing options for a specific p-adic ring or field, and uses these to compute the representations of elements.

cmp_modes()

Returns a comparison of the printing modes of self and other.

Returns 0 if and only if all relevant modes are equal (max_unram_terms is irrelevant if the ring is totally ramified over the base for example). Does not check if the rings are equal (to prevent infinite recursion in the comparison functions of p-adic rings), but it does check if the primes are the same (since the prime affects whether pos is relevant).

EXAMPLES:

```
sage: R = Qp(7, print_mode='digits', print_pos=True)
sage: S = Qp(7, print_mode='digits', print_pos=False)
sage: R._printer.cmp_modes(S._printer)
0
sage: R = Qp(7)
sage: S = Qp(7,print_mode='val-unit')
sage: R == S
False
sage: R._printer.cmp_modes(S._printer)
-1
```

dict()

Returns a dictionary storing all of self's printing options.

EXAMPLES:

```
sage: D = Zp(5)._printer.dict(); D['sep']
'|'
```

repr_gen()

The entry point for printing an element.

INPUT:

- elt -- a p-adic element of the appropriate ring to print.

- do_latex -- whether to return a latex representation or a normal one.

EXAMPLES:

```
sage: R = Zp(5,5); P = R._printer; a = R(-5); a
4*5 + 4*5^2 + 4*5^3 + 4*5^4 + 4*5^5 + O(5^6)
sage: P.repr_gen(a, False, pos=False)
'-5 + O(5^6)'
sage: P.repr_gen(a, False, ram_name='p')
'4*p + 4*p^2 + 4*p^3 + 4*p^4 + 4*p^5 + O(p^6)'
```

27.26 Precision Error.

The errors in this file indicate various styles of precision problems that can go wrong for p-adics and power series.

AUTHORS:

- David Roe

exception HaltingError

exception PrecisionError

exception PrecisionLimitError

27.27 Miscellaneous Functions.

This file contains some miscellaneous functions used by p-adics.

- min – a version of min that returns ∞ on empty input.
- max – a version of max that returns $-\infty$ on empty input.

AUTHORS:

- David Roe

max(*L)

Returns the maximum of the inputs, where the maximum of the empty list is -infinity.

EXAMPLES:

```
sage: from sage.rings.padics.misc import max
sage: max()
-Infinity
sage: max(2,3)
3
```

min(*L)

Returns the minimum of the inputs, where the minimum of the empty list is infinity.

EXAMPLES:

```
sage: from sage.rings.padics.misc import min
sage: min()
+Infinity
sage: min(2,3)
2
```

CHAPTER

TWENTYEIGHT

POLYNOMIAL RINGS

28.1 Univariate Polynomial Rings

Sage implements sparse and dense polynomials over commutative and non-commutative rings. In the non-commutative case, the polynomial variable commutes with the elements of the base ring.

AUTHOR:

- William Stein
- Kiran Kedlaya (2006-02-13): added macaulay2 option
- Martin Albrecht (2006-08-25): removed it again as it isn't needed anymore

EXAMPLES: Creating a polynomial ring injects the variable into the interpreter namespace:

```
sage: z = QQ['z'].0
sage: (z^3 + z - 1)^3
z^9 + 3*z^7 - 3*z^6 + 3*z^5 - 6*z^4 + 4*z^3 - 3*z^2 + 3*z - 1
```

Saving and loading of polynomial rings works:

```
sage: loads(dumps(QQ['x'])) == QQ['x']
True
sage: k = PolynomialRing(QQ['x'],'y'); loads(dumps(k))==k
True
sage: k = PolynomialRing(ZZ,'y'); loads(dumps(k)) == k
True
sage: k = PolynomialRing(ZZ,'y', sparse=True); loads(dumps(k))
Sparse Univariate Polynomial Ring in y over Integer Ring
```

The rings of sparse and dense polynomials in the same variable are canonically isomorphic:

```
sage: PolynomialRing(ZZ,'y', sparse=True) == PolynomialRing(ZZ,'y')
True

sage: QQ['y'] < QQ['x']
False
sage: QQ['y'] < QQ['z']
True
```

We create a polynomial ring over a quaternion algebra:

```
sage: A.<i,j,k> = QuaternionAlgebra(QQ, -1,-1)
sage: R.<w> = PolynomialRing(A,sparse=True)
sage: f = w^3 + (i+j)*w + 1
sage: f
w^3 + (i + j)*w + 1
sage: f^2
w^6 + (2*i + 2*j)*w^4 + 2*w^3 - 2*w^2 + (2*i + 2*j)*w + 1
sage: f = w + i ; g = w + j
sage: f * g
w^2 + (i + j)*w + k
sage: g * f
w^2 + (i + j)*w - k
```

TESTS:

```
sage: K.<x>=FractionField(QQ['x'])
sage: V.<z> = K[]
sage: x+z
z + x
```

These may change over time:

```
sage: type(ZZ['x'].0)
<type 'sage.rings.polynomial.polynomial_integer_dense_flint.Polynomial_integer_dense_flint'>
sage: type(QQ['x'].0)
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_rational_dense'>
sage: type(RR['x'].0)
<type 'sage.rings.polynomial.polynomial_real_mpfr_dense.PolynomialRealDense'>
sage: type(Integers(4)['x'].0)
<type 'sage.rings.polynomial.polynomial_zmod_flint.Polynomial_zmod_flint'>
sage: type(Integers(5*2^100)['x'].0)
<type 'sage.rings.polynomial.polynomial_modn_dense_ntl.Polynomial_dense_modn_ntl_ZZ'>
sage: type(CC['x'].0)
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_generic_dense_field'>
sage: type(CC['t']['x'].0)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
```

class **PolynomialRing_commutative** (*base_ring, name=None, sparse=False, element_class=None*)

Univariate polynomial ring over a commutative ring.

quotient_by_principal_ideal (*f, names=None*)

Return the quotient of this polynomial ring by the principal ideal generated by f.

EXAMPLES:

class **PolynomialRing_dense_mod_n** (*base_ring, name=None, element_class=None, implementation=None*)

modulus ()

EXAMPLES:

```
sage: R.<x> = Zmod(15)[]
sage: R.modulus()
15
```

class **PolynomialRing_dense_mod_p** (*base_ring, name='x', implementation=None*)

class **PolynomialRing_dense_padic_field_capped_relative** (*base_ring, name=None, element_class=None*)

class PolynomialRing_dense_padic_field_generic(*base_ring, name='x', sparse=False, element_class=None*)

class PolynomialRing_dense_padic_field_lazy(*base_ring, name=None, element_class=None*)

class PolynomialRing_dense_padic_ring_capped_absolute(*base_ring, name=None, element_class=None*)

class PolynomialRing_dense_padic_ring_capped_relative(*base_ring, name=None, element_class=None*)

class PolynomialRing_dense_padic_ring_fixed_mod(*base_ring, name=None, element_class=None*)

class PolynomialRing_dense_padic_ring_generic(*base_ring, name='x', sparse=False, implementation=None, element_class=None*)

class PolynomialRing_dense_padic_ring_lazy(*base_ring, name=None, element_class=None*)

class PolynomialRing_field(*base_ring, name='x', sparse=False, element_class=None*)

divided_difference(*points, full_table=False*)

Return the Newton divided-difference coefficients of the n-th Lagrange interpolation polynomial of `points`.

If `points` are $n+1$ distinct points $(x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n))$, then $P_n(x)$ is the n-th Lagrange interpolation polynomial of $f(x)$ that passes through the points $(x_i, f(x_i))$. This method returns the coefficients $F_{i,i}$ such that

$$P_n(x) = \sum_{i=0}^{n} F_{i,i} \prod_{j=0}^{i-1} (x - x_j)$$

INPUT:

- `points` – a list of tuples $(x_0, f(x_0)), (x_1, f(x_1)), \ldots, (x_n, f(x_n))$ where each $x_i \neq x_j$ for $i \neq j$
- `full_table` – (default: `False`) if `True` then return the full divided-difference table; if `False` then only return entries along the main diagonal. The entries along the main diagonal are the Newton divided-difference coefficients $F_{i,i}$.

OUTPUT:

- The Newton divided-difference coefficients of the n-th Lagrange interpolation polynomial that passes through the points in `points`.

EXAMPLES:

Only return the divided-difference coefficients $F_{i,i}$. This example is taken from Example 1, p.121 of [BF05]:

```
sage: points = [(1.0, 0.7651977), (1.3, 0.6200860), (1.6, 0.4554022), (1.9, 0.2818186), (2.2
sage: R = PolynomialRing(QQ, "x")
sage: R.divided_difference(points)
<BLANKLINE>
[0.765197700000000,
-0.483705666666666,
-0.108733888888889,
0.0658783950617283,
0.00182510288066044]
```

Now return the full divided-difference table:

```
sage: points = [(1.0, 0.7651977), (1.3, 0.6200860), (1.6, 0.4554022), (1.9, 0.2818186), (2.2
sage: R = PolynomialRing(QQ, "x")
sage: R.divided_difference(points, full_table=True)
<BLANKLINE>
[[0.765197700000000],
```

```
[0.620086000000000, -0.483705666666666],
[0.455402200000000, -0.548946000000000, -0.108733888888889],
[0.281818600000000,
-0.578612000000000,
-0.0494433333333339,
0.0658783950617283],
[0.110362300000000,
-0.571520999999999,
0.0118183333333349,
0.0680685185185209,
0.00182510288066044]]
```

The following example is taken from Example 4.12, p.225 of [MF99]:

```
sage: points = [(1, -3), (2, 0), (3, 15), (4, 48), (5, 105), (6, 192)]
sage: R = PolynomialRing(RR, "x")
sage: R.divided_difference(points)
[-3, 3, 6, 1, 0, 0]
sage: R.divided_difference(points, full_table=True)
<BLANKLINE>
[[-3],
[0, 3],
[15, 15, 6],
[48, 33, 9, 1],
[105, 57, 12, 1, 0],
[192, 87, 15, 1, 0, 0]]
```

REFERENCES:

lagrange_polynomial(*points, algorithm='divided_difference', previous_row=None*)

Return the Lagrange interpolation polynomial in self associated to the given list of points.

Given a list of points, i.e. tuples of elements of self's base ring, this function returns the interpolation polynomial in the Lagrange form.

INPUT:

- `points` – a list of tuples representing points through which the polynomial returned by this function must pass.
- `algorithm` – (default: `'divided_difference'`) the available values for this option are `'divided_difference'` and `neville`.
  - If `algorithm='divided_difference'` then use the method of divided difference.
  - If `algorithm='neville'` then adapt Neville's method as described on page 144 of [BF05] to recursively generate the Lagrange interpolation polynomilal. Neville's method generates a table of approximating polynomials, where the last row of that table contains the n-th Lagrange interpolation polynomial. The adaptation implemented by this method is to only generate the last row of this table, instead of the full table itself. Generating the full table can be memory inefficient.
- `previous_row` – (default: `None`) This option is only relevant if used together with `algorithm='neville'`. If provided, this should be the last row of the table resulting from a previous use of Neville's method. If such a row is passed in, then `points` should consist of both previous and new interpolating points. Neville's method will then use that last row and the interpolating points to generate a new row which contains a better Lagrange interpolation polynomial.

EXAMPLE:

By default, we use the method of divided-difference:

```
sage: R = PolynomialRing(QQ, 'x')
sage: f = R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)]); f
-23/84*x^3 - 11/84*x^2 + 13/7*x + 1
```

```
sage: f(0)
1
sage: f(2)
2
sage: f(3)
-2
sage: f(-4)
9
sage: R = PolynomialRing(GF(2**3,'a'), 'x')
sage: a = R.base_ring().gen()
sage: f = R.lagrange_polynomial([(a^2+a,a),(a,1),(a^2,a^2+a+1)]); f
a^2*x^2 + a^2*x + a^2
sage: f(a^2+a)
a
sage: f(a)
1
sage: f(a^2)
a^2 + a + 1
```

Now use a memory efficient version of Neville's method:

```
sage: R = PolynomialRing(QQ, 'x')
sage: R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)], algorithm="neville")
<BLANKLINE>
[9,
-11/7*x + 19/7,
-17/42*x^2 - 83/42*x + 53/7,
-23/84*x^3 - 11/84*x^2 + 13/7*x + 1]
sage: R = PolynomialRing(GF(2**3,'a'), 'x')
sage: a = R.base_ring().gen()
sage: R.lagrange_polynomial([(a^2+a,a),(a,1),(a^2,a^2+a+1)], algorithm="neville")
[a^2 + a + 1, x + a + 1, a^2*x^2 + a^2*x + a^2]
```

Repeated use of Neville's method to get better Lagrange interpolation polynomials:

```
sage: R = PolynomialRing(QQ, 'x')
sage: p = R.lagrange_polynomial([(0,1),(2,2)], algorithm="neville")
sage: R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)], algorithm="neville", previous_row=p
-23/84*x^3 - 11/84*x^2 + 13/7*x + 1
sage: R = PolynomialRing(GF(2**3,'a'), 'x')
sage: a = R.base_ring().gen()
sage: p = R.lagrange_polynomial([(a^2+a,a),(a,1)], algorithm="neville")
sage: R.lagrange_polynomial([(a^2+a,a),(a,1),(a^2,a^2+a+1)], algorithm="neville", previous_r
a^2*x^2 + a^2*x + a^2
```

TESTS:

The value for `algorithm` must be either `'divided_difference'` (by default it is), or `'neville'`:

```
sage: R = PolynomialRing(QQ, "x")
sage: R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)], algorithm="abc")
...
ValueError: algorithm must be one of 'divided_difference' or 'neville'
sage: R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)], algorithm="divided difference")
...
ValueError: algorithm must be one of 'divided_difference' or 'neville'
sage: R.lagrange_polynomial([(0,1),(2,2),(3,-2),(-4,9)], algorithm="")
...
ValueError: algorithm must be one of 'divided_difference' or 'neville'
```

REFERENCES:

class **PolynomialRing_general**(*base_ring, name=None, sparse=False, element_class=None*)

Univariate polynomial ring over a ring.

base_extend(*R*)

Return the base extension of this polynomial ring to R.

EXAMPLES:

```
sage: R.<x> = RR[]; R
Univariate Polynomial Ring in x over Real Field with 53 bits of precision
sage: R.base_extend(CC)
Univariate Polynomial Ring in x over Complex Field with 53 bits of precision
sage: R.base_extend(QQ)
...
TypeError: no such base extension
sage: R.change_ring(QQ)
Univariate Polynomial Ring in x over Rational Field
```

change_ring(*R*)

Return the polynomial ring in the same variable as self over R.

EXAMPLES:

```
sage: R.<ZZZ> = RealIntervalField() []; R
Univariate Polynomial Ring in ZZZ over Real Interval Field with 53 bits of precision
sage: R.change_ring(GF(19^2,'b'))
Univariate Polynomial Ring in ZZZ over Finite Field in b of size 19^2
```

change_var(*var*)

Return the polynomial ring in variable var over the same base ring.

EXAMPLES:

```
sage: R.<x> = ZZ[]; R
Univariate Polynomial Ring in x over Integer Ring
sage: R.change_var('y')
Univariate Polynomial Ring in y over Integer Ring
```

characteristic()

Return the characteristic of this polynomial ring, which is the same as that of its base ring.

EXAMPLES:

```
sage: R.<ZZZ> = RealIntervalField() []; R
Univariate Polynomial Ring in ZZZ over Real Interval Field with 53 bits of precision
sage: R.characteristic()
0
sage: S = R.change_ring(GF(19^2,'b')); S
Univariate Polynomial Ring in ZZZ over Finite Field in b of size 19^2
sage: S.characteristic()
19
```

completion(*p, prec=20, extras=None*)

Return the completion of self with respect to the irreducible polynomial p. Currently only implemented for p=self.gen(), i.e. you can only complete R[x] with respect to x, the result being a ring of power series in x. The prec variable controls the precision used in the power series ring.

EXAMPLES:

```
sage: P.<x>=PolynomialRing(QQ)
sage: P
Univariate Polynomial Ring in x over Rational Field
sage: PP=P.completion(x)
sage: PP
Power Series Ring in x over Rational Field
```

```
sage: f=1-x
sage: PP(f)
1 - x
sage: 1/f
1/(-x + 1)
sage: 1/PP(f)
1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + x^9 + x^10 + x^11 + x^12 + x^13 + x^14 + x
```

construction()

cyclotomic_polynomial(*n*)

Return the nth cyclotomic polynomial as a polynomial in this polynomial ring.

EXAMPLES:

```
sage: R = ZZ['x']
sage: R.cyclotomic_polynomial(8)
x^4 + 1
sage: R.cyclotomic_polynomial(12)
x^4 - x^2 + 1
sage: S = PolynomialRing(FiniteField(7), 'x')
sage: S.cyclotomic_polynomial(12)
x^4 + 6*x^2 + 1
sage: S.cyclotomic_polynomial(1)
x + 6
```

TESTS:

Make sure it agrees with other systems for the trivial case:

```
sage: ZZ['x'].cyclotomic_polynomial(1)
x - 1
sage: gp('polcyclo(1)')
x - 1
```

extend_variables(*added_names, order='degrevlex'*)

Returns a multivariate polynomial ring with the same base ring but with added_names as additional variables.

EXAMPLES:

```
sage: R.<x> = ZZ[]; R
Univariate Polynomial Ring in x over Integer Ring
sage: R.extend_variables('y, z')
Multivariate Polynomial Ring in x, y, z over Integer Ring
sage: R.extend_variables(('y', 'z'))
Multivariate Polynomial Ring in x, y, z over Integer Ring
```

gen(*n=0*)

Return the indeterminate generator of this polynomial ring.

EXAMPLES:

```
sage: R.<abc> = Integers(8)[]; R
Univariate Polynomial Ring in abc over Ring of integers modulo 8
sage: t = R.gen(); t
abc
sage: t.is_gen()
True
```

An identical generator is always returned.

```
sage: t is R.gen()
True
```

gens_dict()
Returns a dictionary whose keys are the variable names of this ring as strings and whose values are the corresponding generators.

EXAMPLES:

```
sage: R.<x> = RR[]
sage: R.gens_dict()
{'x': 1.00000000000000*x}
```

is_exact()

is_field()
Return False, since polynomial rings are never fields.

EXAMPLES:

```
sage: R.<z> = Integers(2)[]; R
Univariate Polynomial Ring in z over Ring of integers modulo 2 (using NTL)
sage: R.is_field()
False
```

is_finite()
Return False since polynomial rings are not finite (unless the base ring is 0.)

EXAMPLES:

```
sage: R = Integers(1)['x']
sage: R.is_finite()
True
sage: R = GF(7)['x']
sage: R.is_finite()
False
sage: R['x']['y'].is_finite()
False
```

is_integral_domain()
EXAMPLES:

```
sage: ZZ['x'].is_integral_domain()
True
sage: Integers(8)['x'].is_integral_domain()
False
```

is_noetherian()

is_sparse()
Return true if elements of this polynomial ring have a sparse representation.

EXAMPLES:

```
sage: R.<z> = Integers(8)[]; R
Univariate Polynomial Ring in z over Ring of integers modulo 8
sage: R.is_sparse()
False
sage: R.<W> = PolynomialRing(QQ, sparse=True); R
Sparse Univariate Polynomial Ring in W over Rational Field
sage: R.is_sparse()
True
```

krull_dimension()
Return the Krull dimension of this polynomial ring, which is one more than the Krull dimension of the base ring.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: R.krull_dimension()
1
sage: R.<z> = GF(9,'a')[]; R
Univariate Polynomial Ring in z over Finite Field in a of size 3^2
sage: R.krull_dimension()
1
sage: S.<t> = R[]
sage: S.krull_dimension()
2
sage: for n in range(10):
...       S = PolynomialRing(S,'w')
sage: S.krull_dimension()
12
```

monics(*of_degree=None, max_degree=None*)

Return an iterator over the monic polynomials of specified degree.

INPUT: Pass exactly one of:

- `max_degree` - an int; the iterator will generate all monic polynomials which have degree less than or equal to max_degree
- `of_degree` - an int; the iterator will generate all monic polynomials which have degree of_degree

OUTPUT: an iterator

EXAMPLES:

```
sage: P = PolynomialRing(GF(4,'a'),'y')
sage: for p in P.monics( of_degree = 2 ): print p
y^2
y^2 + a
y^2 + a + 1
y^2 + 1
y^2 + a*y
y^2 + a*y + a
y^2 + a*y + a + 1
y^2 + a*y + 1
y^2 + (a + 1)*y
y^2 + (a + 1)*y + a
y^2 + (a + 1)*y + a + 1
y^2 + (a + 1)*y + 1
y^2 + y
y^2 + y + a
y^2 + y + a + 1
y^2 + y + 1
sage: for p in P.monics( max_degree = 1 ): print p
1
y
y + a
y + a + 1
y + 1
sage: for p in P.monics( max_degree = 1, of_degree = 3 ): print p
...
ValueError: you should pass exactly one of of_degree and max_degree
```

AUTHORS:

- Joel B. Mohler

ngens()

Return the number of generators of this polynomial ring, which is 1 since it is a univariate polynomial ring.

EXAMPLES:

```
sage: R.<z> = Integers(8)[]; R
Univariate Polynomial Ring in z over Ring of integers modulo 8
sage: R.ngens()
1
```

parameter()

Return the generator of this polynomial ring.

This is the same as `self.gen()`.

polynomials(*of_degree=None, max_degree=None*)

Return an iterator over the polynomials of specified degree.

INPUT: Pass exactly one of:

- `max_degree` - an int; the iterator will generate all polynomials which have degree less than or equal to max_degree
- `of_degree` - an int; the iterator will generate all polynomials which have degree of_degree

OUTPUT: an iterator

EXAMPLES:

```
sage: P = PolynomialRing(GF(3),'y')
sage: for p in P.polynomials( of_degree = 2 ): print p
y^2
y^2 + 1
y^2 + 2
y^2 + y
y^2 + y + 1
y^2 + y + 2
y^2 + 2*y
y^2 + 2*y + 1
y^2 + 2*y + 2
2*y^2
2*y^2 + 1
2*y^2 + 2
2*y^2 + y
2*y^2 + y + 1
2*y^2 + y + 2
2*y^2 + 2*y
2*y^2 + 2*y + 1
2*y^2 + 2*y + 2
sage: for p in P.polynomials( max_degree = 1 ): print p
0
1
2
y
y + 1
y + 2
2*y
2*y + 1
2*y + 2
sage: for p in P.polynomials( max_degree = 1, of_degree = 3 ): print p
...
ValueError: you should pass exactly one of of_degree and max_degree
```

AUTHORS:

- Joel B. Mohler

random_element(*degree=2, *args, **kwds*)

Return a random polynomial.

INPUT:

- `degree` - an integer
- `*args, **kwds` - passed on to the `random_element` method for the base ring.

OUTPUT:

- `Polynomial` - A polynomial such that the coefficient of x^i, for i up to `degree`, are coercions to the base ring of random integers between -`bound` and `bound`.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: R.random_element(10, 5,10)
9*x^10 + 8*x^9 + 6*x^8 + 8*x^7 + 8*x^6 + 9*x^5 + 8*x^4 + 8*x^3 + 6*x^2 + 8*x + 8
sage: R.random_element(6)
x^6 - 3*x^5 - x^4 + x^3 - x^2 + x + 1
sage: R.random_element(6)
-2*x^5 + 2*x^4 - 3*x^3 + 1
sage: R.random_element(6)
x^4 - x^3 + x - 2
```

variable_names_recursive(*depth=+Infinity*)

Returns the list of variable names of this and its base rings, as if it were a single multi-variate polynomial.

EXAMPLES:

```
sage: R = QQ['x']['y']['z']
sage: R.variable_names_recursive()
('x', 'y', 'z')
sage: R.variable_names_recursive(2)
('y', 'z')
```

class **PolynomialRing_integral_domain**(*base_ring, name='x', sparse=False, implementation=None, element_class=None*)

is_PolynomialRing(*x*)

Return True if x is a *univariate* polynomial ring (and not a sparse multivariate polynomial ring in one variable).

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_ring import is_PolynomialRing
sage: from sage.rings.polynomial.multi_polynomial_ring import is_MPolynomialRing
sage: is_PolynomialRing(2)
False
```

This polynomial ring is not univariate.

```
sage: is_PolynomialRing(ZZ['x,y,z'])
False
sage: is_MPolynomialRing(ZZ['x,y,z'])
True
```

```
sage: is_PolynomialRing(ZZ['w'])
True
```

Univariate means not only in one variable, but is a specific data type. There is a multivariate (sparse) polynomial ring data type, which supports a single variable as a special case.

```
sage: is_PolynomialRing(PolynomialRing(ZZ,1,'w'))
False
sage: R = PolynomialRing(ZZ,1,'w'); R
Multivariate Polynomial Ring in w over Integer Ring
sage: is_PolynomialRing(R)
```

```
False
sage: type(R)
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomialRing_libsingular'>
```

polygen(*ring_or_element, name='x'*)

Return a polynomial indeterminate.

INPUT:

- polygen(base_ring, name="x")
- polygen(ring_element, name="x")

If the first input is a ring, return a polynomial generator over that ring. If it is a ring element, return a polynomial generator over the parent of the element.

EXAMPLES:

```
sage: z = polygen(QQ,'z')
sage: z^3 + z +1
z^3 + z + 1
sage: parent(z)
Univariate Polynomial Ring in z over Rational Field
```

Note: If you give a list or comma separated string to polygen, you'll get a tuple of indeterminates, exactly as if you called polygens.

polygens(*base_ring, names='x'*)

Return indeterminates over the given base ring with the given names.

EXAMPLES:

```
sage: x,y,z = polygens(QQ,'x,y,z')
sage: (x+y+z)^2
x^2 + 2*x*y + y^2 + 2*x*z + 2*y*z + z^2
sage: parent(x)
Multivariate Polynomial Ring in x, y, z over Rational Field
sage: t = polygens(QQ,['x','yz','abc'])
sage: t
(x, yz, abc)
```

28.2 Univariate Polynomial Base Class

AUTHORS:

- William Stein: first version
- Martin Albrecht: Added singular coercion.
- Robert Bradshaw: Move Polynomial_generic_dense to Cython

TESTS:

```
sage: R.<x> = ZZ[]
sage: f = x^5 + 2*x^2 + (-1)
sage: f == loads(dumps(f))
True
```

class ConstantPolynomialSection()

This class is used for conversion from a polynomial ring to its base ring.

It calls the constant_coefficient method, which can be optimized for a particular polynomial type.

class Polynomial()

A polynomial.

EXAMPLE:

```
sage: R.<y> = QQ['y']
sage: S.<x> = R['x']
sage: f = x*y; f
y*x
sage: type(f)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: p = (y+1)^10; p(1)
1024
```

args()

Returns the generator of this polynomial ring, which is the (only) argument used when calling self.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: x.args()
(x,)
```

A constant polynomial has no variables, but still takes a single argument.

```
sage: R(2).args()
(x,)
```

base_extend()

Return a copy of this polynomial but with coefficients in R, if there is a natural map from coefficient ring of self to R.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^3 - 17*x + 3
sage: f.base_extend(GF(7))
...
TypeError: no such base extension
sage: f.change_ring(GF(7))
x^3 + 4*x + 3
```

base_ring()

Return the base ring of the parent of self.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: x.base_ring()
Integer Ring
sage: (2*x+3).base_ring()
Integer Ring
```

change_ring()

Return a copy of this polynomial but with coefficients in R, if at all possible.

EXAMPLES:

```
sage: K.<z> = CyclotomicField(3)
sage: f = K.defining_polynomial()
```

```
sage: f.change_ring(GF(7))
x^2 + x + 1
```

change_variable_name()

Return a new polynomial over the same base ring but in a different variable.

EXAMPLES:

```
sage: x = polygen(QQ,'x')
sage: f = -2/7*x^3 + (2/3)*x - 19/993; f
-2/7*x^3 + 2/3*x - 19/993
sage: f.change_variable_name('theta')
-2/7*theta^3 + 2/3*theta - 19/993
```

coefficients()

Return the coefficients of the monomials appearing in self.

EXAMPLES:

```
sage: _.<x> = PolynomialRing(ZZ)
sage: f = x^4+2*x^2+1
sage: f.coefficients()
[1, 2, 1]
```

coeffs()

Returns `self.list()`.

(It is potentially slightly faster to use `self.list()` directly.)

EXAMPLES:

```
sage: x = QQ['x'].0
sage: f = 10*x^3 + 5*x + 2/17
sage: f.coeffs()
[2/17, 5, 0, 10]
```

complex_roots()

Return the complex roots of this polynomial, without multiplicities.

Calls self.roots(ring=CC), unless this is a polynomial with floating-point coefficients, in which case it is uses the appropriate precision from the input coefficients.

EXAMPLES:

```
sage: x = polygen(ZZ)
sage: (x^3 - 1).complex_roots()   # note: low order bits slightly different on ppc.
[1.00000000000000, -0.500000000000000 - 0.86602540378443...*I, -0.500000000000000 + 0.866025
```

TESTS:

```
sage: x = polygen(RR)
sage: (x^3 - 1).complex_roots()[0].parent()
Complex Field with 53 bits of precision
sage: x = polygen(RDF)
sage: (x^3 - 1).complex_roots()[0].parent()
Complex Double Field
sage: x = polygen(RealField(200))
sage: (x^3 - 1).complex_roots()[0].parent()
Complex Field with 200 bits of precision
sage: x = polygen(CDF)
sage: (x^3 - 1).complex_roots()[0].parent()
Complex Double Field
sage: x = polygen(ComplexField(200))
sage: (x^3 - 1).complex_roots()[0].parent()
Complex Field with 200 bits of precision
```

constant_coefficient()

Return the constant coefficient of this polynomial.

OUTPUT: element of base ring

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = -2*x^3 + 2*x - 1/3
sage: f.constant_coefficient()
-1/3
```

degree()

Return the degree of this polynomial. The zero polynomial has degree -1.

EXAMPLES:

```
sage: x = ZZ['x'].0
sage: f = x^93 + 2*x + 1
sage: f.degree()
93
sage: x = PolynomialRing(QQ, 'x', sparse=True).0
sage: f = x^100000
sage: f.degree()
100000

sage: x = QQ['x'].0
sage: f = 2006*x^2006 - x^2 + 3
sage: f.degree()
2006
sage: f = 0*x
sage: f.degree()
-1
sage: f = x + 33
sage: f.degree()
1
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples

denominator()

Return the least common multiple of the denominators of the entries of self, when this makes sense, i.e., when the coefficients have a denominator function.

Warning: This is not the denominator of the rational function defined by self, which would always be 1 since self is a polynomial.

EXAMPLES:

First we compute the denominator of a polynomial with integer coefficients, which is of course 1.

```
sage: R.<x> = ZZ[]
sage: f = x^3 + 17*x + 1
sage: f.denominator()
1
```

Next we compute the denominator of a polynomial with rational coefficients.

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = (1/17)*x^19 - (2/3)*x + 1/3; f
1/17*x^19 - 2/3*x + 1/3
sage: f.denominator()
51
```

Finally, we try to compute the denominator of a polynomial with coefficients in the real numbers, which is a ring whose elements do not have a denominator method.

```
sage: R.<x> = RR[]
sage: f = x + RR('0.3'); f
1.00000000000000*x + 0.300000000000000
sage: f.denominator()
...
AttributeError: 'sage.rings.real_mpfr.RealNumber' object has no attribute 'denominator'
```

derivative()

The formal derivative of this polynomial, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See Also:

_derivative()

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: g = -x^4 + x^2/2 - x
sage: g.derivative()
-4*x^3 + x - 1
sage: g.derivative(x)
-4*x^3 + x - 1
sage: g.derivative(x, x)
-12*x^2 + 1
sage: g.derivative(x, 2)
-12*x^2 + 1

sage: R.<t> = PolynomialRing(ZZ)
sage: S.<x> = PolynomialRing(R)
sage: f = t^3*x^2 + t^4*x^3
sage: f.derivative()
3*t^4*x^2 + 2*t^3*x
sage: f.derivative(x)
3*t^4*x^2 + 2*t^3*x
sage: f.derivative(t)
4*t^3*x^3 + 3*t^2*x^2
```

dict()

Return a sparse dictionary representation of this univariate polynomial.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^3 + -1/7*x + 13
sage: f.dict()
{0: 13, 1: -1/7, 3: 1}
```

discriminant()

Returns the discriminant of self.

The discriminant is

$$R_n := a_n^{2n-2} \prod_{1<i<j<n} (r_i - r_j)^2,$$

where n is the degree of self, a_n is the leading coefficient of self and the roots of self are $r_1, \ldots, r_n$.

OUTPUT: An element of the base ring of the polynomial ring.

Note: Uses the identity $R_n(f) := (-1)^{(}n(n-1)/2)R(f,f')a_n^{(}n-k-2)$, where n is the degree of self, a_n is the leading coefficient of self, f' is the derivative of f, and k is the degree of f'. Calls `resultant()`.

EXAMPLES:

In the case of elliptic curves in special form, the discriminant is easy to calculate:

```
sage: R.<x> = QQ[]
sage: f = x^3 + x + 1
sage: d = f.discriminant(); d
-31
sage: d.parent() is QQ
True
sage: EllipticCurve([1, 1]).discriminant()/16
-31

sage: R.<x> = QQ[]
sage: f = 2*x^3 + x + 1
sage: d = f.discriminant(); d
-116
```

We can also compute discriminants over univariate and multivariate polynomial rings, provided that PARI's variable ordering requirements are respected. Usually, your discriminants will work if you always ask for them in the variable `x`:

```
sage: R.<a> = QQ[]
sage: S.<x> = R[]
sage: f = a*x + x + a + 1
sage: d = f.discriminant(); d
1
sage: d.parent() is R
True

sage: R.<a, b> = QQ[]
sage: S.<x> = R[]
sage: f = x^2 + a + b
sage: d = f.discriminant(); d
-4*a - 4*b
sage: d.parent() is R
True
```

Unfortunately Sage does not handle PARI's variable ordering requirements gracefully, so the following fails:

```
sage: R.<x, y> = QQ[]
sage: S.<a> = R[]
sage: f = x^2 + a
sage: f.discriminant()
...
PariError: (8)
```

exponents()

Return the exponents of the monomials appearing in self.

EXAMPLES:

```
sage: _.<x> = PolynomialRing(ZZ)
sage: f = x^4+2*x^2+1
sage: f.exponents()
[0, 2, 4]
```

factor()

Return the factorization of self over the base ring of this polynomial. Factoring polynomials over $\mathbf{Z}/n\mathbf{Z}$ for n composite is at the moment not implemented.

INPUT: a polynomial

OUTPUT:

- `Factorization` - the factorization of self, which is a product of a unit with a product of powers of irreducible factors.

Over a field the irreducible factors are all monic.

EXAMPLES:

We factor some polynomials over $\mathbf{Q}$.

```
sage: x = QQ['x'].0
sage: f = (x^3 - 1)^2
sage: f.factor()
(x - 1)^2 * (x^2 + x + 1)^2
```

Notice that over the field $\mathbf{Q}$ the irreducible factors are monic.

```
sage: f = 10*x^5 - 1
sage: f.factor()
(10) * (x^5 - 1/10)
sage: f = 10*x^5 - 10
sage: f.factor()
(10) * (x - 1) * (x^4 + x^3 + x^2 + x + 1)
```

Over $\mathbf{Z}$ the irreducible factors need not be monic:

```
sage: x = ZZ['x'].0
sage: f = 10*x^5 - 1
sage: f.factor()
10*x^5 - 1
```

We factor a non-monic polynomial over the finite field F_{25}.

```
sage: k.<a> = GF(25)
sage: R.<x> = k[]
sage: f = 2*x^10 + 2*x + 2*a
sage: F = f.factor(); F
(2) * (x + a + 2) * (x^2 + 3*x + 4*a + 4) * (x^2 + (a + 1)*x + a + 2) * (x^5 + (3*a + 4)*x^4
```

Notice that the unit factor is included when we multiply F back out.

```
sage: expand(F)
2*x^10 + 2*x + 2*a
```

Factorization also works even if the variable of the finite field is nefariously labeled "x".

```
sage: x = GF(3^2, 'a')['x'].0
sage: f = x^10 +7*x -13
sage: G = f.factor(); G
(x + a) * (x + 2*a + 1) * (x^4 + (a + 2)*x^3 + (2*a + 2)*x + 2) * (x^4 + 2*a*x^3 + (a + 1)*x
sage: prod(G) == f
True

sage: f.parent().base_ring()._assign_names(['a'])
sage: f.factor()
(x + a) * (x + 2*a + 1) * (x^4 + (a + 2)*x^3 + (2*a + 2)*x + 2) * (x^4 + 2*a*x^3 + (a + 1)*x

sage: k = GF(9,'x')    # purposely calling it x to test robustness
sage: x = PolynomialRing(k,'x0').gen()
sage: f = x^3 + x + 1
sage: f.factor()
```

```
(x0 + 2) * (x0 + x) * (x0 + 2*x + 1)
sage: f = 0*x
sage: f.factor()
...
ValueError: factorization of 0 not defined

sage: f = x^0
sage: f.factor()
1
```

Arbitrary precision real and complex factorization:

```
sage: R.<x> = RealField(100)[]
sage: F = factor(x^2-3); F
(1.0000000000000000000000000000*x - 1.7320508075688772935274463415) * (1.0000000000000000000
sage: expand(F)
1.0000000000000000000000000000*x^2 - 3.0000000000000000000000000000
sage: factor(x^2 + 1)
1.0000000000000000000000000000*x^2 + 1.0000000000000000000000000000
sage: C = ComplexField(100)
sage: R.<x> = C[]
sage: F = factor(x^2+3); F
(1.0000000000000000000000000000*x - 1.7320508075688772935274463415*I) * (1.0000000000000000
sage: expand(F)
1.0000000000000000000000000000*x^2 + 3.0000000000000000000000000000
sage: factor(x^2+1)
(1.0000000000000000000000000000*x - 1.0000000000000000000000000000*I) * (1.0000000000000000
sage: f = C.0 * (x^2 + 1) ; f
1.0000000000000000000000000000*I*x^2 + 1.0000000000000000000000000000*I
sage: F = factor(f); F
(1.0000000000000000000000000000*I) * (1.0000000000000000000000000000*x - 1.0000000000000000
sage: expand(F)
1.0000000000000000000000000000*I*x^2 + 1.0000000000000000000000000000*I
```

Over a complicated number field:

```
sage: x = polygen(QQ, 'x')
sage: f = x^6 + 10/7*x^5 - 867/49*x^4 - 76/245*x^3 + 3148/35*x^2 - 25944/245*x + 48771/1225
sage: K.<a> = NumberField(f)
sage: S.<T> = K[]
sage: ff = S(f); ff
T^6 + 10/7*T^5 - 867/49*T^4 - 76/245*T^3 + 3148/35*T^2 - 25944/245*T + 48771/1225
sage: F = ff.factor()
sage: len(F)
4
sage: F[:2]
[(T - a, 1), (T - 40085763200/924556084127*a^5 - 145475769880/924556084127*a^4 + 5276170964
sage: expand(F)
T^6 + 10/7*T^5 - 867/49*T^4 - 76/245*T^3 + 3148/35*T^2 - 25944/245*T + 48771/1225

sage: f = x^2 - 1/3 ; K.<a> = NumberField(f) ; A.<T> = K[] ; g = A(x^2-1)
sage: g.factor()
(T - 1) * (T + 1)

sage: h = A(3*x^2-1) ; h.factor()
(3) * (T - a) * (T + a)

sage: h = A(x^2-1/3) ; h.factor()
(T - a) * (T + a)
```

Over the real double field:

```
sage: x = polygen(RDF)
sage: f = (x-1)^3
sage: f.factor() # random output (unfortunately)
(1.0*x - 1.00000859959) * (1.0*x^2 - 1.99999140041*x + 0.999991400484)
sage: (-2*x^2 - 1).factor()
(-2.0) * (1.0*x^2 + 0.5)
sage: (-2*x^2 - 1).factor().expand()
-2.0*x^2 - 1.0
```

Note that this factorization suffers from the roots function:

```
sage: f.roots() # random output (unfortunately)
[1.00000859959, 0.999995700205 + 7.44736245561e-06*I, 0.999995700205 - 7.44736245561e-06*I]
```

Over the complex double field. Because this is approximate, all factors will occur with multiplicity 1.

```
sage: x = CDF['x'].0; i = CDF.0
sage: f = (x^2 + 2*i)^3
sage: f.factor()    # random low order bits
(1.0*x + -0.999994409957 + 1.00001040378*I) * (1.0*x + -0.999993785062 + 0.999989956987*I) *
sage: f(-f.factor()[0][0][0])   # random low order bits
-2.38358052913e-14 - 2.57571741713e-14*I
```

Over a relative number field:

```
sage: x = QQ['x'].0
sage: L.<a> = CyclotomicField(3).extension(x^3 - 2)
sage: x = L['x'].0
sage: f = (x^3 + x + a)*(x^5 + x + L.1); f
x^8 + x^6 + a*x^5 + x^4 + zeta3*x^3 + x^2 + (a + zeta3)*x + zeta3*a
sage: f.factor()
(x^3 + x + a) * (x^5 + x + zeta3)
```

Factoring polynomials over $\mathbf{Z}/n\mathbf{Z}$ for composite n is not implemented:

```
sage: R.<x> = PolynomialRing(Integers(35))
sage: f = (x^2+2*x+2)*(x^2+3*x+9)
sage: f.factor()
...
NotImplementedError: factorization of polynomials over rings with composite characteristic i
```

hamming_weight()

Returns the number of non-zero coefficients of self.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: f = x^3 - x
sage: f.hamming_weight()
2
sage: R(0).hamming_weight()
0
sage: f = (x+1)^100
sage: f.hamming_weight()
101
sage: S = GF(5)['y']
sage: S(f).hamming_weight()
5
sage: cyclotomic_polynomial(105).hamming_weight()
33
```

integral()

Return the integral of this polynomial.

Note: The integral is always chosen so the constant term is 0.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: R(0).integral()
0
sage: f = R(2).integral(); f
2*x
```

Note that since the integral is defined over the same base ring the integral is actually in the base ring.

```
sage: f.parent()
Univariate Polynomial Ring in x over Integer Ring
```

If the integral isn't defined over the same base ring, then the base ring is extended:

```
sage: f = x^3 + x - 2
sage: g = f.integral(); g
1/4*x^4 + 1/2*x^2 - 2*x
sage: g.parent()
Univariate Polynomial Ring in x over Rational Field
```

inverse_mod()

Inverts the polynomial a with respect to m, or raises a ValueError if no such inverse exists. The parameter m may be either a single polynomial or an ideal (for consistency with inverse_mod in other rings).

EXAMPLES:

```
sage: S.<t> = QQ[]
sage: f = inverse_mod(t^2 + 1, t^3 + 1); f
-1/2*t^2 - 1/2*t + 1/2
sage: f * (t^2 + 1) % (t^3 + 1)
1
sage: f = t.inverse_mod((t+1)^7); f
-t^6 - 7*t^5 - 21*t^4 - 35*t^3 - 35*t^2 - 21*t - 7
sage: (f * t) + (t+1)^7
1
sage: t.inverse_mod(S.ideal((t + 1)^7)) == f
True
```

It also works over in-exact rings, but note that due to rounding error the product is only guaranteed to be within epsilon of the constant polynomial 1.

```
sage: R.<x> = RDF[]
sage: f = inverse_mod(x^2 + 1, x^5 + x + 1); f
0.4*x^4 - 0.2*x^3 - 0.4*x^2 + 0.2*x + 0.8
sage: f * (x^2 + 1) % (x^5 + x + 1)
2.22044604925e-16*x^2 + 1.11022302463e-16*x + 1.0
sage: f = inverse_mod(x^3 - x + 1, x - 2); f
0.142857142857
sage: f * (x^3 - x + 1) % (x - 2)
1.0
```

ALGORITHM: Solve the system as + mt = 1, returning s as the inverse of a mod m.

Uses the Euclidean algorithm for exact rings, and solves a linear system for the coefficients of s and t for inexact rings (as the Euclidean algorithm may not converge in that case).

AUTHORS:

- Robert Bradshaw (2007-05-31)

inverse_of_unit()

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x - 90283
sage: f.inverse_of_unit()
...
ValueError: self is not a unit.
sage: f = R(-90283); g = f.inverse_of_unit(); g
-1/90283
sage: parent(g)
Univariate Polynomial Ring in x over Rational Field
```

is_constant()

Return True if this is a constant polynomial.

OUTPUT:

- bool - True if and only if this polynomial is constant

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: x.is_constant()
False
sage: R(2).is_constant()
True
sage: R(0).is_constant()
True
```

is_gen()

Return True if this polynomial is the distinguished generator of the parent polynomial ring.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: R(1).is_gen()
False
sage: R(x).is_gen()
True
```

Important - this function doesn't return True if self equals the generator; it returns True if self *is* the generator.

```
sage: f = R([0,1]); f
x
sage: f.is_gen()
False
sage: f is x
False
sage: f == x
True
```

is_irreducible()

Return True precisely if this polynomial is irreducible over its base ring. Testing irreducibility over $\mathbf{Z}/n\mathbf{Z}$ for composite n is not implemented.

The function returns False for polynomials which are units, and raises an exception for the zero polynomial.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: (x^3 + 1).is_irreducible()
False
sage: (x^2 - 1).is_irreducible()
False
sage: (x^3 + 2).is_irreducible()
```

```
True
sage: R(0).is_irreducible()
...
ValueError: self must be nonzero
```

See #5140: sage: R(1).is_irreducible() False sage: R(4).is_irreducible() False sage: R(5).is_irreducible() True

The base ring does matter: for example, 2x is irreducible as a polynomial in QQ[x], but not in ZZ[x]:

sage: R.<x> = ZZ[] sage: R(2*x).is_irreducible() False sage: R.<x> = QQ[] sage: R(2*x).is_irreducible() True

TESTS:

```
sage: F.<t> = NumberField(x^2-5)
sage: Fx.<xF> = PolynomialRing(F)
sage: f = Fx([2*t - 5, 5*t - 10, 3*t - 6, -t, -t + 2, 1])
sage: f.is_irreducible()
False
sage: f = Fx([2*t - 3, 5*t - 10, 3*t - 6, -t, -t + 2, 1])
sage: f.is_irreducible()
True
```

is_monic()

Returns True if this polynomial is monic. The zero polynomial is by definition not monic.

EXAMPLES:

```
sage: x = QQ['x'].0
sage: f = x + 33
sage: f.is_monic()
True
sage: f = 0*x
sage: f.is_monic()
False
sage: f = 3*x^3 + x^4 + x^2
sage: f.is_monic()
True
sage: f = 2*x^2 + x^3 + 56*x^5
sage: f.is_monic()
False
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples

is_monomial()

Returns True if this is a monomial.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: x.is_monomial()
True
sage: (x+1).is_monomial()
False
sage: (x^2).is_monomial()
True
```

is_nilpotent()

Return True if this polynomial is nilpotent.

EXAMPLES:

```
sage: R = Integers(12)
sage: S.<x> = R[]
sage: f = 5 + 6*x
sage: f.is_nilpotent()
False
sage: f = 6 + 6*x^2
sage: f.is_nilpotent()
True
sage: f^2
0
```

EXERCISE (Atiyah-McDonald, Ch 1): Let $A[x]$ be a polynomial ring in one variable. Then $f = \sum a_i x^i \in A[x]$ is nilpotent if and only if every a_i is nilpotent.

is_primitive()

Returns `True` if the polynomial is primitive. The semantics of "primitive" depend on the polynomial coefficients.

- (field theory) A polynomial of degree m over a finite field $\mathbf{F}_q$ is primitive if it is irreducible and its root in $\mathbf{F}_{q^m}$ generates the multiplicative group $\mathbf{F}_{q^m}^*$.
- (ring theory) A polynomial over a ring is primitive if its coefficients generate the unit ideal.

Calling $is_primitive$ on a polynomial over an infinite field will raise an error.

The additional inputs to this function are to speed up computation for field semantics (see note).

INPUTS:

- `n` (default: `None`) - if provided, should equal $q - 1$ where `self.parent()` is the field with q elements; otherwise it will be computed.
- `n_prime_divs` (default: `None`) - if provided, should be a list of the prime divisors of `n`; otherwise it will be computed.

Note: Computation of the prime divisors of `n` can dominate the running time of this method, so performing this computation externally (e.g. `pdivs=n.prime_divisors()`) is a good idea for repeated calls to is_primitive for polynomials of the same degree.

Results may be incorrect if the wrong `n` and/or factorization are provided.

EXAMPLES:

Field semantics examples.

::

```
sage: R.<x> = GF(2)['x']
sage: f = x^4+x^3+x^2+x+1
sage: f.is_irreducible(), f.is_primitive()
(True, False)
sage: f = x^3+x+1
sage: f.is_irreducible(), f.is_primitive()
(True, True)
sage: R.<x> = GF(3)[]
sage: f = x^3-x+1
sage: f.is_irreducible(), f.is_primitive()
(True, True)
sage: f = x^2+1
sage: f.is_irreducible(), f.is_primitive()
(True, False)
sage: R.<x> = GF(5)[]
sage: f = x^2+x+1
sage: f.is_primitive()
False
```

```
  sage: f = x^2-x+2
  sage: f.is_primitive()
  True
  sage: x=polygen(QQ); f=x^2+1
  sage: f.is_primitive()
  Traceback (most recent call last):
  ...
  NotImplementedError: is_primitive() not defined for polynomials over infinite fields.
```

Ring semantics examples.

::

```
  sage: x=polygen(ZZ)
  sage: f = 5*x^2+2
  sage: f.is_primitive()
  True
  sage: f = 5*x^2+5
  sage: f.is_primitive()
  False

  sage: K=NumberField(x^2+5,'a')
  sage: R=K.ring_of_integers()
  sage: a=R.gen(1)
  sage: a^2
  -5
  sage: f=a*x+2
  sage: f.is_primitive()
  True
  sage: f=(1+a)*x+2
  sage: f.is_primitive()
  False

  sage: x=polygen(Integers(10));
  sage: f=5*x^2+2
  sage: #f.is_primitive()  #BUG:: elsewhere in Sage, should return True
  sage: f=4*x^2+2
  sage: #f.is_primitive()  #BUG:: elsewhere in Sage, should return False
```

TESTS:

```
sage: R.<x> = GF(2)['x']
sage: f = x^4+x^3+x^2+x+1
sage: f.is_primitive(15)
False
sage: f.is_primitive(15, [3,5])
False
sage: f.is_primitive(n_prime_divs=[3,5])
False
sage: f = x^3+x+1
sage: f.is_primitive(7, [7])
True
sage: R.<x> = GF(3)[]
sage: f = x^3-x+1
sage: f.is_primitive(26, [2,13])
True
sage: f = x^2+1
sage: f.is_primitive(8, [2])
False
```

```
sage: R.<x> = GF(5)[]
sage: f = x^2+x+1
sage: f.is_primitive(24, [2,3])
False
sage: f = x^2-x+2
sage: f.is_primitive(24, [2,3])
True
sage: x=polygen(Integers(103)); f=x^2+1
sage: f.is_primitive()
False
```

is_square()

Returns whether or not polynomial is square. If the optional argument root is True, then also returns the square root (or None, if the polynomial is not square).

INPUT:

- `root` - whether or not to also return a square root (default: False)

OUTPUT:

- `bool` - whether or not a square
- `object` - (optional) an actual square root if found, and None otherwise.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = 12*(x+1)^2 * (x+3)^2
sage: S.<y> = PolynomialRing(RR)
sage: g = 12*(y+1)^2 * (y+3)^2
sage: f.is_square()
False
sage: f.is_square(root=True)
(False, None)
sage: g.is_square()
True
sage: h = f/3; h
4*x^4 + 32*x^3 + 88*x^2 + 96*x + 36
sage: h.is_square(root=True)
(True, 2*x^2 + 8*x + 6)
```

is_squarefree()

Return True if this polynomial is square free.

EXAMPLES:

```
sage: x = polygen(QQ)
sage: f = (x-1)*(x-2)*(x^2-5)*(x^17-3); f
x^21 - 3*x^20 - 3*x^19 + 15*x^18 - 10*x^17 - 3*x^4 + 9*x^3 + 9*x^2 - 45*x + 30
sage: f.is_squarefree()
True
sage: (f*(x^2-5)).is_squarefree()
False
```

is_unit()

Return True if this polynomial is a unit.

EXAMPLES:

```
sage: a = Integers(90384098234^3)
sage: b = a(2*191*236607587)
sage: b.is_nilpotent()
True
sage: R.<x> = a[]
```

```
sage: f = 3 + b*x + b^2*x^2
sage: f.is_unit()
True
sage: f = 3 + b*x + b^2*x^2 + 17*x^3
sage: f.is_unit()
False
```

EXERCISE (Atiyah-McDonald, Ch 1): Let $A[x]$ be a polynomial ring in one variable. Then $f = \sum a_i x^i \in A[x]$ is a unit if and only if a_0 is a unit and $a_1, \ldots, a_n$ are nilpotent.

lcm()

Let f and g be two polynomials. Then this function returns the monic least common multiple of f and g.

leading_coefficient()

Return the leading coefficient of this polynomial.

OUTPUT: element of the base ring

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = (-2/5)*x^3 + 2*x - 1/3
sage: f.leading_coefficient()
-2/5
```

list()

Return a new copy of the list of the underlying elements of self.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = (-2/5)*x^3 + 2*x - 1/3
sage: v = f.list(); v
[-1/3, 2, 0, -2/5]
```

Note that v is a list, it is mutable, and each call to the list method returns a new list:

```
sage: type(v)
<type 'list'>
sage: v[0] = 5
sage: f.list()
[-1/3, 2, 0, -2/5]
```

Here is an example with a generic polynomial ring:

```
sage: R.<x> = QQ[]
sage: S.<y> = R[]
sage: f = y^3 + x*y -3*x; f
y^3 + x*y - 3*x
sage: type(f)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: v = f.list(); v
[-3*x, x, 0, 1]
sage: v[0] = 10
sage: f.list()
[-3*x, x, 0, 1]
```

monic()

Return this polynomial divided by its leading coefficient. Does not change this polynomial.

EXAMPLES:

```
sage: x = QQ['x'].0
sage: f = 2*x^2 + x^3 + 56*x^5
sage: f.monic()
```

```
x^5 + 1/56*x^3 + 1/28*x^2
sage: f = (1/4)*x^2 + 3*x + 1
sage: f.monic()
x^2 + 12*x + 4
```

The following happens because $f = 0$ cannot be made into a monic polynomial

```
sage: f = 0*x
sage: f.monic()
...
ZeroDivisionError: rational division by zero
```

Notice that the monic version of a polynomial over the integers is defined over the rationals.

```
sage: x = ZZ['x'].0
sage: f = 3*x^19 + x^2 - 37
sage: g = f.monic(); g
x^19 + 1/3*x^2 - 37/3
sage: g.parent()
Univariate Polynomial Ring in x over Rational Field
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples

name()

Note: This function is deprecated. It will be removed in a future release of Sage. Please use the .variable_name() function instead.

Return the string variable name of the indeterminate of this polynomial.

EXAMPLES:

```
sage: R.<theta> = ZZ[];
sage: f = (2-theta)^3; f
-theta^3 + 6*theta^2 - 12*theta + 8
sage: f.name()
doctest:...: DeprecationWarning: This function is deprecated. It will be removed in a future
'theta'
```

newton_raphson()

Return a list of n iterative approximations to a root of this polynomial, computed using the Newton-Raphson method.

The Newton-Raphson method is an iterative root-finding algorithm. For f(x) a polynomial, as is the case here, this is essentially the same as Horner's method.

INPUT:

- `n` - an integer (=the number of iterations),
- `x0` - an initial guess x0.

OUTPUT: A list of numbers hopefully approximating a root of f(x)=0.

If one of the iterates is a critical point of f then a ZeroDivisionError exception is raised.

EXAMPLES:

```
sage: x = PolynomialRing(RealField(), 'x').gen()
sage: f = x^2 - 2
sage: f.newton_raphson(4, 1)
[1.50000000000000, 1.41666666666667, 1.41421568627451, 1.41421356237469]
```

AUTHORS:

- David Joyner and William Stein (2005-11-28)

newton_slopes()

Return the p-adic slopes of the Newton polygon of self, when this makes sense.

OUTPUT: list of rational numbers

EXAMPLES:

```
sage: x = QQ['x'].0
sage: f = x^3 + 2
sage: f.newton_slopes(2)
[1/3, 1/3, 1/3]
```

ALGORITHM: Uses PARI.

norm()

Return the p-norm of this polynomial.

DEFINITION: For integer p, the p-norm of a polynomial is the pth root of the sum of the pth powers of the absolute values of the coefficients of the polynomial.

INPUT:

- p - (positive integer or +infinity) the degree of the norm

EXAMPLES:

```
sage: R.<x> =RR[]
sage: f = x^6 + x^2 + -x^4 - 2*x^3
sage: f.norm(2)
2.64575131106459
sage: (sqrt(1^2 + 1^2 + (-1)^2 + (-2)^2)).n()
2.64575131106459

sage: f.norm(1)
5.00000000000000
sage: f.norm(infinity)
2.00000000000000

sage: f.norm(-1)
...
ValueError: The degree of the norm must be positive
```

TESTS:

```
sage: R.<x> = RR[]
sage: f = x^6 + x^2 + -x^4 -x^3
sage: f.norm(int(2))
2.00000000000000
```

AUTHORS:

- Didier Deshommes
- William Stein: fix bugs, add definition, etc.

ord()

This is the same as the valuation of self at p. See the documentation for `self.valuation`.

EXAMPLES:

```
sage: P,x=PolynomialRing(ZZ,'x').objgen()
sage: (x^2+x).ord(x+1)
1
```

padded_list()

Return list of coefficients of self up to (but not including) q^n.

Includes 0's in the list on the right so that the list has length n.

INPUT:

- n - (default: None); if given, an integer that is at least 0

EXAMPLES:

```
sage: x = polygen(QQ)
sage: f = 1 + x^3 + 23*x^5
sage: f.padded_list()
[1, 0, 0, 1, 0, 23]
sage: f.padded_list(10)
[1, 0, 0, 1, 0, 23, 0, 0, 0, 0]
sage: len(f.padded_list(10))
10
sage: f.padded_list(3)
[1, 0, 0]
sage: f.padded_list(0)
[]
sage: f.padded_list(-1)
...
ValueError: n must be at least 0
```

plot()

Return a plot of this polynomial.

INPUT:

- `xmin` - float
- `xmax` - float
- `*args, **kwds` - passed to either plot or point

OUTPUT: returns a graphic object.

EXAMPLES:

```
sage: x = polygen(GF(389))
sage: plot(x^2 + 1, rgbcolor=(0,0,1))
sage: x = polygen(QQ)
sage: plot(x^2 + 1, rgbcolor=(1,0,0))
```

polynomial()

Let var be one of the variables of the parent of self. This returns self viewed as a univariate polynomial in var over the polynomial ring generated by all the other variables of the parent.

For univariate polynomials, if var is the generator of the parent ring, we return this polynomial, otherwise raise an error.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: (x+1).polynomial(x)
x + 1
```

TESTS:

```
sage: x.polynomial(1)
...
ValueError: given variable is not the generator of parent.
```

prec()

Return the precision of this polynomial. This is always infinity, since polynomials are of infinite precision by definition (there is no big-oh).

EXAMPLES:

```
sage: x = polygen(ZZ)
sage: (x^5 + x + 1).prec()
+Infinity
sage: x.prec()
+Infinity
```

radical()

Returns the radical of self; over a field, this is the product of the distinct irreducible factors of self. (This is also sometimes called the "square-free part" of self, but that term is ambiguous; it is sometimes used to mean the quotient of self by its maximal square factor.)

EXAMPLES:

```
sage: P.<x> = ZZ[]
sage: t = (x^2-x+1)^3 * (3*x-1)^2
sage: t.radical()
3*x^3 - 4*x^2 + 4*x - 1
```

real_roots()

Return the real roots of this polynomial, without multiplicities.

Calls self.roots(ring=RR), unless this is a polynomial with floating-point real coefficients, in which case it calls self.roots().

EXAMPLES:

```
sage: x = polygen(ZZ)
sage: (x^2 - x - 1).real_roots()
[-0.618033988749895, 1.61803398874989]
```

TESTS:

```
sage: x = polygen(RealField(100))
sage: (x^2 - x - 1).real_roots()[0].parent()
    Real Field with 100 bits of precision
sage: x = polygen(RDF)
sage: (x^2 - x - 1).real_roots()[0].parent()
Real Double Field
```

resultant()

Returns the resultant of self and other.

INPUT:

- `other` - a polynomial

OUTPUT: an element of the base ring of the polynomial ring

Note: Implemented using PARI's `polresultant` function.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^3 + x + 1;  g = x^3 - x - 1
sage: r = f.resultant(g); r
-8
sage: r.parent() is QQ
True
```

We can also compute resultants over univariate and multivariate polynomial rings, provided that PARI's variable ordering requirements are respected. Usually, your resultants will work if you always ask for them in the variable `x`:

```
sage: R.<a> = QQ[]
sage: S.<x> = R[]
sage: f = x^2 + a; g = x^3 + a
sage: r = f.resultant(g); r
a^3 + a^2
sage: r.parent() is R
True
```

```
sage: R.<a, b> = QQ[]
sage: S.<x> = R[]
sage: f = x^2 + a; g = x^3 + b
sage: r = f.resultant(g); r
a^3 + b^2
sage: r.parent() is R
True
```

Unfortunately Sage does not handle PARI's variable ordering requirements gracefully, so the following fails:

```
sage: R.<x, y> = QQ[]
sage: S.<a> = R[]
sage: f = x^2 + a; g = y^3 + a
sage: f.resultant(g)
...
PariError: (8)
```

reverse()

Return polynomial but with the coefficients reversed.

EXAMPLES:

```
sage: R.<x> = ZZ[]; S.<y> = R[]
sage: f = y^3 + x*y -3*x; f
y^3 + x*y - 3*x
sage: f.reverse()
-3*x*y^3 + x*y^2 + 1
```

root_field()

Return the field generated by the roots of the irreducible polynomial self. The output is either a number field, relative number field, a quotient of a polynomial ring over a field, or the fraction field of the base ring.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: f = x^3 + x + 17
sage: f.root_field('a')
Number Field in a with defining polynomial x^3 + x + 17

sage: R.<x> = QQ['x']
sage: f = x - 3
sage: f.root_field('b')
Rational Field

sage: R.<x> = ZZ['x']
sage: f = x^3 + x + 17
sage: f.root_field('b')
Number Field in b with defining polynomial x^3 + x + 17

sage: y = QQ['x'].0
sage: L.<a> = NumberField(y^3-2)
sage: R.<x> = L['x']
sage: f = x^3 + x + 17
sage: f.root_field('c')
Number Field in c with defining polynomial x^3 + x + 17 over its base field

sage: R.<x> = PolynomialRing(GF(9,'a'))
sage: f = x^3 + x^2 + 8
sage: K.<alpha> = f.root_field(); K
```

```
Univariate Quotient Polynomial Ring in alpha over Finite Field in a of size 3^2 with modulus
sage: alpha^2 + 1
alpha^2 + 1
sage: alpha^3 + alpha^2
1

sage: R.<x> = QQ[]
sage: f = x^2
sage: K.<alpha> = f.root_field()
...
ValueError: polynomial must be irreducible
```

TESTS:

```
sage: (PolynomialRing(Integers(31),name='x').0+5).root_field('a')
Ring of integers modulo 31
```

roots()

Return the roots of this polynomial (by default, in the base ring of this polynomial).

INPUT:

- `ring` - the ring to find roots in
- `multiplicities` - bool (default: True) if True return list of pairs (r, n), where r is the root and n is the multiplicity. If False, just return the unique roots, with no information about multiplicities.
- `algorithm` - the root-finding algorithm to use. We attempt to select a reasonable algorithm by default, but this lets the caller override our choice.

By default, this finds all the roots that lie in the base ring of the polynomial. However, the ring parameter can be used to specify a ring to look for roots in.

If the polynomial and the output ring are both exact (integers, rationals, finite fields, etc.), then the output should always be correct (or raise an exception, if that case is not yet handled).

If the output ring is approximate (floating-point real or complex numbers), then the answer will be estimated numerically, using floating-point arithmetic of at least the precision of the output ring. If the polynomial is ill-conditioned, meaning that a small change in the coefficients of the polynomial will lead to a relatively large change in the location of the roots, this may give poor results. Distinct roots may be returned as multiple roots, multiple roots may be returned as distinct roots, real roots may be lost entirely (because the numerical estimate thinks they are complex roots). Note that polynomials with multiple roots are always ill-conditioned; there's a footnote at the end of the docstring about this.

If the output ring is a RealIntervalField or ComplexIntervalField of a given precision, then the answer will always be correct (or an exception will be raised, if a case is not implemented). Each root will be contained in one of the returned intervals, and the intervals will be disjoint. (The returned intervals may be of higher precision than the specified output ring.)

At the end of this docstring (after the examples) is a description of all the cases implemented in this function, and the algorithms used. That section also describes the possibilities for "algorithm=", for the cases where multiple algorithms exist.

EXAMPLES:

```
sage: x = QQ['x'].0
sage: f = x^3 - 1
sage: f.roots()
[(1, 1)]
sage: f.roots(ring=CC)    # note -- low order bits slightly different on ppc.
[(1.00000000000000, 1), (-0.500000000000000 - 0.86602540378443...*I, 1), (-0.500000000000000
sage: f = (x^3 - 1)^2
sage: f.roots()
[(1, 2)]
```

```
sage: f = -19*x + 884736
sage: f.roots()
[(884736/19, 1)]
sage: (f^20).roots()
[(884736/19, 20)]

sage: K.<z> = CyclotomicField(3)
sage: f = K.defining_polynomial()
sage: f.roots(ring=GF(7))
[(4, 1), (2, 1)]
sage: g = f.change_ring(GF(7))
sage: g.roots()
[(4, 1), (2, 1)]
sage: g.roots(multiplicities=False)
[4, 2]
```

An example over RR, which illustrates that only the roots in RR are returned:

```
sage: x = RR['x'].0
sage: f = x^3 -2
sage: f.roots()
[(1.25992104989487, 1)]
sage: f.factor()
(1.00000000000000*x - 1.25992104989487) * (1.00000000000000*x^2 + 1.25992104989487*x + 1.587
sage: x = RealField(100)['x'].0
sage: f = x^3 -2
sage: f.roots()
[(1.2599210498948731647672106073, 1)]

sage: x = CC['x'].0
sage: f = x^3 -2
sage: f.roots()
[(1.25992104989487, 1), (-0.62996052494743... - 1.09112363597172*I, 1), (-0.62996052494743..
sage: f.roots(algorithm='pari')
[(1.25992104989487, 1), (-0.629960524947437 - 1.09112363597172*I, 1), (-0.629960524947437 +
```

Another example showing that only roots in the base ring are returned:

```
sage: x = polygen(ZZ)
sage: f = (2*x-3) * (x-1) * (x+1)
sage: f.roots()
[(1, 1), (-1, 1)]
sage: f.roots(ring=QQ)
[(3/2, 1), (1, 1), (-1, 1)]
```

An example involving large numbers:

```
sage: x = RR['x'].0
sage: f = x^2 - 1e100
sage: f.roots()
[(-1.00000000000000e50, 1), (1.00000000000000e50, 1)]
sage: f = x^10 - 2*(5*x-1)^2
sage: f.roots(multiplicities=False)
[-1.6772670339941..., 0.19995479628..., 0.20004530611..., 1.5763035161844...]

sage: x = CC['x'].0
sage: i = CC.0
sage: f = (x - 1)*(x - i)
sage: f.roots(multiplicities=False) #random - this example is numerically rather unstable
[2.22044604925031e-16 + 1.00000000000000*I, 1.00000000000000 + 8.32667268468867e-17*I]
```

```
sage: g=(x-1.33+1.33*i)*(x-2.66-2.66*i)
sage: g.roots(multiplicities=False)
[1.33000000000000 - 1.33000000000000*I, 2.66000000000000 + 2.66000000000000*I]
```

A purely symbolic roots example:

```
sage: X = var('X')
sage: f = expand((X-1)*(X-I)^3*(X^2 - sqrt(2))); f
-sqrt(2)*X^4 + I*sqrt(2) + X^6 - (3*I - 1)*X^5 + (3*I - 3)*X^4 + (3*I + 1)*sqrt(2)*X^3 + (I
sage: print f.roots()
[(I, 3), (-2^(1/4), 1), (2^(1/4), 1), (1, 1)]
```

A couple of examples where the base ring doesn't have a factorization algorithm (yet). Note that this is currently done via naive enumeration, so could be very slow:

```
sage: R = Integers(6)
sage: S.<x> = R['x']
sage: p = x^2-1
sage: p.roots()
...
NotImplementedError: root finding with multiplicities for this polynomial not implemented (t
sage: p.roots(multiplicities=False)
[1, 5]
sage: R = Integers(9)
sage: A = PolynomialRing(R, 'y')
sage: y = A.gen()
sage: f = 10*y^2 - y^3 - 9
sage: f.roots(multiplicities=False)
[0, 1, 3, 6]
```

An example over the complex double field (where root finding is fast, thanks to numpy):

```
sage: R.<x> = CDF[]
sage: f = R.cyclotomic_polynomial(5); f
1.0*x^4 + 1.0*x^3 + 1.0*x^2 + 1.0*x + 1.0
sage: f.roots(multiplicities=False)   # slightly random
[0.309016994375 + 0.951056516295*I, 0.309016994375 - 0.951056516295*I, -0.809016994375 + 0.5
sage: [z^5 for z in f.roots(multiplicities=False)]     # slightly random
[1.0 - 2.44929359829e-16*I, 1.0 + 2.44929359829e-16*I, 1.0 - 4.89858719659e-16*I, 1.0 + 4.89
sage: f = CDF['x']([1,2,3,4]); f
4.0*x^3 + 3.0*x^2 + 2.0*x + 1.0
sage: r = f.roots(multiplicities=False)
sage: [f(a) for a in r]    # slightly random
[2.55351295664e-15, -4.4408920985e-16 - 2.08166817117e-16*I, -4.4408920985e-16 + 2.081668171
```

Another example over RDF:

```
sage: x = RDF['x'].0
sage: ((x^3 -1)).roots()
[(1.0, 1)]
sage: ((x^3 -1)).roots(multiplicities=False)
[1.0]
```

Another examples involving the complex double field:

```
sage: x = CDF['x'].0
sage: i = CDF.0
sage: f = x^3 + 2*i; f
1.0*x^3 + 2.0*I
sage: f.roots()
[(-1.09112363597 - 0.629960524947*I, 1),
 (... + 1.25992104989*I, 1),
```

```
 (1.09112363597 - 0.629960524947*I, 1)]
sage: f.roots(multiplicities=False)
[-1.09112363597 - 0.629960524947*I,
 ... + 1.25992104989r*I,
 1.09112363597 - 0.629960524947*I]
sage: [f(z) for z in f.roots(multiplicities=False)] # random, too close to 0
[-2.56337823492e-15 - 6.66133814775e-15*I,
 3.96533069372e-16 + 1.99840144433e-15*I,
 4.19092485179e-17 - 8.881784197e-16*I]
sage: f = i*x^3 + 2; f
1.0*I*x^3 + 2.0
sage: f.roots()
[(-1.09112363597 + 0.629960524947*I, 1),
 (... - 1.25992104989*I, 1),
 (1.09112363597 + 0.629960524947*I, 1)]
sage: f(f.roots()[0][0]) # random, too close to 0
-2.56337823492e-15 - 6.66133814775e-15*I
```

Examples using real root isolation:

```
sage: x = polygen(ZZ)
sage: f = x^2 - x - 1
sage: f.roots()
[]
sage: f.roots(ring=RIF)
[(-0.618033988749894848204586834365638117720309179805762862135448622?, 1), (1.618033988749894848204586834365638117720309179805762862135448622?, 1)]
sage: f.roots(ring=RIF, multiplicities=False)
[-0.618033988749894848204586834365638117720309179805762862135448622?, 1.618033988749894848204586834365638117720309179805762862135448622?]
sage: f.roots(ring=RealIntervalField(150))
[(-0.618033988749894848204586834365638117720309179805762862135448622?, 1), (1.6180339887498
sage: f.roots(ring=AA)
[(-0.6180339887498950?, 1), (1.6180339887498950?, 1)]
sage: f = f^2 * (x - 1)
sage: f.roots(ring=RIF)
[(-0.618033988749894848204586834365638117720309179805762862135448622?, 2), (1.000000000000000000000000000000000?, 1), (1.61803
sage: f.roots(ring=RIF, multiplicities=False)
[-0.618033988749894848204586834365638117720309179805762862135448622?, 1.000000000000000000000000000000000?, 1.6180339887498948
```

Examples using complex root isolation:

```
sage: x = polygen(ZZ)
sage: p = x^5 - x - 1
sage: p.roots()
[]
sage: p.roots(ring=CIF)
[(1.167303978261419?, 1), (-0.764884433600585? - 0.352471546031727?*I, 1), (-0.764884433600
sage: p.roots(ring=ComplexIntervalField(200))
[(1.16730397826141868425604589985484218072056037152548903914000 82?, 1), (-0.7648844336005847
sage: rts = p.roots(ring=QQbar); rts
[(1.167303978261419?, 1), (-0.7648844336005847? - 0.352471546031726 3?*I, 1), (-0.764884433600
sage: p.roots(ring=AA)
[(1.167303978261419?, 1)]
sage: p = (x - rts[4][0])^2 * (3*x^2 + x + 1)
sage: p.roots(ring=QQbar)
[(-0.1666666666666667? - 0.552770798392567?*I, 1), (-0.1666666666666667? + 0.552770798392567
sage: p.roots(ring=CIF)
[(-0.1666666666666667? - 0.552770798392567?*I, 1), (-0.1666666666666667? + 0.552770798392567
```

Note that coefficients in a number field with defining polynomial $x^2 + 1$ are considered to be Gaussian rationals (with the generator mapping to +I), if you ask for complex roots.

```
sage: K.<im> = NumberField(x^2 + 1)
sage: y = polygen(K)
sage: p = y^4 - 2 - im
sage: p.roots(ring=CC)
[(-1.2146389322441... - 0.14142505258239...*I, 1), (-0.14142505258239... + 1.2146389322441...
sage: p = p^2 * (y^2 - 2)
sage: p.roots(ring=CIF)
[(-1.414213562373095?, 1), (1.414213562373095?, 1), (-1.214638932244183? - 0.14142505258239
```

There are many combinations of floating-point input and output types that work. (Note that some of them are quite pointless... there's no reason to use high-precision input and output, and still use numpy to find the roots.)

```
sage: rflds = (RR, RDF, RealField(100))
sage: cflds = (CC, CDF, ComplexField(100))
sage: def cross(a, b):
...       return list(cartesian_product_iterator([a, b]))
sage: flds = cross(rflds, rflds) + cross(rflds, cflds) + cross(cflds, cflds)
sage: for (fld_in, fld_out) in flds:
...       x = polygen(fld_in)
...       f = x^3 - fld_in(2)
...       x2 = polygen(fld_out)
...       f2 = x2^3 - fld_out(2)
...       for algo in (None, 'pari', 'numpy'):
...           rts = f.roots(ring=fld_out, multiplicities=False)
...           if fld_in == fld_out and algo is None:
...               print fld_in, rts
...           for rt in rts:
...               assert(abs(f2(rt)) <= 1e-10)
...               assert(rt.parent() == fld_out)
Real Field with 53 bits of precision [1.25992104989487]
Real Double Field [1.25992104989]
Real Field with 100 bits of precision [1.2599210498948731647672106073]
Complex Field with 53 bits of precision [1.25992104989487, -0.62996052494743... - 1.09112363
Complex Double Field [1.25992104989, -0.62996052494... - 1.09112363597*I, -0.62996052494...
Complex Field with 100 bits of precision [1.2599210498948731647672106073, -0.629960524947436
```

Note that we can find the roots of a polynomial with algebraic coefficients:

```
sage: rt2 = sqrt(AA(2))
sage: rt3 = sqrt(AA(3))
sage: x = polygen(AA)
sage: f = (x - rt2) * (x - rt3); f
    x^2 - 3.146264369941973?*x + 2.449489742783178?
sage: rts = f.roots(); rts
[(1.414213562373095?, 1), (1.732050807568878?, 1)]
sage: rts[0][0] == rt2
True
sage: f.roots(ring=RealIntervalField(150))
[(1.41421356237309504880168872420969807856967187537694807317667973 8?, 1), (1.732050807568877
```

We can handle polynomials with huge coefficients.

This number doesn't even fit in an IEEE double-precision float, but RR and CC allow a much larger range of floating-point numbers:

```
sage: bigc = 2^1500
sage: CDF(bigc)
+infinity
sage: CC(bigc)
3.50746621104340e451
```

Polynomials using such large coefficients can't be handled by numpy, but pari can deal with them:

```
sage: x = polygen(QQ)
sage: p = x + bigc
sage: p.roots(ring=RR, algorithm='numpy')
...
ValueError: array must not contain infs or NaNs
sage: p.roots(ring=RR, algorithm='pari')
[(-3.50746621104340e451, 1)]
sage: p.roots(ring=AA)
[(-3.5074662110434039?e451, 1)]
sage: p.roots(ring=QQbar)
[(-3.5074662110434039?e451, 1)]
sage: p = bigc*x + 1
sage: p.roots(ring=RR)
[(0.000000000000000, 1)]
sage: p.roots(ring=AA)
[(-2.8510609648967059?e-452, 1)]
sage: p.roots(ring=QQbar)
[(-2.8510609648967059?e-452, 1)]
sage: p = x^2 - bigc
sage: p.roots(ring=RR)
[(-5.92238652153286e225, 1), (5.92238652153286e225, 1)]
sage: p.roots(ring=QQbar)
[(-5.9223865215328558?e225, 1), (5.9223865215328558?e225, 1)]
```

Algorithms used:

For brevity, we will use RR to mean any RealField of any precision; similarly for RIF, CC, and CIF. Since Sage has no specific implementation of Gaussian rationals (or of number fields with embedding, at all), when we refer to Gaussian rationals below we will accept any number field with defining polynomial $x^2 + 1$, mapping the field generator to +I.

We call the base ring of the polynomial K, and the ring given by the ring= argument L. (If ring= is not specified, then L is the same as K.)

If K and L are floating-point (RDF, CDF, RR, or CC), then a floating-point root-finder is used. If L has precision 53 bits or less (RDF and CDF both have precision exactly 53 bits, as do the default RR=RealField() and CC=ComplexField()) then we default to using numpy's roots(); otherwise, we use Pari's polroots(). This choice can be overridden with algorithm='pari' or algorithm='numpy'. If the algorithm is unspecified and numpy's roots() algorithm fails, then we fall back to pari (numpy will fail if some coefficient is infinite, for instance).

If L is AA or RIF, and K is ZZ, QQ, or AA, then the root isolation algorithm sage.rings.polynomial.real_roots.real_roots() is used. (You can call real_roots() directly to get more control than this method gives.)

If L is QQbar or CIF, and K is ZZ, QQ, AA, QQbar, or the Gaussian rationals, then the root isolation algorithm sage.rings.polynomial.complex_roots.complex_roots() is used. (You can call complex_roots() directly to get more control than this method gives.)

If L is AA and K is QQbar or the Gaussian rationals, then complex_roots() is used (as above) to find roots in QQbar, then these roots are filtered to select only the real roots.

If L is floating-point and K is not, then we attempt to change the polynomial ring to L (using .change_ring()) (or, if L is complex and K is not, to the corresponding real field). Then we use either Pari or numpy as specified above.

For all other cases where K is different than L, we just use .change_ring(L) and proceed as below.

The next method, which is used if K is an integral domain, is to attempt to factor the polynomial. If this succeeds, then for every degree-one factor a*x+b, we add -b/a as a root (as long as this quotient is actually in the desired ring).

If factoring over K is not implemented (or K is not an integral domain), and K is finite, then we find the

roots by enumerating all elements of K and checking whether the polynomial evaluates to zero at that value.

Note: We mentioned above that polynomials with multiple roots are always ill-conditioned; if your input is given to n bits of precision, you should not expect more than n/k good bits for a k-fold root. (You can get solutions that make the polynomial evaluate to a number very close to zero; basically the problem is that with a multiple root, there are many such numbers, and it's difficult to choose between them.)

To see why this is true, consider the naive floating-point error analysis model where you just pretend that all floating-point numbers are somewhat imprecise - a little 'fuzzy', if you will. Then the graph of a floating-point polynomial will be a fuzzy line. Consider the graph of $(x-1)^3$; this will be a fuzzy line with a horizontal tangent at $x = 1$, $y = 0$. If the fuzziness extends up and down by about j, then it will extend left and right by about cube_root(j).

TESTS:

```
sage: K.<zeta> = CyclotomicField(2)
sage: R.<x> = K[]
sage: factor(x^3-1)
(x - 1) * (x^2 + x + 1)
```

shift()

Returns this polynomial multiplied by the power x^n. If n is negative, terms below x^n will be discarded. Does not change this polynomial (since polynomials are immutable).

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: p = x^2 + 2*x + 4
sage: p.shift(0)
 x^2 + 2*x + 4
sage: p.shift(-1)
 x + 2
sage: p.shift(-5)
 0
sage: p.shift(2)
 x^4 + 2*x^3 + 4*x^2
```

One can also use the infix shift operator:

```
sage: f = x^3 + x
sage: f >> 2
x
sage: f << 2
x^5 + x^3
```

TESTS:

```
sage: p = R(0)
sage: p.shift(3).is_zero()
True
sage: p.shift(-3).is_zero()
True
```

AUTHORS:

- David Harvey (2006-08-06)
- Robert Bradshaw (2007-04-18): Added support for infix operator.

square()

Returns the square of this polynomial.

TODO:

- This is just a placeholder; for now it just uses ordinary multiplication. But generally speaking, squaring is faster than ordinary multiplication, and it's frequently used, so subclasses may choose to provide a specialised squaring routine.

•Perhaps this even belongs at a lower level? ring_element or something?

AUTHORS:

•David Harvey (2006-09-09)

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^3 + 1
sage: f.square()
x^6 + 2*x^3 + 1
sage: f*f
x^6 + 2*x^3 + 1
```

squarefree_decomposition()

Return the square-free decomposition of self. This is a partial factorization of self into square-free, relatively prime polynomials.

This is the straightforward algorithm, using only polynomial GCD and polynomial division. Faster algorithms exist. The algorithm comes from the Wikipedia article, "Square-free polynomial".

EXAMPLES:

```
sage: x = polygen(QQ)
sage: p = 37 * (x-1)^3 * (x-2)^3 * (x-1/3)^7 * (x-3/7)
sage: p.squarefree_decomposition()
(37*x - 111/7) * (x^2 - 3*x + 2)^3 * (x - 1/3)^7
sage: p = 37 * (x-2/3)^2
sage: p.squarefree_decomposition()
(37) * (x - 2/3)^2
sage: x = polygen(GF(3))
sage: x.squarefree_decomposition()
...
NotImplementedError: Squarefree decomposition not implemented for Univariate Polynomial Ring
```

subs()

Identical to self(*x).

See the docstring for `self.__call__`.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^3 + x - 3
sage: f.subs(x=5)
127
sage: f.subs(5)
127
sage: f.subs({x:2})
7
sage: f.subs({})
x^3 + x - 3
sage: f.subs({'x':2})
...
TypeError: keys do not match self's parent
```

substitute()

Identical to self(*x).

See the docstring for `self.__call__`.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^3 + x - 3
```

```
sage: f.subs(x=5)
127
sage: f.subs(5)
127
sage: f.subs({x:2})
7
sage: f.subs({})
x^3 + x - 3
sage: f.subs({'x':2})
...
TypeError: keys do not match self's parent
```

truncate()

Returns the polynomial of degree ' < n' which is equivalent to self modulo x^n.

EXAMPLES:

```
sage: R.<x> = ZZ[]; S.<y> = R[]
sage: f = y^3 + x*y -3*x; f
y^3 + x*y - 3*x
sage: f.truncate(2)
x*y - 3*x
sage: f.truncate(1)
-3*x
sage: f.truncate(0)
0
```

valuation()

If $f = a_r x^r + a_{r+1} x^{r+1} + \cdots$, with a_r nonzero, then the valuation of f is r. The valuation of the zero polynomial is ∞.

If a prime (or non-prime) p is given, then the valuation is the largest power of p which divides self.

The valuation at ∞ is -self.degree().

EXAMPLES:

```
sage: P,x=PolynomialRing(ZZ,'x').objgen()
sage: (x^2+x).valuation()
1
sage: (x^2+x).valuation(x+1)
1
sage: (x^2+1).valuation()
0
sage: (x^3+1).valuation(infinity)
-3
sage: P(0).valuation()
+Infinity
```

variable_name()

Return name of variable used in this polynomial as a string.

OUTPUT: string

EXAMPLES:

```
sage: R.<t> = QQ[]
sage: f = t^3 + 3/2*t + 5
sage: f.variable_name()
't'
```

variables()

Returns the list of variables occurring in this polynomial.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: x.variables()
(x,)
```

A constant polynomial has no variables.

```
sage: R(2).variables()
()
```

class PolynomialBaseringInjection()

This class is used for conversion from a ring to a polynomial over that ring.

It calls the _new_constant_poly method on the generator, which should be optimized for a particular polynomial type. Technically, it should be a method of the polynomial ring, but few polynomial rings are cython classes.

section()

TESTS:

```
sage: from sage.rings.polynomial.polynomial_element import PolynomialBaseringInjection
sage: m = PolynomialBaseringInjection(RDF, RDF['x'])
sage: m.section()
Generic map:
  From: Univariate Polynomial Ring in x over Real Double Field
  To:   Real Double Field
sage: type(m.section())
<type 'sage.rings.polynomial.polynomial_element.ConstantPolynomialSection'>
```

class Polynomial_generic_dense()

A generic dense polynomial.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(PolynomialRing(QQ,'y'))
sage: f = x^3 - x + 17
sage: type(f)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: loads(f.dumps()) == f
True
```

constant_coefficient()

Return the constant coefficient of this polynomial.

OUTPUT: element of base ring

EXAMPLES: sage: R.<t> = QQ[] sage: S.<x> = R[] sage: f = x*t + x + t sage: f.constant_coefficient() t

degree()

EXAMPLES:

```
sage: R.<x> = RDF[]
sage: f = (1+2*x^7)^5
sage: f.degree()
35
```

list()

Return a new copy of the list of the underlying elements of self.

EXAMPLES:

```
sage: R.<x> = GF(17)[]
sage: f = (1+2*x)^3 + 3*x; f
8*x^3 + 12*x^2 + 9*x + 1
sage: f.list()
[1, 9, 12, 8]
```

shift()

Returns this polynomial multiplied by the power x^n. If n is negative, terms below x^n will be discarded. Does not change this polynomial.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(PolynomialRing(QQ,'y'), 'x')
sage: p = x^2 + 2*x + 4
sage: type(p)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
sage: p.shift(0)
 x^2 + 2*x + 4
sage: p.shift(-1)
 x + 2
sage: p.shift(2)
 x^4 + 2*x^3 + 4*x^2
```

TESTS:

```
sage: p = R(0)
sage: p.shift(3).is_zero()
True
sage: p.shift(-3).is_zero()
True
```

AUTHORS:

- David Harvey (2006-08-06)

truncate()

Returns the polynomial of degree ' < n' which is equivalent to self modulo x^n.

EXAMPLES:

```
sage: S.<q> = QQ['t']['q']
sage: f = (1+q^10+q^11+q^12).truncate(11); f
q^10 + 1
sage: f = (1+q^10+q^100).truncate(50); f
q^10 + 1
sage: f.degree()
10
sage: f = (1+q^10+q^100).truncate(500); f
q^100 + q^10 + 1
```

TESTS:

Make sure we're not actually testing a specialized implementation.

```
sage: type(f)
<type 'sage.rings.polynomial.polynomial_element.Polynomial_generic_dense'>
```

is_Polynomial()

Return True if f is of type univariate polynomial.

INPUT:

- f - an object

EXAMPLES:

```
sage: from sage.rings.polynomial.polynomial_element import is_Polynomial
sage: R.<x> = ZZ[]
sage: is_Polynomial(x^3 + x + 1)
True
sage: S.<y> = R[]
```

```
sage: f = y^3 + x*y -3*x; f
y^3 + x*y - 3*x
sage: is_Polynomial(f)
True
```

However this function does not return True for genuine multivariate polynomial type objects or symbolic polynomials, since those are not of the same data type as univariate polynomials:

```
sage: R.<x,y> = QQ[]
sage: f = y^3 + x*y -3*x; f
y^3 + x*y - 3*x
sage: is_Polynomial(f)
False
sage: var('x,y')
(x, y)
sage: f = y^3 + x*y -3*x; f
y^3 + x*y - 3*x
sage: is_Polynomial(f)
False
```

make_generic_polynomial()

28.3 Quotients of Univariate Polynomial Rings

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: S = R.quotient(x**3-3*x+1, 'alpha')
sage: S.gen()**2 in S
True
sage: x in S
True
sage: S.gen() in R
False
sage: 1 in S
True
```

PolynomialQuotientRing(*ring, polynomial, names=None*)

Create a quotient of a polynomial ring.

INPUT:

- `ring` - a univariate polynomial ring in one variable.
- `polynomial` - element
- `names` - (optional) name for the variable

OUTPUT: Creates the quotient ring R/I, where R is the ring and I is the principal ideal generated by the polynomial.

EXAMPLES:

We create the quotient ring $\mathbf{Z}[x]/(x^3+7)$, and demonstrate many basic functions with it:

```
sage: Z = IntegerRing()
sage: R = PolynomialRing(Z,'x'); x = R.gen()
sage: S = R.quotient(x^3 + 7, 'a'); a = S.gen()
sage: S
```

```
Univariate Quotient Polynomial Ring in a over Integer Ring with modulus x^3 + 7
sage: a^3
-7
sage: S.is_field()
False
sage: a in S
True
sage: x in S
True
sage: a in R
False
sage: S.polynomial_ring()
Univariate Polynomial Ring in x over Integer Ring
sage: S.modulus()
x^3 + 7
sage: S.degree()
3
```

We create the "iterated" polynomial ring quotient

$$R = (\mathbf{F}_2[y]/(y^2+y+1))[x]/(x^3-5).$$

```
sage: A.<y> = PolynomialRing(GF(2)); A
Univariate Polynomial Ring in y over Finite Field of size 2 (using NTL)
sage: B = A.quotient(y^2 + y + 1, 'y2'); print B
Univariate Quotient Polynomial Ring in y2 over Finite Field of size 2 with modulus y^2 + y + 1
sage: C = PolynomialRing(B, 'x'); x=C.gen(); print C
Univariate Polynomial Ring in x over Univariate Quotient Polynomial Ring in y2 over Finite Field
sage: R = C.quotient(x^3 - 5); print R
Univariate Quotient Polynomial Ring in xbar over Univariate Quotient Polynomial Ring in y2 over
```

Next we create a number field, but viewed as a quotient of a polynomial ring over $\mathbf{Q}$:

```
sage: R = PolynomialRing(RationalField(), 'x'); x = R.gen()
sage: S = R.quotient(x^3 + 2*x - 5, 'a')
sage: S
Univariate Quotient Polynomial Ring in a over Rational Field with modulus x^3 + 2*x - 5
sage: S.is_field()
True
sage: S.degree()
3
```

There are conversion functions for easily going back and forth between quotients of polynomial rings over $\mathbf{Q}$ and number fields:

```
sage: K = S.number_field(); K
Number Field in a with defining polynomial x^3 + 2*x - 5
sage: K.polynomial_quotient_ring()
Univariate Quotient Polynomial Ring in a over Rational Field with modulus x^3 + 2*x - 5
```

The leading coefficient must be a unit (but need not be 1).

```
sage: R = PolynomialRing(Integers(9), 'x'); x = R.gen()
sage: S = R.quotient(2*x^4 + 2*x^3 + x + 2, 'a')
sage: S = R.quotient(3*x^4 + 2*x^3 + x + 2, 'a')
...
TypeError: polynomial must have unit leading coefficient
```

Another example:

```
sage: R.<x> = PolynomialRing(IntegerRing())
sage: f = x^2 + 1
sage: R.quotient(f)
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1
```

class **PolynomialQuotientRing_domain**(*ring, polynomial, name=None*)

EXAMPLES:

```
sage: R.<x> = PolynomialRing(ZZ)
sage: S.<xbar> = R.quotient(x^2 + 1)
sage: S
Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 + 1
sage: loads(S.dumps()) == S
True
sage: loads(xbar.dumps()) == xbar
True
```

field_extension(*names*)

Takes a polynomial quotient ring, and returns a tuple with three elements: the NumberField defined by the same polynomial quotient ring, a homomorphism from its parent to the NumberField sending the generators to one another, and the inverse isomorphism.

OUTPUT:

- field
- homomorphism from self to field
- homomorphism from field to self

EXAMPLES:

```
sage: R.<x> = PolynomialRing(Rationals())
sage: S.<alpha> = R.quotient(x^3-2)
sage: F.<b>, f, g = S.field_extension()
sage: F
Number Field in b with defining polynomial x^3 - 2
sage: a = F.gen()
sage: f(alpha)
b
sage: g(a)
alpha
```

Note that the parent ring must be an integral domain:

```
sage: R.<x> = GF(25,'f25')['x']
sage: S.<a> = R.quo(x^3 - 2)
sage: F, g, h = S.field_extension('b')
...
AttributeError: 'PolynomialQuotientRing_generic' object has no attribute 'field_extension'
```

Over a finite field, the corresponding field extension is not a number field:

```
sage: R.<x> = GF(25, 'a')['x']
sage: S.<a> = R.quo(x^3 + 2*x + 1)
sage: F, g, h = S.field_extension('b')
sage: h(F.0^2 + 3)
a^2 + 3
sage: g(x^2 + 2)
b^2 + 2
```

We do an example involving a relative number field:

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3 - 2)
sage: S.<X> = K['X']
sage: Q.<b> = S.quo(X^3 + 2*X + 1)
sage: Q.field_extension('b')
(Number Field in b with defining polynomial X^3 + 2*X + 1 over its base field, ...
  Defn: b |--> b, Relative number field morphism:
  From: Number Field in b with defining polynomial X^3 + 2*X + 1 over its base field
  To:   Univariate Quotient Polynomial Ring in b over Number Field in a with defining polync
  Defn: b |--> b
        a |--> a)
```

We slightly change the example above so it works.

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3 - 2)
sage: S.<X> = K['X']
sage: f = (X+a)^3 + 2*(X+a) + 1
sage: f
X^3 + 3*a*X^2 + (3*a^2 + 2)*X + 2*a + 3
sage: Q.<z> = S.quo(f)
sage: F.<w>, g, h = Q.field_extension()
sage: c = g(z)
sage: f(c)
0
sage: h(g(z))
z
sage: g(h(w))
w
```

AUTHORS:

- Craig Citro (2006-08-07)
- William Stein (2006-08-06)

is_finite()

Return whether or not this quotient ring is finite.

EXAMPLES:

```
sage: R.<x> = ZZ[]
sage: R.quo(1).is_finite()
True
sage: R.quo(x^3-2).is_finite()
False

sage: R.<x> = GF(9,'a')[]
sage: R.quo(2*x^3+x+1).is_finite()
True
sage: R.quo(2).is_finite()
True
```

class PolynomialQuotientRing_field(*ring, polynomial, name=None*)

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<xbar> = R.quotient(x^2 + 1)
sage: S
Univariate Quotient Polynomial Ring in xbar over Rational Field with modulus x^2 + 1
sage: loads(S.dumps()) == S
True
```

```
sage: loads(xbar.dumps()) == xbar
True
```

base_field()
Alias for base_ring, when we're defined over a field.

complex_embeddings(*prec=53*)
Return all homomorphisms of this ring into the approximate complex field with precision prec.
EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^5 + x + 17
sage: k = R.quotient(f)
sage: v = k.complex_embeddings(100)
sage: [phi(k.0^2) for phi in v]
[2.9757207403766761469671194565, -2.4088994371613850098316292196 + 1.9025410530350528612407
```

class **PolynomialQuotientRing_generic**(*ring, polynomial, name=None*)
Quotient of a univariate polynomial ring by an ideal.
EXAMPLES:

```
sage: R.<x> = PolynomialRing(Integers(8)); R
Univariate Polynomial Ring in x over Ring of integers modulo 8
sage: S.<xbar> = R.quotient(x^2 + 1); S
Univariate Quotient Polynomial Ring in xbar over Ring of integers modulo 8 with modulus x^2 + 1
```

We demonstrate object persistence.

```
sage: loads(S.dumps()) == S
True
sage: loads(xbar.dumps()) == xbar
True
```

We create some sample homomorphisms;

```
sage: R.<x> = PolynomialRing(ZZ)
sage: S = R.quo(x^2-4)
sage: f = S.hom([2])
sage: f
Ring morphism:
  From: Univariate Quotient Polynomial Ring in xbar over Integer Ring with modulus x^2 - 4
  To:   Integer Ring
  Defn: xbar |--> 2
sage: f(x)
2
sage: f(x^2 - 4)
0
sage: f(x^2)
4
```

base_ring()
Return the base ring of the polynomial ring, of which this ring is a quotient.
EXAMPLES:
The base ring of $\mathbf{Z}[z]/(z^3+z^2+z+1)$ is $\mathbf{Z}$.

```
sage: R.<z> = PolynomialRing(ZZ)
sage: S.<beta> = R.quo(z^3 + z^2 + z + 1)
```

```
sage: S.base_ring()
Integer Ring
```

Next we make a polynomial quotient ring over S and ask for its basering.

```
sage: T.<t> = PolynomialRing(S)
sage: W = T.quotient(t^99 + 99)
sage: W.base_ring()
Univariate Quotient Polynomial Ring in beta over Integer Ring with modulus z^3 + z^2 + z + 1
```

characteristic()

Return the characteristic of this quotient ring.

This is always the same as the characteristic of the base ring.

EXAMPLES:

```
sage: R.<z> = PolynomialRing(ZZ)
sage: S.<a> = R.quo(z - 19)
sage: S.characteristic()
0
sage: R.<x> = PolynomialRing(GF(9,'a'))
sage: S = R.quotient(x^3 + 1)
sage: S.characteristic()
3
```

degree()

Return the degree of this quotient ring. The degree is the degree of the polynomial that we quotiented out by.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(GF(3))
sage: S = R.quotient(x^2005 + 1)
sage: S.degree()
2005
```

discriminant(*v=None*)

Return the discriminant of this ring over the base ring. This is by definition the discriminant of the polynomial that we quotiented out by.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^3 + x^2 + x + 1)
sage: S.discriminant()
-16
sage: S = R.quotient((x + 1) * (x + 1))
sage: S.discriminant()
0
```

The discriminant of the quotient polynomial ring need not equal the discriminant of the corresponding number field, since the discriminant of a number field is by definition the discriminant of the ring of integers of the number field:

```
sage: S = R.quotient(x^2 - 8)
sage: S.number_field().discriminant()
8
sage: S.discriminant()
32
```

gen(*n=0*)

Return the generator of this quotient ring. This is the equivalence class of the image of the generator of the polynomial ring.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2 - 8, 'gamma')
sage: S.gen()
gamma
```

is_field()

Return whether or not this quotient ring is a field.

EXAMPLES:

```
sage: R.<z> = PolynomialRing(ZZ)
sage: S = R.quo(z^2-2)
sage: S.is_field()
False
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2 - 2)
sage: S.is_field()
True
```

krull_dimension()

modulus()

Return the polynomial modulus of this quotient ring.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(GF(3))
sage: S = R.quotient(x^2 - 2)
sage: S.modulus()
x^2 + 1
```

ngens()

Return the number of generators of this quotient ring over the base ring. This function always returns 1.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<y> = PolynomialRing(R)
sage: T.<z> = S.quotient(y + x)
sage: T
Univariate Quotient Polynomial Ring in z over Univariate Polynomial Ring in x over Rational
sage: T.ngens()
1
```

number_field()

Return the number field isomorphic to this quotient polynomial ring, if possible.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<alpha> = R.quotient(x^29 - 17*x - 1)
sage: K = S.number_field()
sage: K
Number Field in alpha with defining polynomial x^29 - 17*x - 1
sage: alpha = K.gen()
sage: alpha^29
17*alpha + 1
```

order()

Return the number of elements of this quotient ring.

EXAMPLES:

```
sage: F1.<a> = GF(2^7)
sage: P1.<x> = F1[]
sage: F2 = F1.extension(x^2+x+1, 'u')
sage: F2.order()
16384

sage: F1 = QQ
sage: P1.<x> = F1[]
sage: F2 = F1.extension(x^2+x+1, 'u')
sage: F2.order()
+Infinity
```

polynomial_ring()

Return the polynomial ring of which this ring is the quotient.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S = R.quotient(x^2-2)
sage: S.polynomial_ring()
Univariate Polynomial Ring in x over Rational Field
```

random_element()

Return a random element of this quotient ring.

EXAMPLES:

```
sage: F1.<a> = GF(2^7)
sage: P1.<x> = F1[]
sage: F2 = F1.extension(x^2+x+1, 'u')
sage: F2.random_element()
(a^6 + 1)*u + a^5 + a^4 + a^3 + 1
```

is_PolynomialQuotientRing(x)

28.4 Elements of Quotients of Univariate Polynomial Rings

EXAMPLES: We create a quotient of a univariate polynomial ring over $\mathbf{Z}$.

```
sage: R.<x> = ZZ[]
sage: S.<a> = R.quotient(x^3 + 3*x -1)
sage: 2 * a^3
-6*a + 2
```

Next we make a univeriate polynomial ring over $\mathbf{Z}[x]/(x^3 + 3x - 1)$.

```
sage: S.<y> = S[]
```

And, we quotient out that by $y^2 + a$.

```
sage: T.<z> = S.quotient(y^2+a)
```

In the quotient z^2 is $-a$.

```
sage: z^2
-a
```

And since $a^3 = -3x + 1$, we have:

```
sage: z^6
3*a - 1
```

```
sage: R.<x> = PolynomialRing(Integers(9))
sage: S.<a> = R.quotient(x^4 + 2*x^3 + x + 2)
sage: a^100
7*a^3 + 8*a + 7
```

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3-2)
sage: a
a
sage: a^3
2
```

For the purposes of comparison in Sage the quotient element a^3 is equal to x^3. This is because when the comparison is performed, the right element is coerced into the parent of the left element, and x^3 coerces to a^3.

```
sage: a == x
True
sage: a^3 == x^3
True
sage: x^3
x^3
sage: S(x^3)
2
```

AUTHORS:

- William Stein

class **PolynomialQuotientRingElement** (*parent, polynomial, check=True*)

Element of a quotient of a polynomial ring.

charpoly (*var*)

The characteristic polynomial of this element, which is by definition the characteristic polynomial of right multiplication by this element.

INPUT:

•`var` - string - the variable name

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quo(x^3 -389*x^2 + 2*x - 5)
sage: a.charpoly('X')
X^3 - 389*X^2 + 2*X - 5
```

fcp (*var='x'*)

Return the factorization of the characteristic polynomial of this element.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 -389*x^2 + 2*x - 5)
sage: a.fcp('x')
x^3 - 389*x^2 + 2*x - 5
```

```
sage: S(1).fcp('y')
(y - 1)^3
```

field_extension(*names*)

Given a polynomial with base ring a quotient ring, return a 3-tuple: a number field defined by the same polynomial, a homomorphism from its parent to the number field sending the generators to one another, and the inverse isomorphism.

INPUT:

- names - name of generator of output field

OUTPUT:

- field
- homomorphism from self to field
- homomorphism from field to self

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<alpha> = R.quotient(x^3-2)
sage: F.<a>, f, g = alpha.field_extension()
sage: F
Number Field in a with defining polynomial x^3 - 2
sage: a = F.gen()
sage: f(alpha)
a
sage: g(a)
alpha
```

Over a finite field, the corresponding field extension is not a number field:

```
sage: R.<x> = GF(25,'b')['x']
sage: S.<a> = R.quo(x^3 + 2*x + 1)
sage: F.<b>, g, h = a.field_extension()
sage: h(b^2 + 3)
a^2 + 3
sage: g(x^2 + 2)
b^2 + 2
```

We do an example involving a relative number field:

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3-2)
sage: S.<X> = K['X']
sage: Q.<b> = S.quo(X^3 + 2*X + 1)
sage: F, g, h = b.field_extension('c')
```

Another more awkward example:

```
sage: R.<x> = QQ['x']
sage: K.<a> = NumberField(x^3-2)
sage: S.<X> = K['X']
sage: f = (X+a)^3 + 2*(X+a) + 1
sage: f
X^3 + 3*a*X^2 + (3*a^2 + 2)*X + 2*a + 3
sage: Q.<z> = S.quo(f)
sage: F.<w>, g, h = z.field_extension()
sage: c = g(z)
sage: f(c)
0
sage: h(g(z))
```

```
z
sage: g(h(w))
w
```

AUTHORS:

- Craig Citro (2006-08-06)
- William Stein (2006-08-06)

lift()

Return lift of this polynomial quotient ring element to the unique equivalent polynomial of degree less than the modulus.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3-2)
sage: b = a^2 - 3
sage: b
a^2 - 3
sage: b.lift()
x^2 - 3
```

list()

Return list of the elements of self, of length the same as the degree of the quotient polynomial ring.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 + 2*x - 5)
sage: a^10
-134*a^2 - 35*a + 300
sage: (a^10).list()
[300, -35, -134]
```

matrix()

The matrix of right multiplication by this element on the power basis for the quotient ring.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 + 2*x - 5)
sage: a.matrix()
[ 0  1  0]
[ 0  0  1]
[ 5 -2  0]
```

minpoly()

The minimal polynomial of this element, which is by definition the minimal polynomial of right multiplication by this element.

norm()

The norm of this element, which is the norm of the matrix of right multiplication by this element.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 -389*x^2 + 2*x - 5)
sage: a.norm()
5
```

trace()

The trace of this element, which is the trace of the matrix of right multiplication by this element.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: S.<a> = R.quotient(x^3 -389*x^2 + 2*x - 5)
sage: a.trace()
389
```

28.5 Term Orderings

Sage supports the following term orderings:

Lexicographic (*lex*) $x^a < x^b \Leftrightarrow \exists\, 0 \leq i < n : a_0 = b_0, \ldots, a_{i-1} = b_{i-1}, a_i < b_i$

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: x > y
True
sage: x > y^2
True
sage: x > 1
True
sage: x^1*y^2 > y^3*z^4
True
sage: x^3*y^2*z^4 < x^3*y^2*z^1
False
```

This term ordering is called 'lp' in Singular.

Degree reverse lexicographic (*degrevlex*) Let $deg(x^a) = a_0 + \cdots + a_{n-1}$, then $x^a < x^b \Leftrightarrow deg(x^a) < deg(x^b)$ or $deg(x^a) = deg(x^b)$ and $\exists\, 0 \leq i < n : a_{n-1} = b_{n-1}, \ldots, a_{i+1} = b_{i+1}, a_i > b_i$.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='degrevlex')
sage: x > y
True
sage: x > y^2*z
False
sage: x > 1
True
sage: x^1*y^5*z^2 > x^4*y^1*z^3
True
sage: x^2*y*z^2 > x*y^3*z
False
```

This term ordering is called 'dp' in Singular.

Degree lexicographic (*deglex*) Let $deg(x^a) = a_0 + \cdots + a_{n-1}$, then $x^a < x^b \Leftrightarrow deg(x^a) < deg(x^b)$ or $deg(x^a) = deg(x^b)$ and $\exists\, 0 \leq i < n : a_0 = b_0, \ldots, a_{i-1} = b_{i-1}, a_i < b_i$.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='deglex')
sage: x > y
True
sage: x > y^2*z
False
sage: x > 1
True
```

```
sage: x^1*y^2*z^3 > x^3*y^2*z^0
True
sage: x^2*y*z^2 > x*y^3*z
True
```

This term order is called 'Dp' in Singular.

Inverse lexicographic (*invlex*) $x^a < x^b \Leftrightarrow \exists\ 0 \leq i < n : a_{n-1} = b_{n-1}, \ldots, a_{i+1} = b_{i+1}, a_i < b_i$.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='invlex')
sage: x > y
False
sage: y > x^2
True
sage: x > 1
True
sage: x*y > z
False
```

This term ordering only makes sense in a non-commutative setting because if P is the ring $k[x_1, \ldots, x_n]$ and term ordering 'invlex' then it is equivalent to the ring $k[x_n, \ldots, x_1]$ with term ordering 'lex'.

This ordering is called 'rp' in Singular.

Negative lexicographic (*neglex*) $x^a < x^b \Leftrightarrow \exists\ 0 \leq i < n : a_0 = b_0, \ldots, a_{i-1} = b_{i-1}, a_i > b_i$

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='neglex')
sage: x > y
False
sage: x > 1
False
sage: x^1*y^2 > y^3*z^4
False
sage: x^3*y^2*z^4 < x^3*y^2*z^1
True
sage: x*y > z
False
```

This term ordering is called 'ls' in Singular.

Negative degree reverse lexicographic (*negdegrevlex*) Let $deg(x^a) = a_0 + \cdots + a_{n-1}$, then $x^a < x^b \Leftrightarrow deg(x^a) > deg(x^b)$ or $deg(x^a) = deg(x^b)$ and $\exists\ 0 \leq i < n : a_{n-1} = b_{n-1}, \ldots, a_{i+1} = b_{i+1}, a_i > b_i$.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='negdegrevlex')
sage: x > y
True
sage: x > x^2
True
sage: x > 1
False
sage: x^1*y^2 > y^3*z^4
True
sage: x^2*y*z^2 > x*y^3*z
False
```

This term ordering is called 'ds' in Singular.

Negative degree lexicographic (*negdeglex*) Let $deg(x^a) = a_0 + \cdots + a_{n-1}$, then $x^a < x^b \Leftrightarrow deg(x^a) > deg(x^b)$ or $deg(x^a) = deg(x^b)$ and $\exists\ 0 \leq i < n : a_0 = b_0, \ldots, a_{i-1} = b_{i-1}, a_i < b_i$.

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQ, 3, order='negdeglex')
sage: x > y
True
sage: x > x^2
True
sage: x > 1
False
sage: x^1*y^2 > y^3*z^4
True
sage: x^2*y*z^2 > x*y^3*z
True
```

This term ordering is called 'Ds' in Singular.

Of these, only 'degrevlex', 'deglex', 'invlex' and 'lex' are global orderings.

Additionally all these monomial orderings may be combined to product or block orderings, defined as:

Let $x = (x_0, \ldots, x_{n-1})$ and $y = (y_0, \ldots, y_{m-1})$ be two ordered sets of variables, $<_1$ a monomial ordering on $k[x]$ and $<_2$ a monomial ordering on $k[y]$.

The product ordering (or block ordering) $< := (<_1, <_2)$ on $k[x, y]$ is defined as: $x^a y^b < x^A y^B \Leftrightarrow x^a <_1 x^A$ or $(x^a = x^A$ and $y^b <_2 y^B)$.

These block orderings are constructed in Sage by giving a comma separated list of monomial orderings with the length of each block attached to them.

EXAMPLE:

As an example, consider constructing a block ordering where the first four variables are compared using the degree reverse lexicographical ordering while the last two variables in the second block are compared using negative lexicographical ordering.

```
sage: P.<a,b,c,d,e,f> = PolynomialRing(QQ, 6,order='degrevlex(4),neglex(2)')
sage: a > c^4
False
sage: a > e^4
True
sage: e > f^2
False
```

The same result can be achieved by:

```
sage: T1 = TermOrder('degrevlex',4)
sage: T2 = TermOrder('neglex',2)
sage: T = T1 + T2
sage: P.<a,b,c,d,e,f> = PolynomialRing(QQ, 6, order=T)
sage: a > c^4
False
sage: a > e^4
True
```

If any other unsupported term ordering is given the provided string is passed through as is to Singular, Macaulay2, and Magma. This ensures that it is for example possible to calculate a Groebner basis with respect to some term ordering

Singular supports but Sage doesn't. However a warning is issued to make the user aware of the situation and potential typos:

```
sage: T = TermOrder("royalorder")
verbose 0 (...: term_order.py, __init__) Term ordering 'royalorder' unknown.
```

AUTHORS:

- David Joyner and William Stein: initial version multi_polynomial_ring
- Kiran S. Kedlaya: added macaulay2 interface
- Martin Albrecht: implemented native term orderings, refactoring

class TermOrder (*name='lex', n=0, blocks=True*)

A term order.

See `sage.rings.polynomial.term_order` for details on supported term orderings.

compare_tuples_Dp (*f, g*)

Compares two exponent tuples with respect to the degree lexicographical term order.

INPUT:

- `f` - exponent tuple
- `g` - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='deglex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
True
```

compare_tuples_Ds (*f, g*)

Compares two exponent tuples with respect to the negative degree lexicographical term order.

INPUT:

- `f` - exponent tuple
- `g` - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='negdeglex')
sage: x > y^2 # indirect doctest
True
sage: x > 1
False
```

compare_tuples_block (*f, g*)

Compares two exponent tuples with respect to the block ordering as specified when constructing this element.

INPUT:

- `f` - exponent tuple
- `g` - exponent tuple

EXAMPLE:

```
sage: P.<a,b,c,d,e,f>=PolynomialRing(QQbar, 6, order='degrevlex(3),degrevlex(3)')
sage: a > c^4 # indirect doctest
False
sage: a > e^4
True
```

compare_tuples_dp(*f*, *g*)

Compares two exponent tuples with respect to the degree reversed lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='degrevlex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
True
```

compare_tuples_ds(*f*, *g*)

Compares two exponent tuples with respect to the negative degree reverse lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='negdegrevlex')
sage: x > y^2 # indirect doctest
True
sage: x > 1
False
```

compare_tuples_lp(*f*, *g*)

Compares two exponent tuples with respect to the lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='lex')
sage: x > y^2 # indirect doctest
True
sage: x > 1
True
```

compare_tuples_ls(*f*, *g*)

Compares two exponent tuples with respect to the negative lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='neglex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
False
```

compare_tuples_rp(*f*, *g*)

Compares two exponent tuples with respect to the inversed lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y> = PolynomialRing(QQbar, 2, order='invlex')
sage: x > y^2 # indirect doctest
False
sage: x > 1
True
```

greater_tuple_Dp (*f*, *g*)

Returns the greater exponent tuple with respect to the total degree lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='deglex')
sage: f = x + y; f.lm() # indirect doctest
x
sage: f = x + y^2*z; f.lm()
y^2*z
```

This method is called by the lm/lc/lt methods of `MPolynomial_polydict`.

greater_tuple_Ds (*f*, *g*)

Returns the greater exponent tuple with respect to the negative degree lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='negdeglex')
sage: f = x + y; f.lm() # indirect doctest
x
sage: f = x + x^2; f.lm()
x
sage: f = x^2*y*z^2 + x*y^3*z; f.lm()
x^2*y*z^2
```

This method is called by the lm/lc/lt methods of `MPolynomial_polydict`.

greater_tuple_block (*f*, *g*)

Returns the greater exponent tuple with respect to the block ordering as specified when constructing this element.

This method is called by the lm/lc/lt methods of `MPolynomial_polydict`.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<a,b,c,d,e,f>=PolynomialRing(QQbar, 6, order='degrevlex(3),degrevlex(3)')
sage: f = a + c^4; f.lm() # indirect doctest
c^4
sage: g = a + e^4; g.lm()
a
```

greater_tuple_dp(*f*, *g*)

Returns the greater exponent tuple with respect to the total degree reversed lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='degrevlex')
sage: f = x + y; f.lm() # indirect doctest
x
sage: f = x + y^2*z; f.lm()
y^2*z
```

This method is called by the lm/lc/lt methods of MPolynomial_polydict.

greater_tuple_ds(*f*, *g*)

Returns the greater exponent tuple with respect to the negative degree reverse lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='negdegrevlex')
sage: f = x + y; f.lm() # indirect doctest
x
sage: f = x + x^2; f.lm()
x
sage: f = x^2*y*z^2 + x*y^3*z; f.lm()
x*y^3*z
```

This method is called by the lm/lc/lt methods of MPolynomial_polydict.

greater_tuple_lp(*f*, *g*)

Returns the greater exponent tuple with respect to the lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLES:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='lex')
sage: f = x + y^2; f.lm() # indirect doctest
x
```

This method is called by the lm/lc/lt methods of MPolynomial_polydict.

greater_tuple_ls(*f*, *g*)

Returns the greater exponent tuple with respect to the negative lexicographical term order.

This method is called by the lm/lc/lt methods of MPolynomial_polydict.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<a,b,c,d,e,f>=PolynomialRing(QQbar, 6, order='degrevlex(3),degrevlex(3)')
sage: f = a + c^4; f.lm() # indirect doctest
c^4
sage: g = a + e^4; g.lm()
a
```

greater_tuple_rp(*f*, *g*)

Returns the greater exponent tuple with respect to the inversed lexicographical term order.

INPUT:

- f - exponent tuple
- g - exponent tuple

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQbar, 3, order='invlex')
sage: f = x + y; f.lm() # indirect doctest
y
sage: f = y + x^2; f.lm()
y
```

This method is called by the lm/lc/lt methods of MPolynomial_polydict.

is_global()

Return True if this term ordering is definitely global. Return False otherwise, which includes unknown term orderings.

EXAMPLE:

```
sage: T = TermOrder('lex')
sage: T.is_global()
True
sage: T = TermOrder('degrevlex', 3) + TermOrder('degrevlex', 3)
sage: T.is_global()
True
sage: T = TermOrder('degrevlex', 3) + TermOrder('negdegrevlex', 3)
sage: T.is_global()
False
```

is_local()

Return True if this term ordering is definitely local. Return False otherwise, which includes unknown term orderings.

EXAMPLE:

```
sage: T = TermOrder('lex')
sage: T.is_local()
False
sage: T = TermOrder('negdeglex', 3) + TermOrder('negdegrevlex', 3)
sage: T.is_local()
True
sage: T = TermOrder('degrevlex', 3) + TermOrder('negdegrevlex', 3)
sage: T.is_local()
False
```

macaulay2_str()

Return a Macaulay2 representation of self.

Used to convert polynomial rings to their Macaulay2 representation.

EXAMPLE:

```
sage: P = PolynomialRing(GF(127), 8,names='x',order='degrevlex(3),lex(5)')
sage: T = P.term_order()
sage: T.macaulay2_str()
'(GRevLex => 3,Lex => 5)'
sage: P._macaulay2_() # optional - macaulay2
ZZ/127 [x0, x1, x2, x3, x4, x5, x6, x7, MonomialOrder => {GRevLex => 3, Lex => 5}, MonomialS
```

magma_str()

Return a MAGMA representation of self.

Used to convert polynomial rings to their MAGMA representation.

EXAMPLE:

```
sage: P = PolynomialRing(GF(127), 10,names='x',order='degrevlex')
sage: magma(P)                                                    # optional - magma
Polynomial ring of rank 10 over GF(127)
Graded Reverse Lexicographical Order
Variables: x0, x1, x2, x3, x4, x5, x6, x7, x8, x9

sage: T = P.term_order()
sage: T.magma_str()
'"grevlex"'
```

name()

EXAMPLE:

```
sage: TermOrder('lex').name()
'lex'
```

singular_str()

Return a SINGULAR representation of self.

Used to convert polynomial rings to their SINGULAR representation.

EXAMPLE:

```
sage: P = PolynomialRing(GF(127),10,names='x',order='lex(3),deglex(5),lex(2)')
sage: T = P.term_order()
sage: T.singular_str()
'(lp(3),Dp(5),lp(2))'
sage: P._singular_()
//   characteristic : 127
//   number of vars : 10
//        block   1 : ordering lp
//                  : names    x0 x1 x2
//        block   2 : ordering Dp
//                  : names    x3 x4 x5 x6 x7
//        block   3 : ordering lp
//                  : names    x8 x9
//        block   4 : ordering C
```

28.6 Multivariate Polynomial Rings

Sage implements multivariate polynomial rings through several backends. The generic implementation used the classes `PolyDict` and `ETuple` to construct a dictionary with exponent tuples as keys and coefficients as values.

Additionally, specialized and optimized implementations are provided for multivariate polynomials over $\mathbf{Q}$ and $\mathbf{F}_p$. These are implemented in the classes `MPolynomialRing_libsingular` and `MPolynomial_libsingular`

AUTHORS:

- David Joyner and William Stein
- Kiran S. Kedlaya (2006-02-12): added Macaulay2 analogues of Singular features
- Martin Albrecht (2006-04-21): reorganize class hiearchy for singular rep
- Martin Albrecht (2007-04-20): reorganized class hierarchy to support Pyrex implementations
- Robert Bradshaw (2007-08-15): Coercions from rings in a subset of the variables.

EXAMPLES:

We construct the Frobenius morphism on $\mathrm{F}_5[x, y, z]$ over $\mathbf{F}_5$:

```
sage: R, (x,y,z) = PolynomialRing(GF(5), 3, 'xyz').objgens()
sage: frob = R.hom([x^5, y^5, z^5])
sage: frob(x^2 + 2*y - z^4)
-z^20 + x^10 + 2*y^5
sage: frob((x + 2*y)^3)
x^15 + x^10*y^5 + 2*x^5*y^10 - 2*y^15
sage: (x^5 + 2*y^5)^3
x^15 + x^10*y^5 + 2*x^5*y^10 - 2*y^15
```

We make a polynomial ring in one variable over a polynomial ring in two variables:

```
sage: R.<x, y> = PolynomialRing(QQ, 2)
sage: S.<t> = PowerSeriesRing(R)
sage: t*(x+y)
(x + y)*t
```

class MPolynomialRing_macaulay2_repr()

is_exact()

class MPolynomialRing_polydict(*base_ring, n, names, order*)

Multivariable polynomial ring.

EXAMPLES:

```
sage: R = PolynomialRing(Integers(12), 'x', 5); R
Multivariate Polynomial Ring in x0, x1, x2, x3, x4 over Ring of integers modulo 12
sage: loads(R.dumps()) == R
True
```

class MPolynomialRing_polydict_domain(*base_ring, n, names, order*)

ideal(**gens, **kwds*)

Create an ideal in this polynomial ring.

is_field()

is_integral_domain()

monomial_all_divisors(*t*)

Return a list of all monomials that divide t, coefficients are ignored.

INPUT:

- t - a monomial

OUTPUT: a list of monomials

EXAMPLE:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_all_divisors(x^2*z^3)
[x, x^2, z, x*z, x^2*z, z^2, x*z^2, x^2*z^2, z^3, x*z^3, x^2*z^3]
```

ALGORITHM: addwithcarry idea by Toon Segers

monomial_divides(*a, b*)

Return False if a does not divide b and True otherwise.

INPUT:

- a - monomial
- b - monomial

EXAMPLES:

```
sage: P.<x,y,z>=MPolynomialRing(ZZ,3, order='degrevlex')
doctest:1: DeprecationWarning: MPolynomialRing is deprecated, use PolynomialRing instead!
sage: P.monomial_divides(x*y*z, x^3*y^2*z^4)
True
sage: P.monomial_divides(x^3*y^2*z^4, x*y*z)
False
```

TESTS:

```
sage: P.<x,y,z>=MPolynomialRing(ZZ,3, order='degrevlex')
sage: P.monomial_divides(P(1), P(0))
True
sage: P.monomial_divides(P(1), x)
True
```

monomial_lcm(*f*, *g*)

LCM for monomials. Coefficients are ignored.

INPUT:

- f - monomial
- g - monomial

EXAMPLE:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_lcm(3/2*x*y,x)
x*y
```

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: R.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_lcm(x*y,R.gen())
x*y

sage: P.monomial_lcm(P(3/2),P(2/3))
1

sage: P.monomial_lcm(x,P(1))
x
```

monomial_pairwise_prime(*h*, *g*)

Return True if h and g are pairwise prime. Both are treated as monomials.

INPUT:

- h - monomial
- g - monomial

EXAMPLES:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_pairwise_prime(x^2*z^3, y^4)
True
```

```
sage: P.monomial_pairwise_prime(1/2*x^3*y^2, 3/4*y^3)
False
```

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: Q.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_pairwise_prime(x^2*z^3, Q('y^4'))
True

sage: P.monomial_pairwise_prime(1/2*x^3*y^2, Q(0))
True

sage: P.monomial_pairwise_prime(P(1/2),x)
False
```

monomial_quotient (*f, g, coeff=False*)

Return f/g, where both f and g are treated as monomials. Coefficients are ignored by default.

INPUT:

- `f` - monomial
- `g` - monomial
- `coeff` - divide coefficents as well (default: False)

EXAMPLE:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ, 3, order='degrevlex')
sage: P.monomial_quotient(3/2*x*y,x)
y

sage: P.monomial_quotient(3/2*x*y,2*x,coeff=True)
3/4*y
```

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: R.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: P.monomial_quotient(x*y,x)
y

sage: P.monomial_quotient(x*y,R.gen())
y

sage: P.monomial_quotient(P(0),P(1))
0

sage: P.monomial_quotient(P(1),P(0))
...
ZeroDivisionError

sage: P.monomial_quotient(P(3/2),P(2/3), coeff=True)
9/4

sage: P.monomial_quotient(x,y) # Note the wrong result
x*y^-1
```

```
sage: P.monomial_quotient(x,P(1))
x
```

Note: Assumes that the head term of f is a multiple of the head term of g and return the multiplicant m. If this rule is violated, funny things may happen.

monomial_reduce (*f, G*)

Try to find a g in G where g.lm() divides f. If found (g,flt) is returned, (0,0) otherwise, where flt is f/g.lm(). It is assumed that G is iterable and contains ONLY elements in self.

INPUT:

- f - monomial
- G - list/set of mpolynomials

EXAMPLES:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: f = x*y^2
sage: G = [ 3/2*x^3 + y^2 + 1/2, 1/4*x*y + 2/7, P(1/2)  ]
sage: P.monomial_reduce(f,G)
(y, 1/4*x*y + 2/7)
```

TESTS:

```
sage: from sage.rings.polynomial.multi_polynomial_ring import MPolynomialRing_polydict_domai
sage: P.<x,y,z>=MPolynomialRing_polydict_domain(QQ,3, order='degrevlex')
sage: f = x*y^2
sage: G = [ 3/2*x^3 + y^2 + 1/2, 1/4*x*y + 2/7, P(1/2)  ]

sage: P.monomial_reduce(P(0),G)
(0, 0)

sage: P.monomial_reduce(f,[P(0)])
(0, 0)
```

28.7 Multivariate Polynomials

AUTHORS:

- David Joyner: first version
- William Stein: use dict's instead of lists
- Martin Albrecht malb@informatik.uni-bremen.de: some functions added
- William Stein (2006-02-11): added better __div__ behavior.
- Kiran S. Kedlaya (2006-02-12): added Macaulay2 analogues of some Singular features
- William Stein (2006-04-19): added e.g., `f[1,3]` to get coeff of xy^3; added examples of the new `R.x,y = PolynomialRing(QQ,2)` notation.
- Martin Albrecht: improved singular coercions (restructured class hierarchy) and added ETuples
- Robert Bradshaw (2007-08-14): added support for coercion of polynomials in a subset of variables (including multi-level univariate rings)
- Joel B. Mohler (2008-03): Refactored interactions with ETuples.

EXAMPLES:

We verify Lagrange's four squares identity:

```
sage: R.<a0,a1,a2,a3,b0,b1,b2,b3> = QQbar[]
sage: (a0^2 + a1^2 + a2^2 + a3^2)*(b0^2 + b1^2 + b2^2 + b3^2) == (a0*b0 - a1*b1 - a2*b2 - a3*b3)^2 +
True
```

class MPolynomial_element (*parent, x*)

change_ring (*R*)

element ()

class MPolynomial_polydict (*parent, x*)

Multivariate polynomials implemented in pure python using polydicts.

coefficient (*degrees*)

Return the coefficient of the variables with the degrees specified in the python dictionary `degrees`. Mathematically, this is the coefficient in the base ring adjoined by the variables of this ring not listed in `degrees`. However, the result has the same parent as this polynomial.

This function contrasts with the function `monomial_coefficient` which returns the coefficient in the base ring of a monomial.

INPUT:

- `degrees` - Can be any of:
  - a dictionary of degree restrictions
  - a list of degree restrictions (with None in the unrestricted variables)
  - a monomial (very fast, but not as flexible)

OUTPUT: element of the parent of self

See Also:

For coefficients of specific monomials, look at `monomial_coefficient()`.

EXAMPLES:

```
sage: R.<x, y> = QQbar[]
sage: f = 2 * x * y
sage: c = f.coefficient({x:1,y:1}); c
2
sage: c.parent()
Multivariate Polynomial Ring in x, y over Algebraic Field
sage: c in PolynomialRing(QQbar, 2, names = ['x','y'])
True
sage: f = y^2 - x^9 - 7*x + 5*x*y
sage: f.coefficient({y:1})
5*x
sage: f.coefficient({y:0})
-x^9 + (-7)*x
sage: f.coefficient({x:0,y:0})
0
sage: f=(1+y+y^2)*(1+x+x^2)
sage: f.coefficient({x:0})
y^2 + y + 1
sage: f.coefficient([0,None])
y^2 + y + 1
sage: f.coefficient(x)
y^2 + y + 1
sage: # Be aware that this may not be what you think!
sage: # The physical appearance of the variable x is deceiving -- particularly if the expone
```

```
sage: f.coefficient(x^0) # outputs the full polynomial
x^2*y^2 + x^2*y + x*y^2 + x^2 + x*y + y^2 + x + y + 1

sage: R.<x,y> = RR[]
sage: f=x*y+5
sage: c=f.coefficient({x:0,y:0}); c
5.00000000000000
sage: parent(c)
Multivariate Polynomial Ring in x, y over Real Field with 53 bits of precision
```

AUTHORS:

- Joel B. Mohler (2007-10-31)

constant_coefficient()

Return the constant coefficient of this multivariate polynomial.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.constant_coefficient()
5
sage: f = 3*x^2
sage: f.constant_coefficient()
0
```

degree(*x=None*)

Return the degree of self in x, where x must be one of the generators for the parent of self.

INPUT:

- x - multivariate polynomial (a generator of the parent of self) If x is not specified (or is None), return the total degree, which is the maximum degree of any monomial.

OUTPUT: integer

EXAMPLE:

```
sage: R.<x,y> = RR[]
sage: f = y^2 - x^9 - x
sage: f.degree(x)
9
sage: f.degree(y)
2
sage: (y^10*x - 7*x^2*y^5 + 5*x^3).degree(x)
3
sage: (y^10*x - 7*x^2*y^5 + 5*x^3).degree(y)
10
```

degrees()

Returns a tuple (precisely - an ETuple) with the degree of each variable in this polynomial. The list of degrees is, of course, ordered by the order of the generators.

EXAMPLES:

```
sage: R.<x,y,z>=PolynomialRing(QQbar)
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.degrees()
(2, 2, 0)
sage: f = x^2+z^2
sage: f.degrees()
(2, 0, 2)
sage: f.total_degree()  # this simply illustrates that total degree is not the sum of the de
2
```

```
sage: R.<x,y,z,u>=PolynomialRing(QQbar)
sage: f=(1-x)*(1+y+z+x^3)^5
sage: f.degrees()
(16, 5, 5, 0)
sage: R(0).degrees()
(0, 0, 0, 0)
```

dict()
Return underlying dictioniary with keys the exponents and values the coefficients of this polynomial.

exponents()
Return the exponents of the monomials appearing in self.
EXAMPLES:

```
sage: R.<a,b,c> = PolynomialRing(QQbar, 3)
sage: f = a^3 + b + 2*b^2
sage: f.exponents()
[(3, 0, 0), (0, 2, 0), (0, 1, 0)]
```

factor()
Compute the irreducible factorization of this polynomial if it is univariate.
ALGORITHM: Use univariate factorization code.
If a polynomial is univariate, the appropriate univariate factorization code is called.

```
sage: R.<z> = PolynomialRing(CC,1)
sage: f = z^4 - 6*z + 3
sage: f.factor()
(z - 1.60443920904349) * (z - 0.511399619393097) * (z + 1.05791941421830 - 1.59281852704435*
```

inverse_of_unit()

is_constant()
True if polynomial is constant, and False otherwise.
EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.is_constant()
False
sage: g = 10*x^0
sage: g.is_constant()
True
```

is_generator()
Returns True if self is a generator of it's parent.
EXAMPLES:

```
sage: R.<x,y>=QQbar[]
sage: x.is_generator()
True
sage: (x+y-y).is_generator()
True
sage: (x*y).is_generator()
False
```

is_homogeneous()
Return True if self is a homogeneous polynomial.
EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: (x+y).is_homogeneous()
True
sage: (x.parent()(0)).is_homogeneous()
True
sage: (x+y^2).is_homogeneous()
False
sage: (x^2 + y^2).is_homogeneous()
True
sage: (x^2 + y^2*x).is_homogeneous()
False
sage: (x^2*y + y^2*x).is_homogeneous()
True
```

is_monomial()

Returns True if self is a monomial. A monomial is defined to be a product of generators with coefficient 1.

EXAMPLES:

```
sage: R.<x,y>=QQbar[]
sage: x.is_monomial()
True
sage: (x+2*y).is_monomial()
False
sage: (2*x).is_monomial()
False
sage: (x*y).is_monomial()
True
```

is_unit()

Return True if self is a unit.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: (x+y).is_unit()
False
sage: R(0).is_unit()
False
sage: R(-1).is_unit()
True
sage: R(-1 + x).is_unit()
False
sage: R(2).is_unit()
True
```

is_univariate()

Returns True if this multivariate polynomial is univariate and False otherwise.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.is_univariate()
False
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.is_univariate()
True
sage: f = x^0
sage: f.is_univariate()
True
```

lc()

Returns the leading coefficent of self i.e., self.coefficient(self.lm())

EXAMPLES:

```
sage: R.<x,y,z>=QQbar[]
sage: f=3*x^2-y^2-x*y
sage: f.lc()
3
```

lift(*I*)

given an ideal I = (f_1,...,f_r) and some g (== self) in I, find s_1,...,s_r such that g = s_1 f_1 + ... + s_r f_r

ALGORITHM: Use Singular.

EXAMPLE:

```
sage: A.<x,y> = PolynomialRing(CC,2,order='degrevlex')
sage: I = A.ideal([x^10 + x^9*y^2, y^8 - x^2*y^7 ])
sage: f = x*y^13 + y^12
sage: M = f.lift(I)
sage: M
[y^7, x^7*y^2 + x^8 + x^5*y^3 + x^6*y + x^3*y^4 + x^4*y^2 + x*y^5 + x^2*y^3 + y^4]
sage: sum( map( mul , zip( M, I.gens() ) ) ) == f
True
```

lm()

Returns the lead monomial of self with respect to the term order of self.parent().

EXAMPLES:

```
sage: R.<x,y,z>=PolynomialRing(GF(7),3,order='lex')
sage: (x^1*y^2 + y^3*z^4).lm()
x*y^2
sage: (x^3*y^2*z^4 + x^3*y^2*z^1).lm()
x^3*y^2*z^4

sage: R.<x,y,z>=PolynomialRing(CC,3,order='deglex')
sage: (x^1*y^2*z^3 + x^3*y^2*z^0).lm()
x*y^2*z^3
sage: (x^1*y^2*z^4 + x^1*y^1*z^5).lm()
x*y^2*z^4

sage: R.<x,y,z>=PolynomialRing(QQbar,3,order='degrevlex')
sage: (x^1*y^5*z^2 + x^4*y^1*z^3).lm()
x*y^5*z^2
sage: (x^4*y^7*z^1 + x^4*y^2*z^3).lm()
x^4*y^7*z
```

lt()

Returns the leading term of self i.e., self.lc()*self.lm(). The notion of "leading term" depends on the ordering defined in the parent ring.

EXAMPLES:

```
sage: R.<x,y,z>=PolynomialRing(QQbar)
sage: f=3*x^2-y^2-x*y
sage: f.lt()
3*x^2
sage: R.<x,y,z>=PolynomialRing(QQbar,order="invlex")
sage: f=3*x^2-y^2-x*y
sage: f.lt()
-y^2
```

monomial_coefficient(*mon*)

Return the coefficient in the base ring of the monomial mon in self, where mon must have the same parent as self.

This function contrasts with the function `coefficient` which returns the coefficient of a monomial viewing this polynomial in a polynomial ring over a base ring having fewer variables.

INPUT:

- `mon` - a monomial

OUTPUT: coefficient in base ring

See Also:

For coefficients in a base ring of fewer variables, look at `coefficient()`.

EXAMPLES:

The parent of the return is a member of the base ring.

```
sage: R.<x,y>=QQbar[]
```

The parent of the return is a member of the base ring.

```
sage: f = 2 * x * y
sage: c = f.monomial_coefficient(x*y); c
2
sage: c.parent()
Algebraic Field

sage: f = y^2 + y^2*x - x^9 - 7*x + 5*x*y
sage: f.monomial_coefficient(y^2)
1
sage: f.monomial_coefficient(x*y)
5
sage: f.monomial_coefficient(x^9)
-1
sage: f.monomial_coefficient(x^10)
0

sage: var('a')
a
sage: K.<a> = NumberField(a^2+a+1)
sage: P.<x,y> = K[]
sage: f=(a*x-1)*((a+1)*y-1); f
-x*y + (-a)*x + (-a - 1)*y + 1
sage: f.monomial_coefficient(x)
-a
```

monomials()

Returns the list of monomials in self. The returned list is decreasingly ordered by the term ordering of self.parent().

OUTPUT: list of MPolynomials representing Monomials

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.monomials()
[x^2*y^2, x^2, y, 1]

sage: R.<fx,fy,gx,gy> = QQbar[]
sage: F = ((fx*gy - fy*gx)^3)
sage: F
-fy^3*gx^3 + 3*fx*fy^2*gx^2*gy + (-3)*fx^2*fy*gx*gy^2 + fx^3*gy^3
```

```
sage: F.monomials()
[fy^3*gx^3, fx*fy^2*gx^2*gy, fx^2*fy*gx*gy^2, fx^3*gy^3]
sage: F.coefficients()
[-1, 3, -3, 1]
sage: sum(map(mul,zip(F.coefficients(),F.monomials()))) == F
True
```

nvariables()

Number of variables in this polynomial

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.nvariables ()
2
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.nvariables ()
1
```

quo_rem(*right*)

Returns quotient and remainder of self and right.

EXAMPLE:

```
sage: R.<x,y> = CC[]
sage: f = y*x^2 + x + 1
sage: f.quo_rem(x)
(x*y + 1.00000000000000, 1.00000000000000)
```

ALGORITHM: Use Singular.

reduce(*I*)

Reduce this polynomial by the the polynomials in I.

INPUT:

- `I` - a list of polynomials or an ideal

EXAMPLE:

```
sage: P.<x,y,z> = QQbar[]
sage: f1 = -2 * x^2 + x^3
sage: f2 = -2 * y + x* y
sage: f3 = -x^2 + y^2
sage: F = Ideal([f1,f2,f3])
sage: g = x*y - 3*x*y^2
sage: g.reduce(F)
(-6)*y^2 + 2*y
sage: g.reduce(F.gens())
(-6)*y^2 + 2*y

sage: f = 3*x
sage: f.reduce([2*x,y])
0

sage: k.<w> = CyclotomicField(3)
sage: A.<y9,y12,y13,y15> = PolynomialRing(k)
sage: J = [ y9 + y12]
sage: f = y9 - y12; f.reduce(J)
-2*y12
sage: f = y13*y15; f.reduce(J)
y13*y15
```

```
sage: f = y13*y15 + y9 - y12; f.reduce(J)
y13*y15 - 2*y12
```

subs (*fixed=None, **kw*)

Fixes some given variables in a given multivariate polynomial and returns the changed multivariate polynomials. The polynomial itself is not affected. The variable,value pairs for fixing are to be provided as a dictionary of the form {variable:value}.

This is a special case of evaluating the polynomial with some of the variables constants and the others the original variables.

INPUT:

- `fixed` - (optional) dictionary of inputs
- `**kw` - named parameters

OUTPUT: new MPolynomial

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = x^2 + y + x^2*y^2 + 5
sage: f((5,y))
25*y^2 + y + 30
sage: f.subs({x:5})
25*y^2 + y + 30
```

total_degree()

Return the total degree of self, which is the maximum degree of any monomial in self.

EXAMPLES:

```
sage: R.<x,y,z> = QQbar[]
sage: f=2*x*y^3*z^2
sage: f.total_degree()
6
sage: f=4*x^2*y^2*z^3
sage: f.total_degree()
7
sage: f=99*x^6*y^3*z^9
sage: f.total_degree()
18
sage: f=x*y^3*z^6+3*x^2
sage: f.total_degree()
10
sage: f=z^3+8*x^4*y^5*z
sage: f.total_degree()
10
sage: f=z^9+10*x^4+y^8*x^2
sage: f.total_degree()
10
```

univariate_polynomial (*R=None*)

Returns a univariate polynomial associated to this multivariate polynomial.

INPUT:

- `R` - (default: None) PolynomialRing

If this polynomial is not in at most one variable, then a ValueError exception is raised. This is checked using the is_univariate() method. The new Polynomial is over the same base ring as the given MPolynomial.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.univariate_polynomial()
...
TypeError: polynomial must involve at most one variable
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.univariate_polynomial ()
700*y^2 - 2*y + 305
sage: g.univariate_polynomial(PolynomialRing(QQ,'z'))
700*z^2 - 2*z + 305
```

variable(*i*)

Returns i-th variable occuring in this polynomial.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.variable(0)
x
sage: f.variable(1)
y
```

variables()

Returns the list of variables occuring in this polynomial.

EXAMPLES:

```
sage: R.<x,y> = QQbar[]
sage: f = 3*x^2 - 2*y + 7*x^2*y^2 + 5
sage: f.variables()
[x, y]
sage: g = f.subs({x:10}); g
700*y^2 + (-2)*y + 305
sage: g.variables()
[y]
```

degree_lowest_rational_function(*r*, *x*)

INPUT:

- r - a multivariate rational function
- x - a multivariate polynomial ring generator x

OUTPUT:

- integer - the degree of r in x and its "leading" (in the x-adic sense) coefficient.

Note: This function is dependent on the ordering of a python dict. Thus, it isn't really mathematically well-defined. I think that it should made a method of the FractionFieldElement class and rewritten.

EXAMPLES:

```
sage: R1 = PolynomialRing(FiniteField(5), 3, names = ["a","b","c"])
sage: F = FractionField(R1)
sage: a,b,c = R1.gens()
sage: f = 3*a*b^2*c^3+4*a*b*c
sage: g = a^2*b*c^2+2*a^2*b^4*c^7
```

Consider the quotient $f/g = \frac{4+3bc^2}{ac+2ab^3c^6}$ (note the cancellation).

```
sage: r = f/g; r
(-2*b*c^2 - 1)/(2*a*b^3*c^6 + a*c)
sage: degree_lowest_rational_function(r,a)
(-1, 3)
sage: degree_lowest_rational_function(r,b)
(0, 4)
sage: degree_lowest_rational_function(r,c)
(-1, 4)
```

is_MPolynomial(x)

28.8 Infinite Polynomial Rings

By Infinite Polynomial Rings, we mean polynomial rings in a countably infinite number of variables. The implementation consists of a wrapper around the current *finite* polynomial rings in Sage.

AUTHORS:

- Simon King <simon.king@uni-jena.de>
- Mike Hansen <mhansen@gmail.com>

An Infinite Polynomial Ring has finitely many generators $x_*, y_*, ...$ and infinitely many variables of the form $x_0, x_1, x_2, ..., y_0, y_1, y_2, ..., ...$ We refer to the natural number n as the *index* of the variable x_n.

INPUT:

- `R`, the base ring, which must in fact be a *field*
- `names`, a list of generator names. Generator names must be pairwise different. We only allow generator names that are single characters.
- `order` (optional string). The default order is `'lex'` (lexicographic). `'deglex'` is degree lexicographic, and `'degrevlex'` (degree reverse lexicographic) is possible but discouraged.

Each generator `x` produces an infinite sequence of variables `x[1], x[2], ...` which are printed on screen as `x1, x2, ...` and are latex typeset as x_1, x_2. Then, the Infinite Polynomial Ring is formed by polynomials in these variables.

By default, the monomials are ordered lexicographically. Alternatively, degree (reverse) lexicographic ordering is possible as well. However, we do not guarantee that the computation of Groebner bases will terminate in this case.

In either case, the variables of a Infinite Polynomial Ring X are ordered according to the following rule:

`X.gen(i)[m] < X.gen(j)[n]` if and only if `i<j or (i==j and m<n)`

We provide a 'dense' and a 'sparse' implementation. In the dense implementation, the Infinite Polynomial Ring carries a finite polynomial ring that comprises *all* variables up to the maximal index that has been used so far. This is potentially a very big ring and may also comprise many variables that are not used.

In the sparse implementation, we try to keep the underlying finite polynomial rings small, using only those variables that are really needed. By default, we use the dense implementation, since it usually is much faster.

We provide coercion from an Infinite Polynomial Ring `X` over a field `F_X` to an Infinite Polynomial ring `Y` over a field `F_Y` (regardless of dense or sparse implementation and of monomial ordering) if and only if there is a coercion from

F_X to F_Y, and the set of generator names of X is a subset of the set of generator names of Y. The coercion map is name preserving. We also allow coercion from a *classical* polynomial ring to X if base rings and variable names fit.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: A.<a,b> = InfinitePolynomialRing(QQ, order='deglex')

sage: f = x[5] + 2; f
x5 + 2
sage: g = 3*y[1]; g
3*y1
sage: g._p.parent()
Univariate Polynomial Ring in y1 over Rational Field

sage: f2 = a[5] + 2; f2
a5 + 2
sage: g2 = 3*b[1]; g2
3*b1
sage: A.polynomial_ring()
Multivariate Polynomial Ring in b5, b4, b3, b2, b1, b0, a5, a4, a3, a2, a1, a0 over Rational Field
```

Of course, we provide the usual polynomial arithmetic:

```
sage: f+g
3*y1 + x5 + 2
sage: p = x[10]^2*(f+g); p
3*y1*x10^2 + x10^2*x5 + 2*x10^2
```

There is a permutation action on the variables, by permuting positive variable indices:

```
sage: P = Permutation(((10,1)))
sage: p^P
3*y10*x1^2 + x5*x1^2 + 2*x1^2
```

Note that $x_0^P = x_0$, since the permutations only change *positive* variable indices.

We also implemented ideals of Infinite Polynomial Rings. Here, it is thoroughly assumed that the ideals are set-wise invariant under the permutation action. We therefore refer to these ideals as *Symmetric Ideals*. Symmetric Ideals are finitely generated modulo addition, multiplication by ring elements and permutation of variables, and (at least in the default case of a lexicographic order), one can compute Groebner bases:

```
sage: I = (x[1]*y[2])*X
sage: I.groebner_basis()
[y1*x2, y2*x1]
sage: J = A*(a[1]*b[2])
sage: J.groebner_basis()
[b1*a2, b2*a1]
```

For more details, see SymmetricIdeal.

class InfinitePolynomialGen (*parent, name*)

This class provides the object which is responsible for returning variables in an infinite polynomial ring (implemented in __getitem__()).

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(RR)
sage: x
Generator for the x's in Infinite polynomial ring in x, y over Real Field with 53 bits of precis
sage: x[5]
x5
sage: x == loads(dumps(x))
True
```

class InfinitePolynomialRingFactory()

A factory for creating infinite polynomial ring elements. It handles making sure that they are unique as well as handling pickling. For more details, see `UniqueFactory` and `infinite_polynomial_ring`.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: X2.<x> = InfinitePolynomialRing(QQ)
sage: X is X2
True
sage: X3.<x> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: X is X3
False

sage: X is loads(dumps(X))
True
```

create_key(*R, names=('x',), order='lex', implementation='dense'*)

Creates a key which uniquely defines the infinite polynomial ring.

TESTS:

```
sage: InfinitePolynomialRing.create_key(QQ)
(Rational Field, ('x',), 'lex', 'dense')
sage: InfinitePolynomialRing.create_key(QQ, 'y')
(Rational Field, 'y', 'lex', 'dense')
sage: InfinitePolynomialRing.create_key(QQ, names='y', order='deglex', implementation='spars
(Rational Field, 'y', 'deglex', 'sparse')
sage: InfinitePolynomialRing.create_key(QQ, names=['x','y'], implementation='dense')
(Rational Field, ('x', 'y'), 'lex', 'dense')
```

create_object(*version, key*)

Returns the infinite polynomial ring corresponding to the key `key`.

TESTS:

```
sage: InfinitePolynomialRing.create_object('1.0', (QQ, 'x', 'deglex', 'sparse'))
Infinite polynomial ring in x over Rational Field
```

class InfinitePolynomialRing_dense(*R, names, order*)

Dense implementation of Infinite Polynomial Rings

Compared with `InfinitePolynomialRing_sparse`, from which this class inherits, it keeps a polynomial ring that comprises all elements that have been created so far.

polynomial_ring()

Returns the underlying *finite* polynomial ring.

Note: This ring returned can change over time as more variables are used.

EXAMPLES:

```
sage: X.<x, y> = InfinitePolynomialRing(QQ)
sage: X.polynomial_ring()
Multivariate Polynomial Ring in y0, x0 over Rational Field
```

```
sage: a = y[3]
sage: X.polynomial_ring()
Multivariate Polynomial Ring in y3, y2, y1, y0, x3, x2, x1, x0 over Rational Field
```

class InfinitePolynomialRing_sparse(*R, names, order*)

Sparse implementation of Infinite Polynomial Rings.

An Infinite Polynomial Ring with generators $x_*, y_*, ...$ over a field F is a free commutative F-algebra generated by $x_0, x_1, x_2, ..., y_0, y_1, y_2, ..., ...$ and is equipped with a permutation action on the generators, namely $x_n^P = x_{P(n)}, y_n^P = y_{P(n)}, ...$ for any permutation P (note that variables of index zero are invariant under such permutation).

It is known that any permutation invariant ideal in an Infinite Polynomial Ring is finitely generated modulo the permutation action – see `SymmetricIdeal` for more details.

Usually, an instance of this class is created using `InfinitePolynomialRing` with the optional parameter `implementation='sparse'`. This takes care of uniqueness of parent structures. However, a direct construction is possible, in principle:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: Y.<x,y> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: X is Y
True
sage: from sage.rings.polynomial.infinite_polynomial_ring import InfinitePolynomialRing_sparse
sage: Z = InfinitePolynomialRing_sparse(QQ, ['x','y'], 'lex')
sage: Z == X
True
sage: Z is X
False
```

The last parameter ('lex' in the above example) can also be 'deglex' or 'degrevlex'; this would result in an Infinite Polynomial Ring in degree lexicographic or degree reverse lexicographic order.

See `infinite_polynomial_ring` for more details.

characteristic()

Return the characteristic of the base field.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(GF(25,'a'))
sage: X
Infinite polynomial ring in x, y over Finite Field in a of size 5^2
sage: X.characteristic()
5
```

gen()

Returns the i^{th} 'generator' (see the description in `ngens()`) of this infinite polynomial ring.

EXAMPLES:

```
sage: X = InfinitePolynomialRing(QQ)
sage: x = X.gen()
sage: x[1]
x1
sage: X.gen() is X.gen(0)
True
sage: XX = InfinitePolynomialRing(GF(5))
sage: XX.gen(0) is XX.gen()
True
```

ngens()

Returns the number of generators for this ring. Since there are countably infinitely many variables

in this polynomial ring, by 'generators' we mean the number of infinite families of variables. See `infinite_polynomial_ring` for more details.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: X.ngens()
1

sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: X.ngens()
2
```

varname_cmp(*x, y*)

Comparison of two variable names

INPUT:

- `x, y` should both be strings of the form `a+str(n)`, where a is a single character appearing in the list of generator names, and n is an integer

RETURN:

-1,0,1 if x<y, x==y, x>y, respectively, where the order is defined as follows: x<y $\iff$ the letter `x[0]` is earlier in the list of generator names of self than `y[0]`, or (`x[0]==y[0]` and `int(x[1:])<int(y[1:])`)

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: X.varname_cmp('x1','y10')
-1
sage: X.varname_cmp('y1','x10')
1
sage: X.varname_cmp('y1','y10')
-1
```

28.9 Elements of Infinite Polynomial Rings

AUTHORS:

- Simon King <simon.king@uni-jena.de>
- Mike Hansen <mhansen@gmail.com>

An Infinite Polynomial Ring has generators $x_*, y_*, ...$, so that the variables are of the form $x_0, x_1, x_2, ..., y_0, y_1, y_2, ..., ...$ (see `infinite_polynomial_ring`). Using the generators, we can create elements as follows:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: a = x[3]
sage: b = y[4]
sage: a
x3
sage: b
y4
sage: c = a*b+a^3-2*b^4
sage: c
-2*y4^4 + y4*x3 + x3^3
```

Any Infinite Polynomial Ring `X` is equiped with a monomial ordering. We only consider monomial orderings in which:

`X.gen(i)[m] < X.gen(j)[n]` $\iff$ `i<j`, or `i==j` and `m<n`

Under this restriction, the monomial ordering can be lexicographic (default), degree lexicographic, or degree reverse lexicographic. Here, the ordering is lexicographic, and elements can be compared as usual:

```
sage: X._order
'lex'
sage: a < b
True
```

Note that, when a method is called that is not directly implemented for 'InfinitePolynomial', it is tried to call this method for the underlying *classical* polynomial. This holds, e.g., when applying the `latex` function:

```
sage: latex(c)
-2 y_{4}^{4} + y_{4} x_{3} + x_{3}^{3}
```

There is a permutation action on Infinite Polynomial Rings by permuting the indices of the variables:

```
sage: P = Permutation(((4,5),(2,3)))
sage: c^P
-2*y5^4 + y5*x2 + x2^3
```

Note that `P(0)==0`, and thus variables of index zero are invariant under the permutation action. More generally, if `P` is any callable object that accepts non-negative integers as input and returns non-negative integers, then `c^P` means to apply `P` to the variable indices occuring in `c`.

InfinitePolynomial(*A, p, is_good_poly=False*)

Create an element of a Polynomial Ring with a Countably Infinite Number of Variables

Usually, an InfinitePolynomial is obtained by using the generators of an Infinite Polynomial Ring (see `infinite_polynomial_ring`). But a direct construction is possible as well.

INPUT:

- `A`, an Infinite Polynomial Ring
- `p`, which is either an Infinite Polynomial, a *classical* polynomial, a string, or anything else that can be interpreted in `A`.

If the optional parameter `is_good_poly` is given, then it is assumed that `p` is a *classical* polynomial that can be interpreted in `A`. Otherwise, the input is checked.

EXAMPLES:

```
sage: from sage.rings.polynomial.infinite_polynomial_element import InfinitePolynomial
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: a = InfinitePolynomial(X, '(x1+x2)^2')
sage: a
x2^2 + 2*x2*x1 + x1^2
sage: p = a.polynomial()
sage: b = InfinitePolynomial(X, p)
sage: a==b
True
sage: InfinitePolynomial(X, int(1))
1
sage: InfinitePolynomial(X, 1)
1
```

```
sage: Y.<x,y> = InfinitePolynomialRing(GF(2), implementation='sparse')
sage: InfinitePolynomial(Y, a)
x2^2 + x1^2
```

If it is sure that p is a polynomial that fits well to X then the optional parameter is_good_poly can be used to speed up the element creation. However, this option should be used with care:

```
sage: R.<z0> = QQ[]
sage: InfinitePolynomial(Y, z0, is_good_poly=True)
z0
```

The preceding answer means that it was assumed that z0 fits into an Infinite Polynomial Ring generated by x and y – and in the sparse implementation of Infinite Polynomial Rings this is not tested. In the default implementation, it is somehow indirectly tested: The dense implementation of Infinite Polynomial Rings has an underlying ring, and if is_good_poly is granted, then that underlying ring will not be changed, resulting in a traceback.

```
sage: InfinitePolynomial(X, z0, is_good_poly=True)
...
TypeError: not a constant polynomial
sage: S.<x4> = QQ[]
sage: X.polynomial_ring()
Multivariate Polynomial Ring in x2, x1, x0 over Rational Field
sage: InfinitePolynomial(X, x4, is_good_poly=True)
...
TypeError: not a constant polynomial
```

Without the optional parameter, the underlying ring is appropriately changed:

```
sage: InfinitePolynomial(X, x4)
x4
sage: X.polynomial_ring()
Multivariate Polynomial Ring in x4, x3, x2, x1, x0 over Rational Field
```

class InfinitePolynomial_dense(*A, p, is_good_poly=False*)
Element of a dense Polynomial Ring with a Countably Infinite Number of Variables

INPUT:

- X, an Infinite Polynomial Ring in dense implementation
- p, which is either an Infinite Polynomial, a *classical* polynomial, a string, or anything else that can be interpreted in X.

If the optional parameter is_good_poly is given, then it is assumed that p is a *classical* polynomial that can be interpreted in X. Otherwise, the input is checked.

This class inherits from InfinitePolynomial_sparse. See there for a description of the methods.

class InfinitePolynomial_sparse(*A, p, is_good_poly=False*)
Element of a sparse Polynomial Ring with a Countably Infinite Number of Variables

INPUT:

- A, an Infinite Polynomial Ring in sparse implementation
- p, which is either an Infinite Polynomial, a *classical* polynomial, a string, or anything else that can be interpreted in A.

If the optional parameter is_good_poly is given, then it is assumed that p is a *classical* polynomial that can be interpreted in A. Otherwise, the input is checked.

coefficient(*monomial*)

Returns the coefficient of a monomial in this polynomial.

INPUT:

- A monomial (element of the parent of self) or
- **a dictionary that describes a monomial (the keys** are variables of the parent of self, the values are the corresponding exponents)

EXAMPLES:

We can get the coefficient in front of monomials:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: a = 2*x[0]*x[1] + x[1] + x[2]
sage: a.coefficient(x[0])
2*x1
sage: a.coefficient(x[1])
2*x0 + 1
sage: a.coefficient(x[2])
1
sage: a.coefficient(x[0]*x[1])
2
```

We can also pass in a dictionary:

```
sage: a.coefficient({x[0]:1, x[1]:1})
2
```

footprint()

Leading exponents in increasing order sorted by generator names

OUTPUT: `D` – a dictionary whose keys are the occuring variable indices. `D[s]` is a list `[i_1,...,i_n]`, where `i_j` gives the exponent of `self.parent().gen(j)[s]` in the leading term of self.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p = x[30]*y[1]^3*x[1]^2+2*x[10]*y[30]
sage: p.footprint()
{10: [1, 0], 30: [0, 1]}
```

lc()

The coefficient of the leading term of self

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p = 2*x[10]*y[30]+3*x[10]*y[1]^3*x[1]^2
sage: p.lc()
2
```

lm()

The leading monomial of self

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p = 2*x[10]*y[30]+x[10]*y[1]^3*x[1]^2
sage: p.lm()
y30*x10
```

lt()

The leading term (= product of coefficient and monomial) of self

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p = 2*x[10]*y[30]+3*x[10]*y[1]^3*x[1]^2
sage: p.lt()
2*y30*x10
```

max_index()

Return the maximal index of a variable occuring in self, or -1 if self is scalar

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p=x[1]^2+y[2]^2+x[1]*x[2]*y[3]+x[1]*y[4]
sage: p.max_index()
4
sage: x[0].max_index()
0
sage: X(10).max_index()
-1
```

polynomial()

Return the underlying polynomial

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(GF(7))
sage: p=x[2]*y[1]+3*y[0]
sage: p
y1*x2 + 3*y0
sage: p.polynomial()
y1*x2 + 3*y0
sage: p.polynomial().parent()
Multivariate Polynomial Ring in y2, y1, y0, x2, x1, x0 over Finite Field of size 7
sage: p.parent()
Infinite polynomial ring in x, y over Finite Field of size 7
```

reduce(*I, tailreduce=False, report=None*)

Symmetrical reduction of self with respect to a symmetric ideal (or list of Infinite Polynomials)

Reducing an element p of an Infinite Polynomial Ring X by some other element q means the following:

1. Let M and N be the leading terms of p and q.
2. Test whether there is a permutation P that does not does not diminish the variable indices occuring in N and preserves their order, so that there is some term $T \in X$ with $TN^P = M$. If there is no such permutation, return p
3. Replace p by $p - Tq^P$ and continue with step 1.

INPUT:

- I – a `SymmetricIdeal` or a list of Infinite Polynomials
- Sometimes it is useful to reduce not only the leading term of p. This *tail reduction* is performed if the optional parameter `tailreduce` is True.
- If the optional parameter `report` is not None, some information on the progress of computation is given, since reduction of huge polynomials may be expensive.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p = y[1]^2*y[3]+y[1]*x[3]
sage: p.reduce([y[2]^2*y[1]])
y3*y1^2 + y1*x3
```

The preceding is correct, since any permutation that turns `y[2]^2*y[1]` into a factor of `y[1]^2*y[3]` interchanges the variable indices 1 and 2 – which is not allowed. However, reduction by `y[1]^2*y[2]` works, since one can change variable index 1 into 2 and 2 into 3:

```
sage: p.reduce([y[1]^2*y[2]])
y1*x3
```

The next example shows that tail reduction is not done, unless it is explicitly adviced. The input can also be a Symmetric Ideal:

```
sage: I = (x[2])*X
sage: p.reduce(I)
y3*y1^2 + y1*x3
sage: p.reduce(I, tailreduce=True)
y3*y1^2
```

Last, we demonstrate the `report` option:

```
sage: p=x[1]^2+y[2]^2+x[1]*x[2]*y[3]+x[1]*y[4]
sage: p.reduce(I, tailreduce=True, report=True)
T[3]:>
>
y4*x1 + y2^2 + x1^2
```

The output 'T[3]' means that tail reduction is performed on a polynomial with three terms. ':' means that there was one reduction of the leading monomial. At '>', one round of the reduction process is finished.

ring()

The ring which self belongs to. This is the same as `self.parent()`.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ,implementation='sparse')
sage: p = x[100]*y[1]^3*x[1]^2+2*x[10]*y[30]
sage: p.ring()
Infinite polynomial ring in x, y over Rational Field
```

squeezed()

Reduce the variable indices occuring in self

OUTPUT: Apply a permutation to self that does not change the order of the variable indices of self but squeezes them into the range 1,2,...

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ,implementation='sparse')
sage: p = x[1]*y[100] + x[50]*y[1000]
sage: p.squeezed()
y4*x2 + y3*x1
```

stretch(k)

Replace v_n with $v_{n \cdot k}$ for all generators v_* occuring in self.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: a = x[0] + x[1] + x[2]
sage: a.stretch(2)
x4 + x2 + x0

sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: a = x[0] + x[1] + y[0]*y[1]; a
y1*y0 + x1 + x0
sage: a.stretch(2)
y2*y0 + x2 + x0
```

TESTS:

```
sage: X.<x> = InfinitePolynomialRing(QQ, implementation='sparse')
sage: a = x[2] + x[3]
sage: a.stretch(2000)
x6000 + x4000
```

symmetric_cancellation_order(*other*)

Comparison of leading terms by Symmetric Cancellation Order, $<_{sc}$

INPUT: self, other – two Infinite Polynomials

ASSUMPTION: Both Infintie Polynomials are non-zero

OUTPUT: `(c, sigma, w)`, where

- c = -1,0,1, or None if the leading monomial of `self` is smaller, equal, greater, or incomparable with respect to `other` in the monomial ordering of the Infinite Polynomial Ring
- sigma is a permutation witnessing `self` $<_{sc}$ `other` (resp. `self` $>_{sc}$ `other`) or is 1 if `self.lm()==other.lm()`
- w is 1 or is a term so that `w*self.lt()^sigma == other.lt()` if $c \leq 0$, and `w*other.lt()^sigma == self.lt()` if $c = 1$

THEORY: If the Symmetric Cancellation Order is a well-quasi-ordering then computation of Groebner bases always terminates. This is the case, e.g., if the monomial order is lexicographic. For that reason, lexicographic order is our default order.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: (x[2]*x[1]).symmetric_cancellation_order(x[2]^2)
(None, 1, 1)
sage: (x[2]*x[1]).symmetric_cancellation_order(x[2]*x[3]*y[1])
(-1, [2, 3, 1], y1)
sage: (x[2]*x[1]*y[1]).symmetric_cancellation_order(x[2]*x[3]*y[1])
(None, 1, 1)
sage: (x[2]*x[1]*y[1]).symmetric_cancellation_order(x[2]*x[3]*y[2])
(-1, [2, 3, 1], 1)
```

tail()

The tail of self (this is self minus its leading term)

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p = 2*x[10]*y[30]+3*x[10]*y[1]^3*x[1]^2
sage: p.tail()
3*y1^3*x10*x1^2
```

variables()

Return the variables occuring in self (list of elements of some polynomial ring)

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: p = x[1] + x[2] - 2*x[1]*x[3]
sage: p.variables()
[x3, x2, x1]
sage: x[1].variables()
[x1]
sage: X(1).variables()
[]
```

28.10 Ideals in multivariate polynomial rings.

Sage has a powerful system to compute with multivariate polynomial rings. Most algorithms dealing with these ideals are centered the computation of *Groebner bases*. Sage makes use of Singular to implement this functionality. Singular is widely regarded as the best open-source system for Groebner basis calculation in multivariate polynomial rings over fields.

AUTHORS:

- William Stein
- Kiran S. Kedlaya (2006-02-12): added Macaulay2 analogues of some Singular features
- Martin Albrecht (2008,2007): refactoring, many Singular related functions

EXAMPLES:

We compute a Groebner basis for some given ideal. The type returned by the `groebner_basis` method is `Sequence`, i.e. it is not an `MPolynomialIdeal`.

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: I = ideal(x^5 + y^4 + z^3 - 1,  x^3 + y^3 + z^2 - 1)
sage: B = I.groebner_basis()
sage: type(B)
<class 'sage.structure.sequence.Sequence'>
```

Groebner bases can be used to solve the ideal membership problem.

```
sage: f,g,h = B
sage: (2*x*f + g).reduce(B)
0

sage: (2*x*f + g) in I
True

sage: (2*x*f + 2*z*h + y^3).reduce(B)
y^3

sage: (2*x*f + 2*z*h + y^3) in I
False
```

We compute a Groebner basis for cyclic 6, which is a standard benchmark and test ideal.

```
sage: R.<x,y,z,t,u,v> = QQ['x,y,z,t,u,v']
sage: I = sage.rings.ideal.Cyclic(R,6)
sage: B = I.groebner_basis()
sage: len(B)
45
```

We compute in a quotient of a polynomial ring over $\mathbf{Z}/17\mathbf{Z}$:

```
sage: R.<x,y> = ZZ[]
sage: S.<a,b> = R.quotient((x^2 + y^2, 17))      # optional -- macaulay2
...
verbose 0 ... Warning: falling back to very slow toy implementation.
sage: S                                          # optional -- macaulay2
Quotient of Multivariate Polynomial Ring in x, y over Integer Ring
by the ideal (x^2 + y^2, 17)

sage: a^2 + b^2 == 0                             # optional -- macaulay2
```

```
True
sage: a^3 - b^2                                    # optional -- macaulay2
a*b^2 - b^2
sage: (a+b)^17                                     # optional -- macaulay2
a*b^16 + b^17
sage: S(17) == 0                                   # optional -- macaulay2
True
```

Working with a polynomial ring over $\mathbf{Z}$:

```
sage: R.<x,y,z,w> = ZZ['x,y,z,w']
sage: i = ideal(x^2 + y^2 - z^2 - w^2, x-y)
sage: j = i^2
sage: j.groebner_basis('macaulay2')           # optional - macaulay2
[4*y^4 - 4*y^2*z^2 + z^4 - 4*y^2*w^2 + 2*z^2*w^2 + w^4,
 2*x*y^2 - 2*y^3 - x*z^2 + y*z^2 - x*w^2 + y*w^2,
 x^2 - 2*x*y + y^2]

sage: y^2 - 2*x*y + x^2 in j                  # optional - macaulay2
True
sage: 0 in j                                  # optional - macaulay2
True
```

We do a Groebner basis computation over a number field:

```
sage: K.<zeta> = CyclotomicField(3)
sage: R.<x,y,z> = K[]; R
Multivariate Polynomial Ring in x, y, z over Cyclotomic Field of order 3 and degree 2

sage: i = ideal(x - zeta*y + 1, x^3 - zeta*y^3); i
Ideal (x + (-zeta)*y + 1, x^3 + (-zeta)*y^3) of Multivariate
Polynomial Ring in x, y, z over Cyclotomic Field of order 3 and degree 2

sage: i.groebner_basis()
[y^3 + (2*zeta + 1)*y^2 + (zeta - 1)*y + (-1/3*zeta - 2/3), x + (-zeta)*y + 1]

sage: S = R.quotient(i); S
Quotient of Multivariate Polynomial Ring in x, y, z over
Cyclotomic Field of order 3 and degree 2 by the ideal (x +
(-zeta)*y + 1, x^3 + (-zeta)*y^3)

sage: S.0  - zeta*S.1
-1
sage: S.0^3 - zeta*S.1^3
0
```

Two examples from the Mathematica documentation (done in Sage):

We compute a Groebner basis:

```
sage: R.<x,y> = PolynomialRing(QQ, order='lex')
sage: ideal(x^2 - 2*y^2, x*y - 3).groebner_basis()
 [x - 2/3*y^3, y^4 - 9/2]
```

We show that three polynomials have no common root:

```
sage: R.<x,y> = QQ[]
sage: ideal(x+y, x^2 - 1, y^2 - 2*x).groebner_basis()
[1]
```

The next example shows how we can use Groebner bases over $\mathbf{Z}$ to find the primes modulo which a system of equations has a solution, when the system has no solutions over the rationals.

We first form a certain ideal I in $\mathbf{Z}[x, y, z]$, and note that the Groebner basis of I over $\mathbf{Q}$ contains 1, so there are no solutions over $\mathbf{Q}$ or an algebraic closure of it (this is not surprising as there are 4 equations in 3 unknowns).

```
sage: P.<x,y,z> = PolynomialRing(ZZ,order='lex')
sage: I = ideal(-y^2 - 3*y + z^2 + 3, -2*y*z + z^2 + 2*z + 1, \
                x*z + y*z + z^2, -3*x*y + 2*y*z + 6*z^2)
sage: I.change_ring(P.change_ring(QQ)).groebner_basis()
[1]
```

However, when we compute the Groebner basis of I (defined over \$ZZ\$), we note that there is a certain integer in the ideal which is not 1.

```
sage: I.groebner_basis()          # optional -- macaulay2
...
verbose 0 ... Warning: falling back to very slow toy implementation.
[x + y + z, y^2 + y + 23234, y*z + y + 26532, 2*y + 158864, z^2 + 17223, 2*z + 41856, 164878]
```

Now for each prime p dividing this integer 164878, the Groebner basis of I modulo p will be non-trivial and will thus give a solution of the original system modulo p.

```
sage: factor(164878)
2 * 7 * 11777

sage: I.change_ring(P.change_ring( GF(2) )).groebner_basis()
[x + y + z, y^2 + y, y*z + y, z^2 + 1]
sage: I.change_ring(P.change_ring( GF(7) )).groebner_basis()
[x - 1, y + 3, z - 2]
sage: I.change_ring(P.change_ring( GF(11777 ))).groebner_basis()
[x + 5633, y - 3007, z - 2626]
```

The Groebner basis modulo any product of the prime factors is also non-trivial.

```
sage: I.change_ring(P.change_ring( IntegerModRing(2*7) )).groebner_basis()   # optional -- macau
verbose 0 (...: multi_polynomial_ideal.py, groebner_basis) Warning: falling back to very slow to
[x + y + z, y^2 + y + 8, y*z + y + 2, 2*y + 6, z^2 + 3, 2*z + 10]
```

Modulo any other prime the Groebner basis is trivial so there are no other solutions. For example:

```
sage: I.change_ring( P.change_ring( GF(3) ) ).groebner_basis()
[1]
```

TESTS:

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: I = ideal(x^5 + y^4 + z^3 - 1,  x^3 + y^3 + z^2 - 1)
sage: I == loads(dumps(I))
True
```

class MPolynomialIdeal (*ring, gens, coerce=True*)

change_ring(*P*)

Return the ideal `I` in `P` spanned by the generators $g_1, ..., g_n$ of self as returned by `self.gens()`.

INPUT:

- `P` - a multivariate polynomial ring

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQ,3,order='lex')
sage: I = sage.rings.ideal.Cyclic(P)
sage: I
Ideal (x + y + z, x*y + x*z + y*z, x*y*z - 1) of
Multivariate Polynomial Ring in x, y, z over Rational Field

sage: I.groebner_basis()
[x + y + z, y^2 + y*z + z^2, z^3 - 1]

sage: Q.<x,y,z> = P.change_ring(order='degrevlex'); Q
Multivariate Polynomial Ring in x, y, z over Rational Field
sage: Q.term_order()
Degree reverse lexicographic term order

sage: J = I.change_ring(Q); J
Ideal (x + y + z, x*y + x*z + y*z, x*y*z - 1) of
Multivariate Polynomial Ring in x, y, z over Rational Field

sage: J.groebner_basis()
[z^3 - 1, y^2 + y*z + z^2, x + y + z]
```

groebner_basis()

Return the reduced Groebner basis of this ideal. A Groebner basis $g_1, ..., g_n$ for an ideal I is a basis such that $< LM(g_i) >= LM(I)$, i.e., the leading monomial ideal of I is spanned by the leading terms of $g_1, ..., g_n$. Groebner bases are the key concept in computational ideal theory in multivariate polynomial rings which allows a variety of problems to be solved. Additionally, a *reduced* Groebner basis G is a unique representation for the ideal $< G >$ with respect to the chosen monomial ordering.

INPUT:

- `algorithm` - determines the algorithm to use, see below for available algorithms.
- `*args` - additional parameters passed to the respective implementations
- `**kwds` - additional keyword parameters passed to the respective implementations

ALGORITHMS:

'' autoselect (default)
'singular:groebner' Singular's `groebner` command
'singular:std' Singular's `std` command
'singular:stdhilb' Singular's `stdhib` command
'singular:stdfglm' Singular's `stdfglm` command
'singular:slimgb' Singular's `slimgb` command
'libsingular:std' libSingular's `std` command
'libsingular:slimgb' libSingular's `slimgb` command
'toy:buchberger' Sage's toy/educational buchberger without Buchberger criteria
'toy:buchberger2' Sage's toy/educational buchberger with Buchberger criteria
'toy:d_basis' Sage's toy/educational algorithm for computation over PIDs
'macaulay2:gb' Macaulay2's `gb` command (if available)

'magma:GroebnerBasis' Magma's `Groebnerbasis` command (if available)

If only a system is given - e.g. 'magma' - the default algorithm is chosen for that system.

Note: The Singular and libSingular versions of the respective algorithms are identical, but the former calls an external Singular process while the later calls a C function, i.e. the calling overhead is smaller.

EXAMPLES:

Consider Katsura-3 over QQ with lexicographical term ordering. We compute the reduced Groebner basis using every available implementation and check their equality.

```
sage: P.<a,b,c> = PolynomialRing(QQ,3, order='lex')
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis()
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('singular:groebner')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('singular:std')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('singular:stdhilb')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('singular:stdfglm')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('singular:slimgb')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c
```

Note that `toy:buchberger` does not return the reduced Groebner basis,

```
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('toy:buchberger')
[a^2 - a + 2*b^2 + 2*c^2,
 a*b + b*c - 1/2*b, a + 2*b + 2*c - 1,
 b^2 + 4/3*b*c - 1/3*b + c^2 - 1/3*c,
 b*c - 1/10*b + 6/5*c^2 - 2/5*c,
 b + 30*c^3 - 79/7*c^2 + 3/7*c,
 c^4 - 10/21*c^3 + 1/84*c^2 + 1/84*c]
```

but that `toy:buchberger2` does.

```
sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('toy:buchberger2')
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('macaulay2:gb') # optional - macaulay2
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c

sage: I = sage.rings.ideal.Katsura(P,3) # regenerate to prevent caching
sage: I.groebner_basis('magma:GroebnerBasis') # optional - magma
[a - 60*c^3 + 158/7*c^2 + 8/7*c - 1, b + 30*c^3 - 79/7*c^2 + 3/7*c, c^4 - 10/21*c^3 + 1/84*c
```

Groebner bases over **Z** can be computed. However, the native implementation is very slow. If available Macaulay2 is used, which is an optional Sage package.

```
sage: P.<a,b,c> = PolynomialRing(ZZ,3)
sage: I = P * (a + 2*b + 2*c - 1, a^2 - a + 2*b^2 + 2*c^2, 2*a*b + 2*b*c - b)
sage: I.groebner_basis()                                  # optional - macaulay2
...
[b^3 - b*c^2 - 24*c^3 - b^2 - 6*b*c + 2*c^2 + 2*c,
 2*b*c^2 + 36*c^3 - 9*b*c - 24*c^2 + 2*b + 4*c,
 42*c^3 + b^2 + 2*b*c - 14*c^2 + b,
 2*b^2 - 4*b*c - 6*c^2 + 2*c,
 10*b*c + 12*c^2 - b - 4*c,
 a + 2*b + 2*c - 1]

sage: I.groebner_basis('macaulay2') # optional - macaulay2
[b^3 + b*c^2 + 12*c^3 + b^2 + b*c - 4*c^2,
 2*b*c^2 - 6*c^3 + b^2 + 5*b*c + 8*c^2 - b - 2*c,
 42*c^3 + b^2 + 2*b*c - 14*c^2 + b,
 2*b^2 - 4*b*c - 6*c^2 + 2*c, 10*b*c + 12*c^2 - b - 4*c,
 a + 2*b + 2*c - 1]
```

Sage also supports local orderings:

```
sage: P.<x,y,z> = PolynomialRing(QQ,3,order='negdegrevlex')
sage: I = P * (  x*y*z + z^5, 2*x^2 + y^3 + z^7, 3*z^5 +y ^5 )
sage: I.groebner_basis()
[x^2 + 1/2*y^3, x*y*z + z^5, y^5 + 3*z^5, y^4*z - 2*x*z^5, z^6]
```

We can represent every element in the ideal as a combination of the generators using the `lift()` method:

```
sage: P.<x,y,z> = PolynomialRing(QQ,3)
sage: I = P * ( x*y*z + z^5, 2*x^2 + y^3 + z^7, 3*z^5 +y ^5 )
sage: J = Ideal(I.groebner_basis())
sage: f = sum(P.random_element(terms=2)*f for f in I.gens())
sage: f
1/2*y^2*z^7 - 1/4*y*z^8 + 2*x*z^5 + 95*z^6 + 1/2*y^5 - 1/4*y^4*z + x^2*y^2 + 3/2*x^2*y*z + 9
sage: f.lift(I.gens())
[2*x + 95*z, 1/2*y^2 - 1/4*y*z, 0]
sage: l = f.lift(J.gens()); l
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1/2*y^2 + 1/4*y*z, 1/2*y^2*z^2 - 1/4*y*z^3 + 2*x +
sage: sum(map(mul, zip(l,J.gens()))) == f
True
```

ALGORITHM: Uses Singular, Magma (if available), Macaulay2 (if available), or a toy implementation.

groebner_fan(*is_groebner_basis=False, symmetry=None, verbose=False*)

Return the Groebner fan of this ideal.

The base ring must be $\mathbf{Q}$ or a finite field $\mathbf{F}_p$ of with $p <= 32749$.

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQ)
sage: i = ideal(x^2 - y^2 + 1)
sage: g = i.groebner_fan()
sage: g.reduced_groebner_bases()
[[x^2 - y^2 + 1], [-x^2 + y^2 - 1]]
```

INPUT:

- `is_groebner_basis` - bool (default False). if True, then I.gens() must be a Groebner basis with respect to the standard degree lexicographic term order.
- `symmetry` - default: None; if not None, describes symmetries of the ideal
- `verbose` - default: False; if True, printout useful info during computations

homogenize(*var='h'*)

Return homogeneous ideal spanned by the homogeneous polynomials generated by homogenizing the generators of this ideal.

INPUT:

•h - variable name or variable in cover ring (default: 'h')

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(GF(2))
sage: I = Ideal([x^2*y + z + 1, x + y^2 + 1]); I
Ideal (x^2*y + z + 1, y^2 + x + 1) of Multivariate
Polynomial Ring in x, y, z over Finite Field of size 2

sage: I.homogenize()
Ideal (x^2*y + z*h^2 + h^3, y^2 + x*h + h^2) of
Multivariate Polynomial Ring in x, y, z, h over Finite
Field of size 2

sage: I.homogenize(y)
Ideal (x^2*y + y^3 + y^2*z, x*y) of Multivariate
Polynomial Ring in x, y, z over Finite Field of size 2

       sage: I = Ideal([x^2*y + z^3 + y^2*x, x + y^2 + 1])
sage: I.homogenize()
Ideal (x^2*y + x*y^2 + z^3, y^2 + x*h + h^2) of
Multivariate Polynomial Ring in x, y, z, h over Finite
Field of size 2
```

is_homogeneous()

Return `True` if this ideal is spanned by homogeneous polynomials, i.e. if it is a homogeneous ideal.

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQ,3)
sage: I = sage.rings.ideal.Katsura(P)
sage: I
Ideal (x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y +
2*y*z - y) of Multivariate Polynomial Ring in x, y, z over
Rational Field

sage: I.is_homogeneous()
False

sage: J = I.homogenize()
sage: J
Ideal (x + 2*y + 2*z - h, x^2 + 2*y^2 + 2*z^2 - x*h, 2*x*y
+ 2*y*z - y*h) of Multivariate Polynomial Ring in x, y, z,
h over Rational Field

sage: J.is_homogeneous()
True
```

normal_basis(*algorithm='libsingular', singular=Singular*)

Returns a vector space basis (consisting of monomials) of the quotient ring by the ideal, resp. of a free module by the module, in case it is finite dimensional and if the input is a standard basis with respect to the ring ordering.

INPUT: algorithm - defaults to use libsingular, if it is anything else we will use the kbase() command

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: I = R.ideal(x^2+y^2+z^2-4, x^2+2*y^2-5, x*z-1)
```

```
sage: I.normal_basis()
[y*z^2, z^2, y*z, z, x*y, y, x, 1]
sage: I.normal_basis(algorithm='singular')
[y*z^2, z^2, y*z, z, x*y, y, x, 1]
```

plot(**args, **kwds*)

Plot the real zero locus of this principal ideal.

INPUT:

- `self` - a principal ideal in 2 variables
- `algorithm` - set this to 'surf' if you want 'surf' to plot the ideal (default: None)
- `*args` - optional tuples `(variable, minimum, maximum)` for plotting dimensions
- `**kwds` - optional keyword arguments passed on to `implicit_plot`

EXAMPLES:

Implicit plotting in 2-d:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: I = R.ideal([y^3 - x^2])
sage: I.plot()                              # cusp

sage: I = R.ideal([y^2 - x^2 - 1])
sage: I.plot((x,-3, 3), (y, -2, 2))  # hyperbola

sage: I = R.ideal([y^2 + x^2*(1/4) - 1])
sage: I.plot()                              # ellipse

sage: I = R.ideal([y^2-(x^2-1)*(x-2)])
sage: I.plot()                              # elliptic curve

sage: f = ((x+3)^3 + 2*(x+3)^2 - y^2)*(x^3 - y^2)*((x-3)^3-2*(x-3)^2-y^2)
sage: I = R.ideal(f)
sage: I.plot()                              # the Singular logo
```

This used to be trac #5267:

```
sage: I = R.ideal([-x^2*y+1])
sage: I.plot()
```

AUTHORS:

- Martin Albrecht (2008-09)

reduce(*f*)

Reduce an element modulo the reduced Groebner basis for this ideal. This returns 0 if and only if the element is in this ideal. In any case, this reduction is unique up to monomial orders.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: I = (x^3 + y, y)*R
sage: I.reduce(y)
0
sage: I.reduce(x^3)
0
sage: I.reduce(x - y)
x

sage: I = (y^2 - (x^3 + x))*R
sage: I.reduce(x^3)
y^2 - x
```

```
sage: I.reduce(x^6)
y^4 - 2*x*y^2 + x^2
sage: (y^2 - x)^2
y^4 - 2*x*y^2 + x^2
```

Note: Requires computation of a Groebner basis, which can be a very expensive operation.

weil_restriction()

Compute the Weil restriction of this ideal over some extension field.

A Weil restriction of scalars - denoted $Res_{L/k}$ - is a functor which, for any finite extension of fields L/k and any algebraic variety X over L, produces another corresponding variety $Res_{L/k}(X)$, defined over k. It is useful for reducing questions about varieties over large fields to questions about more complicated varieties over smaller fields.

This function does not compute this Weil restriction directly but computes on generating sets of polynomial ideals:

Let d be the degree of the field extension L/k, let a a generator of L/k and p the minimal polynomial of L/k. Denote this ideal by I.

Specifically, this function first maps each variable x to its representation over k: $\sum_{i=0}^{d-1} a^i x_i$. Then each generator of I is evaulated over these representations and reduced modulo the minimal polynomial p. The result is interpreted as a univariate polynomial in a and its coefficients are the new generators of the returned ideal.

If the input and the output ideals are radical, this is equivalent to the statement about algebraic varieties above.

EXAMPLE:

```
sage: k.<a> = GF(2^2)
sage: P.<x,y> = PolynomialRing(k,2)
sage: I = Ideal([x*y + 1, a*x + 1])
sage: I.variety()
[{y: a, x: a + 1}]
sage: J = I.weil_restriction()
sage: J
Ideal (x1*y0 + x0*y1 + x1*y1, x0*y0 + x1*y1 + 1, x0 + x1, x1 + 1) of
Multivariate Polynomial Ring in x0, x1, y0, y1 over Finite Field of size 2
sage: J += sage.rings.ideal.FieldIdeal(J.ring()) # ensure radical ideal
sage: J.variety()
[{y1: 1, x1: 1, x0: 1, y0: 0}]

sage: J.weil_restriction() # returns J
Ideal (x1*y0 + x0*y1 + x1*y1, x0*y0 + x1*y1 + 1, x0 + x1, x1 + 1, x0^2 + x0,
       x1^2 + x1, y0^2 + y0, y1^2 + y1)
       of Multivariate Polynomial Ring in x0, x1, y0, y1 over Finite Field of size 2

sage: k.<a> = GF(3^5)
sage: P.<x,y,z> = PolynomialRing(k)
sage: I = sage.rings.ideal.Katsura(P)
sage: I.dimension()
0
sage: I.variety()
[{y: 0, z: 0, x: 1}]

sage: J = I.weil_restriction(); J
Ideal (x4 - y4 - z4, x3 - y3 - z3, x2 - y2 - z2, x1 - y1 - z1, x0 - y0 - z0 - 1,
       x2^2 - x1*x3 - x0*x4 + x4^2 - y2^2 + y1*y3 + y0*y4 - y4^2 - z2^2 + z1*z3 + z0*z4 - z4
       -x1*x2 - x0*x3 - x3*x4 - x4^2 + y1*y2 + y0*y3 + y3*y4 + y4^2 + z1*z2 + z0*z3 + z3*z4
       x1^2 - x0*x2 + x3^2 - x2*x4 + x3*x4 - y1^2 + y0*y2 - y3^2 + y2*y4 - y3*y4 - z1^2 + z0
       -x0*x1 - x2*x3 - x3^2 - x1*x4 + x2*x4 + y0*y1 + y2*y3 + y3^2 + y1*y4 - y2*y4 + z0*z1
```

```
        x0^2 + x2*x3 + x1*x4 - y0^2 - y2*y3 - y1*y4 - z0^2 - z2*z3 - z1*z4 - x0,
        -x4*y0 - x3*y1 - x2*y2 - x1*y3 - x0*y4 - x4*y4 - y4*z0 - y3*z1 - y2*z2 - y1*z3 - y0*z
        -x3*y0 - x2*y1 - x1*y2 - x0*y3 - x4*y3 - x3*y4 + x4*y4 - y3*z0 - y2*z1 - y1*z2 - y0*z
        -x2*y0 - x1*y1 - x0*y2 - x4*y2 - x3*y3 + x4*y3 - x2*y4 + x3*y4 - y2*z0 - y1*z1 - y0*z
        -x1*y0 - x0*y1 - x4*y1 - x3*y2 + x4*y2 - x2*y3 + x3*y3 - x1*y4 + x2*y4 - y1*z0 - y0*z
        -x0*y0 + x4*y1 + x3*y2 + x2*y3 + x1*y4 - y0*z0 + y4*z1 + y3*z2 + y2*z3 + y1*z4 - y0)
        x0, x1, x2, x3, x4, y0, y1, y2, y3, y4, z0, z1, z2, z3, z4 over Finite Field of size
sage: J += sage.rings.ideal.FieldIdeal(J.ring()) # ensure radical ideal
sage: J.variety()
[{y1: 0, y4: 0, z2: 0, y2: 0, x0: 1, y0: 0, x2: 0, z4: 0, z3: 0, x4: 0, x1: 0, z1: 0, z0: 0,
```

Weil restrictions are often used to study elliptic curves over extension fields so we give a simple example involving those:

```
sage: K.<a> = QuadraticField(1/3)
sage: E = EllipticCurve(K,[1,2,3,4,5])
```

We pick a point on `E`:

```
sage: p = E.lift_x(1); p
(1 : 2 : 1)

sage: I = E.defining_ideal(); I
Ideal (-x^3 - 2*x^2*z + x*y*z + y^2*z - 4*x*z^2 + 3*y*z^2 - 5*z^3)
of Multivariate Polynomial Ring in x, y, z over Number Field in a with defining polynomial x
```

Of course, the point `p` is a root of all generators of `I`:

```
sage: [f.subs(x=1,y=2,z=1) for f in I.gens()]
[0]
```

`I` is also radical:

```
sage: I.radical() == I
True
```

So we compute its Weil restriction:

```
sage: J = I.weil_restriction()
sage: J
Ideal (-3*x0^2*x1 - 1/3*x1^3 - 4*x0*x1*z0 + x1*y0*z0 + x0*y1*z0 + 2*y0*y1*z0 - 4*x1*z0^2 + 3
        - 2/3*x1^2*z1 + x0*y0*z1 + y0^2*z1 + 1/3*x1*y1*z1 + 1/3*y1^2*z1 - 8*x0*z0*z1 + 6*y0*
       -x0^3 - x0*x1^2 - 2*x0^2*z0 - 2/3*x1^2*z0 + x0*y0*z0 + y0^2*z0 + 1/3*x1*y1*z0 + 1/3*y
        + 1/3*x1*y0*z1 + 1/3*x0*y1*z1 + 2/3*y0*y1*z1 - 8/3*x1*z0*z1 + 2*y1*z0*z1 - 4/3*x0*z1
of Multivariate Polynomial Ring in x0, x1, y0, y1, z0, z1 over Rational Field
```

We can check that the point `p` is still a root of all generators of `J`:

```
sage: [f.subs(x0=1,y0=2,z0=1,x1=0,y1=0,z1=0) for f in J.gens()]
[0, 0]
```

Note: Based on a Singular implementation by Michael Brickenstein

class MPolynomialIdeal_macaulay2_repr()

An ideal in a multivariate polynomial ring, which has an underlying Macaulay2 ring associated to it.

EXAMPLES:

```
sage: R.<x,y,z,w> = PolynomialRing(ZZ, 4)
sage: I = ideal(x*y-z^2, y^2-w^2)
sage: I
Ideal (x*y - z^2, y^2 - w^2) of Multivariate Polynomial Ring in x, y, z, w over Integer Ring
```

class MPolynomialIdeal_magma_repr()

class MPolynomialIdeal_singular_repr()

An ideal in a multivariate polynomial ring, which has an underlying Singular ring associated to it.

associated_primes(**args, **kwds*)

Return a list of primary ideals (and their associated primes) such that their intersection is I = `self`.

An ideal Q is called primary if it is a proper ideal of the ring R and if whenever $ab \in Q$ and $a \notin Q$ then $b^n \in Q$ for some $n \in \mathbf{Z}$.

If Q is a primary ideal of the ring R, then the radical ideal P of Q, i.e. $P = \{a \in R, a^n \in Q\}$ for some $n \in \mathbf{Z}$, is called the *associated prime* of Q.

If I is a proper ideal of the ring R then there exists a decomposition in primary ideals Q_i such that

- their intersection is I
- none of the Q_i contains the intersection of the rest, and
- the associated prime ideals of Q_i are pairwise different.

This method returns the associated primes of the Q_i.

INPUT:

- `algorithm` - string:
- `'sy'` - (default) use the shimoyama-yokoyama algorithm
- `'gtz'` - use the gianni-trager-zacharias algorithm

OUTPUT:

- `list` - a list of primary ideals and their associated primes [(primary ideal, associated prime), ...]

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: pd = I.associated_primes(); pd
[Ideal (z^3 + 2, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field,
 Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field]
```

ALGORITHM: Uses Singular.

REFERENCES:

- Thomas Becker and Volker Weispfenning. Groebner Bases - A Computational Approach To Commutative Algebra. Springer, New York 1993.

basis_is_groebner(*singular=Singular*)

Returns `True` if the generators of `self` (`self.gens()`) form a Groebner basis.

Let I be the set of generators of this ideal. The check is performed by trying to lift $Syz(LM(I))$ to $Syz(I)$ as I forms a Groebner basis if and only if for every element S in $Syz(LM(I))$:

$$S \cdot G = \sum_{i=0}^{m} h_i g_i \rightarrow_G 0.$$

.

ALGORITHM: Uses Singular

EXAMPLE:

```
sage: R.<a,b,c,d,e,f,g,h,i,j> = PolynomialRing(GF(127),10)
sage: I = sage.rings.ideal.Cyclic(R,4)
sage: I.basis_is_groebner()
False
sage: I2 = Ideal(I.groebner_basis())
sage: I2.basis_is_groebner()
True
```

A more complicated example:

```
sage: R.<U6,U5,U4,U3,U2, u6,u5,u4,u3,u2, h> = PolynomialRing(GF(7583))
sage: l = [u6 + u5 + u4 + u3 + u2 - 3791*h, \
           U6 + U5 + U4 + U3 + U2 - 3791*h, \
           U2*u2 - h^2, U3*u3 - h^2, U4*u4 - h^2, \
           U5*u4 + U5*u3 + U4*u3 + U5*u2 + U4*u2 + U3*u2 - 3791*U5*h - 3791*U4*h - 3791*U3*h
           U4*u5 + U3*u5 + U2*u5 + U3*u4 + U2*u4 + U2*u3 - 3791*u5*h - 3791*u4*h - 3791*u3*h
           U5*u5 - h^2, U4*U2*u3 + U5*U3*u2 + U4*U3*u2 + U3^2*u2 - 3791*U5*U3*h - 3791*U4*U3
            - 3791*U4*U2*h + U3*U2*h - 3791*U2^2*h - 3791*U4*u3*h - 3791*U4*u2*h - 3791*U3*u
           U3*u5*u4 + U2*u5*u4 + U3*u4^2 + U2*u4^2 + U2*u4*u3 - 3791*u5*u4*h - 3791*u4^2*h -
           U2*u5*u4*u3 + U2*u4^2*u3 + U2*u4*u3^2 - 3791*u5*u4*u3*h - 3791*u4^2*u3*h - 3791*u
           U5^2*U4*u3 + U5*U4^2*u3 + U5^2*U4*u2 + U5*U4^2*u2 + U5^2*U3*u2 + 2*U5*U4*U3*u2 +
            + U5*U4*U3*h - 3791*U5*U3^2*h - 3791*U5^2*U2*h + U5*U4*U2*h + U5*U3*U2*h - 3791*
            - U4^2*h^2 - 947*U5*U3*h^2 - U4*U3*h^2 - 948*U5*U2*h^2 - U4*U2*h^2 - 1422*U5*h^3
           u5*u4*u3*u2*h + u4^2*u3*u2*h + u4*u3^2*u2*h + u4*u3*u2^2*h + 2*u5*u4*u3*h^2 + 2*u
            + 2*u5*u3*u2*h^2 + 1899*u4*u3*u2*h^2, \
           U5^2*U4*U3*u2 + U5*U4^2*U3*u2 + U5*U4*U3^2*u2 - 3791*U5^2*U4*U3*h - 3791*U5*U4^2*
            + 3791*U5*U4*U3*u2*h + U5^2*U4*h^2 + U5*U4^2*h^2 + U5^2*U3*h^2 - U4^2*U3*h^2 - U
            - U5*U3*U2*h^2 - U4*U3*U2*h^2 + 3791*U5*U4*h^3 + 3791*U5*U3*h^3 + 3791*U4*U3*h^3
           u4^2*u3*u2*h^2 + 1515*u5*u3^2*u2*h^2 + u4*u3^2*u2*h^2 + 1515*u5*u4*u2^2*h^2 + 151
            + 1521*u5*u4*u3*h^3 - 3028*u4^2*u3*h^3 - 3028*u4*u3^2*h^3 + 1521*u5*u4*u2*h^3 -
           U5^2*U4*U3*U2*h + U5*U4^2*U3*U2*h + U5*U4*U3^2*U2*h + U5*U4*U3*U2^2*h + 2*U5^2*U4
            + 2*U5^2*U4*U2*h^2 + 2*U5*U4^2*U2*h^2 + 2*U5^2*U3*U2*h^2 - 2*U4^2*U3*U2*h^2 - 2*
             - 2*U5*U4*U2^2*h^2 - 2*U5*U3*U2^2*h^2 - 2*U4*U3*U2^2*h^2 - U5*U4*U3*h^3 - U5*U4

sage: Ideal(l).basis_is_groebner()
False
sage: gb = Ideal(l).groebner_basis()
sage: Ideal(gb).basis_is_groebner()
True
```

Note: From the Singular Manual for the reduce function we use in this method: 'The result may have no meaning if the second argument (`self`) is not a standard basis'. I (malb) believe this refers to the mathematical fact that the results may have no meaning if self is no standard basis, i.e., Singular doesn't 'add' any additional 'nonsense' to the result. So we may actually use reduce to determine if self is a Groebner basis.

complete_primary_decomposition(**args, **kwds*)

Return a list of primary ideals and their associated primes such that the intersection of the primary ideal Q_i is I = `self`.

An ideal Q is called primary if it is a proper ideal of the ring R and if whenever $ab \in Q$ and $a \notin Q$ then $b^n \in Q$ for some $n \in \mathbf{Z}$.

If Q is a primary ideal of the ring R, then the radical ideal P of Q, i.e. $P = \{a \in R, a^n \in Q\}$ for some $n \in \mathbf{Z}$, is called the *associated prime* of Q.

If I is a proper ideal of the ring R then there exists a decomposition in primary ideals Q_i such that

- their intersection is I
- none of the Q_i contains the intersection of the rest, and
- the associated prime ideals of Q_i are pairwise different.

This method returns these Q_i and their associated primes.

INPUT:

- `algorithm` - string:
- `'sy'` - (default) use the shimoyama-yokoyama algorithm
- `'gtz'` - use the gianni-trager-zacharias algorithm

OUTPUT:

- `list` - a list of primary ideals and their associated primes [(primary ideal, associated prime), ...]

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: pd = I.complete_primary_decomposition(); pd
[(Ideal (z^6 + 4*z^3 + 4, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational
  Ideal (z^3 + 2, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field),
 (Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field,
  Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field)]

sage: I.complete_primary_decomposition(algorithm = 'gtz')
[(Ideal (z^6 + 4*z^3 + 4, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational
  Ideal (z^3 + 2, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field),
 (Ideal (z^2 + 1, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field,
  Ideal (z^2 + 1, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational Field)]

sage: reduce(lambda Qi,Qj: Qi.intersection(Qj), [Qi for (Qi,radQi) in pd]) == I
True

sage: [Qi.radical() == radQi for (Qi,radQi) in pd]
[True, True]
```

ALGORITHM: Uses Singular.

REFERENCES:

- Thomas Becker and Volker Weispfenning. Groebner Bases - A Computational Approach To Commutative Algebra. Springer, New York 1993.

dimension (*singular=Singular*)

The dimension of the ring modulo this ideal.

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(GF(32003),order='degrevlex')
sage: I = ideal(x^2-y,x^3)
sage: I.dimension()
1
```

For polynomials over a finite field of order too large for Singular, this falls back on a toy implementation of Buchberger to compute the Groebner basis, then uses the algorithm described in Chapter 9, Section 1 of Cox, Little, and O'Shea's "Ideals, Varieties, and Algorithms".

EXAMPLE:

```
sage: R.<x,y> = PolynomialRing(GF(2147483659),order='lex')
sage: I = R.ideal([x*y,x*y+1])
sage: I.dimension()
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy
0
sage: I=ideal([x*(x*y+1),y*(x*y+1)])
sage: I.dimension()
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy
1
sage: I = R.ideal([x^3*y,x*y^2])
sage: I.dimension()
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy
1
sage: R.<x,y> = PolynomialRing(GF(2147483659),order='lex')
sage: I = R.ideal(0)
sage: I.dimension()
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy
2
```

ALGORITHM: Uses Singular, unless the characteristic is too large.

Note: Requires computation of a Groebner basis, which can be a very expensive operation.

elimination_ideal(**args, **kwds*)

Returns the elimination ideal this ideal with respect to the variables given in `variables`.

INPUT:

- `variables` - a list or tuple of variables in `self.ring()`

EXAMPLE:

```
sage: R.<x,y,t,s,z> = PolynomialRing(QQ,5)
sage: I = R * [x-t,y-t^2,z-t^3,s-x+y^3]
sage: I.elimination_ideal([t,s])
Ideal (y^2 - x*z, x*y - z, x^2 - y) of Multivariate
Polynomial Ring in x, y, t, s, z over Rational Field
```

ALGORITHM: Uses SINGULAR

Note: Requires computation of a Groebner basis, which can be a very expensive operation.

genus()

Return the genus of the projective curve defined by this ideal, which must be 1 dimensional.

EXAMPLE: Consider the hyperelliptic curve $y^2 = 4x^5 - 30x^3 + 45x - 22$ over $\mathbf{Q}$, it has genus 2:

```
sage: P, x = PolynomialRing(QQ,"x").objgen()
sage: f = 4*x^5 - 30*x^3 + 45*x - 22
sage: C = HyperellipticCurve(f); C
Hyperelliptic Curve over Rational Field defined by y^2 = 4*x^5 - 30*x^3 + 45*x - 22
sage: C.genus()
2

sage: P.<x,y> = PolynomialRing(QQ)
sage: f = y^2 - 4*x^5 - 30*x^3 + 45*x - 22
sage: I = Ideal([f])
sage: I.genus()
2
```

hilbert_polynomial()

Return the Hilbert polynomial of this ideal.

Let I = `self` be a homogeneous ideal and R = `self.ring()` be a graded commutative algebra ($R = \oplus R_d$) over a field K. The Hilbert polynomial is the unique polynomial $HP(t)$ with rational coefficients such that $HP(d) = dim_K R_d$ for all but finitely many positive integers d.

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5])
sage: I.hilbert_polynomial()
5*t - 5
```

hilbert_series(*singular=Singular*)

Return the Hilbert series of this ideal.

Let I = `self` be a homogeneous ideal and R = `self.ring()` be a graded commutative algebra ($R = \oplus R_d$) over a field K. Then the Hilbert function is defined as $H(d) = dim_K R_d$ and the Hilbert series of I is defined as the formal power series $HS(t) = \sum_0^\infty H(d)t^d$.

This power series can be expressed as $HS(t) = Q(t)/(1-t)^n$ where $Q(t)$ is a polynomial over Z and n the number of variables in R. This method returns $Q(t)/(1-t)^n$.

EXAMPLE:

```
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([x^3*y^2 + 3*x^2*y^2*z + y^3*z^2 + z^5])
```

```
sage: I.hilbert_series()
(-t^4 - t^3 - t^2 - t - 1)/(-t^2 + 2*t - 1)
```

integral_closure(**args, **kwds*)

Let I = `self`.

Returns the integral closure of $I, ..., I^p$, where sI is an ideal in the polynomial ring $R = k[x(1), ...x(n)]$. If p is not given, or $p = 0$, compute the closure of all powers up to the maximum degree in t occurring in the closure of $R[It]$ (so this is the last power whose closure is not just the sum/product of the smaller). If r is given and `r is True`, `I.integral_closure()` starts with a check whether I is already a radical ideal.

INPUT:

- `p` - powers of I (default: 0)
- `r` - check whether self is a radical ideal first (default: True)

EXAMPLE:

```
sage: R.<x,y> = QQ[]
sage: I = ideal([x^2,x*y^4,y^5])
sage: I.integral_closure()
[x^2, y^5, -x*y^3]
```

ALGORITHM: Use Singular

interreduced_basis(**args, **kwds*)

If this ideal is spanned by $(f_1, ..., f_n)$ this method returns $(g_1, ..., g_s)$ such that:

- $(f_1, ..., f_n) = (g_1, ..., g_s)$
- $LT(g_i)! = LT(g_j)$ for all $i! = j$
- $LT(g_i)$ does not divide m for all monomials m of $\{g_1, ..., g_{i-1}, g_{i+1}, ..., g_s\}$
- $LC(g_i) == 1$ for all i.

EXAMPLE:

```
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([z*x+y^3,z+y^3,z+x*y])
sage: I.interreduced_basis()
[x*z - z, x*y + z, y^3 + z]

sage: R.<x,y,z> = PolynomialRing(QQ,order='negdegrevlex')
sage: I = Ideal([z*x+y^3,z+y^3,z+x*y])
sage: I.interreduced_basis()
[x*z + y^3, x*y - y^3, z + y^3]
```

ALGORITHM: Uses Singular's interred command or `toy_buchberger.inter_reduction` if conversion to Singular fails.

intersection(**args, **kwds*)

Return the intersection of the two ideals.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ, 2, order='lex')
sage: I = x*R
sage: J = y*R
sage: I.intersection(J)
Ideal (x*y) of Multivariate Polynomial Ring in x, y over Rational Field
```

The following simple example illustrates that the product need not equal the intersection.

```
sage: I = (x^2, y)*R
sage: J = (y^2, x)*R
```

```
sage: K = I.intersection(J); K
Ideal (y^2, x*y, x^2) of Multivariate Polynomial Ring in x, y over Rational Field
sage: IJ = I*J; IJ
Ideal (x^2*y^2, x^3, y^3, x*y) of Multivariate Polynomial Ring in x, y over Rational Field
sage: IJ == K
False
```

minimal_associated_primes(**args, **kwds*)

OUTPUT:

- `list` - a list of prime ideals

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3, 'xyz')
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: I.minimal_associated_primes ()
[Ideal (z^2 + 1, -z^2 + y) of Multivariate Polynomial Ring
in x, y, z over Rational Field, Ideal (z^3 + 2, -z^2 + y)
of Multivariate Polynomial Ring in x, y, z over Rational
Field]
```

ALGORITHM: Uses Singular.

plot(*singular=Singular*)

If you somehow manage to install surf, perhaps you can use this function to implicitly plot the real zero locus of this ideal (if principal).

INPUT:

- `self` - must be a principal ideal in 2 or 3 vars over QQ.

EXAMPLES:

Implicit plotting in 2-d:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: I = R.ideal([y^3 - x^2])
sage: I.plot()        # cusp         optional - surf
sage: I = R.ideal([y^2 - x^2 - 1])
sage: I.plot()        # hyperbola    optional - surf
sage: I = R.ideal([y^2 + x^2*(1/4) - 1])
sage: I.plot()        # ellipse      optional - surf
sage: I = R.ideal([y^2-(x^2-1)*(x-2)])
sage: I.plot()        # elliptic curve  optional - surf
```

Implicit plotting in 3-d:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: I = R.ideal([y^2 + x^2*(1/4) - z])
sage: I.plot()          # a cone         optional - surf
sage: I = R.ideal([y^2 + z^2*(1/4) - x])
sage: I.plot()          # same code, from a different angle  optional - surf
sage: I = R.ideal([x^2*y^2+x^2*z^2+y^2*z^2-16*x*y*z])
sage: I.plot()          # Steiner surface   optional - surf
```

AUTHORS:

- David Joyner (2006-02-12)

primary_decomposition(*algorithm='sy'*)

Return a list of primary ideals such that their intersection is I = `self`.

An ideal Q is called primary if it is a proper ideal of the ring R and if whenever $ab \in Q$ and $a \notin Q$ then $b^n \in Q$ for some $n \in \mathbf{Z}$.

If I is a proper ideal of the ring R then there exists a decomposition in primary ideals Q_i such that

- their intersection is I
- none of the Q_i contains the intersection of the rest, and
- the associated prime ideals of Q_i are pairwise different.

This method returns these Q_i.

INPUT:

- `algorithm` - string:
- `'sy'` - (default) use the shimoyama-yokoyama algorithm
- `'gtz'` - use the gianni-trager-zacharias algorithm

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: pd = I.primary_decomposition(); pd
[Ideal (z^6 + 4*z^3 + 4, y - z^2) of Multivariate Polynomial Ring in x, y, z over Rational F
 Ideal (z^2 + 1, y + 1) of Multivariate Polynomial Ring in x, y, z over Rational Field]

sage: reduce(lambda Qi,Qj: Qi.intersection(Qj), pd) == I
True
```

ALGORITHM: Uses Singular.

REFERENCES:

- Thomas Becker and Volker Weispfenning. Groebner Bases - A Computational Approach To Commutative Algebra. Springer, New York 1993.

quotient (**args, **kwds*)

Given ideals I = `self` and J in the same polynomial ring P, return the ideal quotient of I by J consisting of the polynomials a of P such that $\{aJ \subset I\}$.

This is also referred to as the colon ideal $(I{:}J)$.

INPUT:

- `J` - multivariate polynomial ideal

EXAMPLE:

```
sage: R.<x,y,z> = PolynomialRing(GF(181),3)
sage: I = Ideal([x^2+x*y*z,y^2-z^3*y,z^3+y^5*x*z])
sage: J = Ideal([x])
sage: Q = I.quotient(J)
sage: y*z + x in I
False
sage: x in J
True
sage: x * (y*z + x) in I
True
```

radical (**args, **kwds*)

The radical of this ideal.

EXAMPLES:

This is an obviously not radical ideal:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3)
sage: I = (x^2, y^3, (x*z)^4 + y^3 + 10*x^2)*R
sage: I.radical()
Ideal (y, x) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

That the radical is correct is clear from the Groebner basis.

```
sage: I.groebner_basis()
[y^3, x^2]
```

This is the example from the Singular manual:

```
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y-z^2)*R
sage: I.radical()
Ideal (z^2 - y, y^2*z + y*z + 2*y + 2) of Multivariate Polynomial Ring in x, y, z over Ratio
```

Note: From the Singular manual: A combination of the algorithms of Krick/Logar and Kemper is used. Works also in positive characteristic (Kempers algorithm).

```
sage: R.<x,y,z> = PolynomialRing(GF(37), 3)
sage: p = z^2 + 1; q = z^3 + 2
sage: I = (p*q^2, y - z^2)*R
sage: I.radical()
Ideal (z^2 - y, y^2*z + y*z + 2*y + 2) of Multivariate Polynomial Ring in x, y, z over Finit
```

reduced_basis(**args, **kwds*)

Warning: This function is deprecated. It will be removed in a future release of Sage. Please use the `interreduced_basis()` function instead.

If this ideal is spanned by $(f_1, ..., f_n)$ this method returns $(g_1, ..., g_s)$ such that:

- $(f_1, ..., f_n) = (g_1, ..., g_s)$
- $LT(g_i) != LT(g_j)$ for all $i != j$
- $LT(g_i)$ does not divide m for all monomials m of $\{g_1, ..., g_{i-1}, g_{i+1}, ..., g_s\}$
- $LC(g_i) == 1$ for all i.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: I = Ideal([z*x+y^3,z+y^3,z+x*y])
sage: I.reduced_basis()
doctest:...: DeprecationWarning: This function is deprecated. It will be removed in a future
[x*z - z, x*y + z, y^3 + z]

sage: R.<x,y,z> = PolynomialRing(QQ,order='negdegrevlex')
sage: I = Ideal([z*x+y^3,z+y^3,z+x*y])
sage: I.reduced_basis()
[x*z + y^3, x*y - y^3, z + y^3]
```

ALGORITHM:

Uses Singular's interred command or `toy_buchberger.inter_reduction` if conversion to Singular fails.

syzygy_module()

Computes the first syzygy (i.e., the module of relations of the given generators) of the ideal.

EXAMPLE:

```
sage: R.<x,y> = PolynomialRing(QQ)
sage: f = 2*x^2 + y
sage: g = y
sage: h = 2*f + g
sage: I = Ideal([f,g,h])
sage: M = I.syzygy_module(); M
[       -2        -1         1]
[       -y 2*x^2 + y         0]
sage: G = vector(I.gens())
```

```
sage: M*G
(0, 0)
```

ALGORITHM: Uses Singular's syz command

transformed_basis(**args, **kwds*)

Returns a lex or other_ring Groebner Basis for this ideal.

INPUT:

- algorithm - see below for options.
- other_ring - only valid for algorithm 'fglm', if provided conversion will be performed to this ring. Otherwise a lex Groebner basis will be returned.

ALGORITHMS:

fglm FGLM algorithm. The input ideal must be given with a reduced Groebner Basis of a zero-dimensional ideal

gwalk Groebner Walk algorithm (*default*)

awalk1 'first alternative' algorithm

awalk2 'second alternative' algorithm

twalk Tran algorithm

fwalk Fractal Walk algorithm

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: I = Ideal([y^3+x^2,x^2*y+x^2, x^3-x^2, z^4-x^2-y])
sage: I = Ideal(I.groebner_basis())
sage: S.<z,x,y> = PolynomialRing(QQ,3,order='lex')
sage: J = Ideal(I.transformed_basis('fglm',S))
sage: J
Ideal (y^4 + y^3, x*y^3 - y^3, x^2 + y^3, z^4 + y^3 - y)
 of Multivariate Polynomial Ring in z, x, y over Rational Field

sage: R.<z,y,x>=PolynomialRing(GF(32003),3,order='lex')
sage: I=Ideal([y^3+x*y*z+y^2*z+x*z^3,3+x*y+x^2*y+y^2*z])
sage: I.transformed_basis('gwalk')
[y^9 - y^7*x^2 - y^7*x - y^6*x^3 - y^6*x^2 - 3*y^6 - 3*y^5*x - y^3*x^7
 - 3*y^3*x^6 - 3*y^3*x^5 - y^3*x^4 - 9*y^2*x^5 - 18*y^2*x^4 - 9*y^2*x^3
 - 27*y*x^3 - 27*y*x^2 - 27*x,
 z*x + 8297*y^8*x^2 + 8297*y^8*x + 3556*y^7 - 8297*y^6*x^4 + 15409*y^6*x^3 - 8297*y^6*x^2
 - 8297*y^5*x^5 + 15409*y^5*x^4 - 8297*y^5*x^3 + 3556*y^5*x^2 + 3556*y^5*x + 3556*y^4*x^3
 + 3556*y^4*x^2 - 10668*y^4 - 10668*y^3*x - 8297*y^2*x^9 - 1185*y^2*x^8 + 14224*y^2*x^7
 - 1185*y^2*x^6 - 8297*y^2*x^5 - 14223*y*x^7 - 10666*y*x^6 - 10666*y*x^5 - 14223*y*x^4
 + x^5 + 2*x^4 + x^3, z*y^2 + y*x^2 + y*x + 3]
```

ALGORITHM: Uses Singular

triangular_decomposition(*algorithm=None, singular=Singular*)

Decompose zero-dimensional ideal self into triangular sets.

This requires that the given basis is reduced w.r.t. to the lexicographical monomial ordering. If the basis of self does not have this property, the required Groebner basis is computed implicitly.

INPUT:

- algorithm - string or None (default: None)
  ALGORITHMS:
  - singular:triangL - decomposition of self into triangular systems (Lazard).
  - singular:triangLfak - decomp. of self into tri. systems plus factorization.
  - singular:triangM - decomposition of self into triangular systems (Moeller).

OUTPUT: a list T of lists t such that the variety of `self` is the union of the varieties of t in L and each t is in triangular form.

EXAMPLE:

```
sage: P.<e,d,c,b,a> = PolynomialRing(QQ,5,order='lex')
sage: I = sage.rings.ideal.Cyclic(P)
sage: GB = Ideal(I.groebner_basis('singular:stdfglm'))
sage: GB.triangular_decomposition('singular:triangLfak')
[Ideal (a - 1, b - 1, c - 1, d^2 + 3*d + 1, e + d + 3) of Multivariate Polynomial Ring in e,
Ideal (a - 1, b - 1, c^2 + 3*c + 1, d + c + 3, e - 1) of Multivariate Polynomial Ring in e,
Ideal (a - 1, b^2 + 3*b + 1, c + b + 3, d - 1, e - 1) of Multivariate Polynomial Ring in e,
Ideal (a - 1, b^4 + b^3 + b^2 + b + 1, c - b^2, d - b^3, e + b^3 + b^2 + b + 1) of Multivari
Ideal (a^2 + 3*a + 1, b - 1, c - 1, d - 1, e + a + 3) of Multivariate Polynomial Ring in e,
Ideal (a^2 + 3*a + 1, b + a + 3, c - 1, d - 1, e - 1) of Multivariate Polynomial Ring in e,
Ideal (a^4 - 4*a^3 + 6*a^2 + a + 1, 11*b^2 - 6*b*a^3 + 26*b*a^2 - 41*b*a + 4*b + 8*a^3 - 31*
Ideal (a^4 + a^3 + a^2 + a + 1, b - 1, c + a^3 + a^2 + a + 1, d - a^3, e - a^2) of Multivari
Ideal (a^4 + a^3 + a^2 + a + 1, b - a, c - a, d^2 + 3*d*a + a^2, e + d + 3*a) of Multivariat
Ideal (a^4 + a^3 + a^2 + a + 1, b - a, c^2 + 3*c*a + a^2, d + c + 3*a, e - a) of Multivariat
Ideal (a^4 + a^3 + a^2 + a + 1, b^2 + 3*b*a + a^2, c + b + 3*a, d - a, e - a) of Multivariat
Ideal (a^4 + a^3 + a^2 + a + 1, b^3 + b^2*a + b^2 + b*a^2 + b*a + b + a^3 + a^2 + a + 1, c +
Ideal (a^4 + a^3 + 6*a^2 - 4*a + 1, 11*b^2 - 6*b*a^3 - 10*b*a^2 - 39*b*a - 2*b - 16*a^3 - 23
```

variety(*ring=None, proof=True*)

Return the variety of `self`.

Given a zero-dimensional ideal I (== `self`) of a polynomial ring P whose order is lexicographic, return the variety of I as a list of dictionaries with (variable, value) pairs. By default, the variety of the ideal over its coefficient field K is returned; `ring` can be specified to find the variety over a different ring.

These dictionaries have cardinality equal to the number of variables in P and represent assignments of values to these variables such that all polynomials in I vanish.

If `ring` is specified, then a triangular decomposition of `self` is found over the original coefficient field K; then the triangular systems are solved using root-finding over `ring`. This is particularly useful when K is `QQ` (to allow fast symbolic computation of the triangular decomposition) and `ring` is `RR`, `AA`, `CC`, or `QQbar` (to compute the whole real or complex variety of the ideal).

Note that with `ring``=``RR` or `CC`, computation is done numerically and potentially inaccurately; in particular, the number of points in the real variety may be miscomputed. With `ring``=``AA` or `QQbar`, computation is done exactly (which may be much slower, of course).

INPUT:

- `ring` - return roots in the `ring` instead of the base ring of this ideal (default: `None`)
- `proof` - return a provably correct result (default: `True`)

EXAMPLE:

```
sage: K.<w> = GF(27) # this example is from the MAGMA handbook
sage: P.<x, y> = PolynomialRing(K, 2, order='lex')
sage: I = Ideal([ x^8 + y + 2, y^6 + x*y^5 + x^2 ])
sage: I = Ideal(I.groebner_basis()); I
Ideal (x - y^47 - y^45 + y^44 - y^43 + y^41 - y^39 - y^38
- y^37 - y^36 + y^35 - y^34 - y^33 + y^32 - y^31 + y^30 +
y^28 + y^27 + y^26 + y^25 - y^23 + y^22 + y^21 - y^19 -
y^18 - y^16 + y^15 + y^13 + y^12 - y^10 + y^9 + y^8 + y^7
- y^6 + y^4 + y^3 + y^2 + y - 1, y^48 + y^41 - y^40 + y^37
- y^36 - y^33 + y^32 - y^29 + y^28 - y^25 + y^24 + y^2 + y
+ 1) of Multivariate Polynomial Ring in x, y over Finite
Field in w of size 3^3

sage: V = I.variety(); V
[{y: w^2 + 2, x: 2*w}, {y: w^2 + w, x: 2*w + 1}, {y: w^2 + 2*w, x: 2*w + 2}]
```

```
sage: [f.subs(v) for f in I.gens() for v in V] # check that all polynomials vanish
[0, 0, 0, 0, 0, 0]
```

However, we only account for solutions in the ground field and not in the algebraic closure:

```
sage: I.vector_space_dimension()
48
```

Here we compute the points of intersection of a hyperbola and a circle, in several fields:

```
sage: K.<x, y> = PolynomialRing(QQ, 2, order='lex')
sage: I = Ideal([ x*y - 1, (x-2)^2 + (y-1)^2 - 1])
sage: I = Ideal(I.groebner_basis()); I
Ideal (x + y^3 - 2*y^2 + 4*y - 4, y^4 - 2*y^3 + 4*y^2 - 4*y + 1)
of Multivariate Polynomial Ring in x, y over Rational Field
```

These two curves have one rational intersection:

```
sage: I.variety()
[{y: 1, x: 1}]
```

There are two real intersections:

```
sage: I.variety(ring=RR)
[{y: 0.361103080528647, x: 2.76929235423863},
 {y: 1.00000000000000, x: 1.00000000000000}]
sage: I.variety(ring=AA)
[{x: 2.769292354238632?, y: 0.3611030805286474?},
 {x: 1, y: 1}]
```

and a total of four intersections:

```
sage: I.variety(ring=CC)
[{y: 0.31944845973567... - 1.6331702409152...*I,
  x: 0.11535382288068... + 0.58974280502220...*I},
 {y: 0.31944845973567... + 1.6331702409152...*I,
  x: 0.11535382288068... - 0.58974280502220...*I},
 {y: 0.36110308052864..., x: 2.7692923542386...},
 {y: 1.00000000000000, x: 1.00000000000000}]
sage: I.variety(ring=QQbar)
[{y: 0.3194484597356763? - 1.633170240915238?*I,
  x: 0.11535382288068429? + 0.5897428050222055?*I},
 {y: 0.3194484597356763? + 1.633170240915238?*I,
  x: 0.11535382288068429? - 0.5897428050222055?*I},
 {y: 0.3611030805286474?, x: 2.769292354238632?},
 {y: 1, x: 1}]
```

If the ground field's characteristic is too large for Singular, we resort to a toy implementation:

```
sage: R.<x,y> = PolynomialRing(GF(2147483659),order='lex')
sage: I=ideal([x^3-2*y^2,3*x+y^4])
sage: I.variety()
verbose 0 (...: multi_polynomial_ideal.py, variety) Warning: falling back to very slow toy i
verbose 0 (...: multi_polynomial_ideal.py, groebner_basis) Warning: falling back to very slo
[{y: 0, x: 0}]
```

TESTS:

```
sage: K.<w> = GF(27)
sage: P.<x, y> = PolynomialRing(K, 2, order='lex')
sage: I = Ideal([ x^8 + y + 2, y^6 + x*y^5 + x^2 ])
```

Testing the robustness of the Singular interface

```
sage: T = I.triangular_decomposition('singular:triangLfak')
sage: I.variety()
[{y: w^2 + 2, x: 2*w}, {y: w^2 + w, x: 2*w + 1}, {y: w^2 + 2*w, x: 2*w + 2}]
```

Testing that a bug is indeed fixed.

```
sage: R = PolynomialRing(GF(2), 30, ['x%d'%(i+1) for i in range(30)], order='lex')
sage: R.inject_variables()
Defining...
sage: I = Ideal([x1 + 1, x2, x3 + 1, x5*x10 + x10 + x18, x5*x11 + x11, \
                 x5*x18, x6, x7 + 1, x9, x10*x11 + x10 + x18, x10*x18 + x18, \
                 x11*x18, x12, x13, x14, x15, x16 + 1, x17 + x18 + 1, x19, x20, \
                 x21 + 1, x22, x23, x24, x25 + 1, x28 + 1, x29 + 1, x30, x8, \
                 x26, x1^2 + x1, x2^2 + x2, x3^2 + x3, x4^2 + x4, x5^2 + x5, \
                 x6^2 + x6, x7^2 + x7, x8^2 + x8, x9^2 + x9, x10^2 + x10, \
                 x11^2 + x11, x12^2 + x12, x13^2 + x13, x14^2 + x14, x15^2 + x15, \
                 x16^2 + x16, x17^2 + x17, x18^2 + x18, x19^2 + x19, x20^2 + x20, \
                 x21^2 + x21, x22^2 + x22, x23^2 + x23, x24^2 + x24, x25^2 + x25, \
                 x26^2 + x26, x27^2 + x27, x28^2 + x28, x29^2 + x29, x30^2 + x30])
sage: I.basis_is_groebner()
True
sage: for V in I.variety():
...     print V
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 0, x19: 0, x18: 1, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 1, x19: 0, x18: 1, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 0, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 1, x4: 1, x19: 0, x18: 0, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 0, x19: 0, x18: 1, x7: 1, x6: 0, x1
{x14: 0, x24: 0, x16: 1, x1: 1, x3: 1, x2: 0, x5: 0, x4: 1, x19: 0, x18: 1, x7: 1, x6: 0, x1
```

ALGORITHM: Uses triangular decomposition.

vector_space_dimension()

Return the vector space dimension of the ring modulo this ideal. If the ideal is not zero-dimensional, a TypeError is raised.

ALGORITHM: Uses Singular.

EXAMPLE:

```
sage: R.<u,v> = PolynomialRing(QQ)
sage: g = u^4 + v^4 + u^3 + v^3
sage: I = ideal(g) + ideal(g.gradient())
sage: I.dimension()
0
sage: I.vector_space_dimension()
4
```

class RedSBContext (*singular=Singular*)
Within this context all Singular Groebner basis calculations are reduced automatically.

AUTHORS:

•Martin Albrecht

is_MPolynomialIdeal (*x*)
Return `True` if the provided argument `x` is an ideal in the multivariate polynomial ring.

INPUT:

•`x` - an arbitrary object

EXAMPLES:

```
sage: from sage.rings.polynomial.all import is_MPolynomialIdeal
sage: P.<x,y,z> = PolynomialRing(QQ)
sage: I = [x + 2*y + 2*z - 1, x^2 + 2*y^2 + 2*z^2 - x, 2*x*y + 2*y*z - y]
```

Sage distinguishes between a list of generators for an ideal and the ideal itself. This distinction is inconsistent with Singular but matches Magma's behavior.

```
sage: is_MPolynomialIdeal(I)
False


sage: I = Ideal(I)
sage: is_MPolynomialIdeal(I)
True
```

redSB (*func*)
Decorator to force a reduced Singular groebner basis.

TESTS:

```
sage: P.<a,b,c,d,e> = PolynomialRing(GF(127))
sage: J = sage.rings.ideal.Cyclic(P).homogenize()
sage: from sage.misc.sageinspect import sage_getsource
sage: "buchberger" in sage_getsource(J.interreduced_basis)
True
```

Note: This decorator is used automatically internally so the user does not need to use it manually.

28.11 Symmetric Ideals of Infinite Polynomial Rings

This module provides an implementation of ideals of polynomial rings in a countably infinite number of variables that are invariant under variable permuation. Such ideals are called 'Symmetric Ideals' in the rest of this document. Our implementation is based on the theory of M. Aschenbrenner and C. Hillar.

AUTHORS:

- Simon King <simon.king@uni-jena.de>

class SymmetricIdeal (*ring, gens, coerce=True*)
Ideal in an Infinite Polynomial Ring, invariant under permutation of variable indices

THEORY:

An Infinite Polynomial Ring with finitely many generators $x_*, y_*, ...$ over a field F is a free commutative F-algebra generated by infinitely many 'variables' $x_0, x_1, x_2, ..., y_0, y_1, y_2, ...$ We refer to the natural number n as the *index* of the variable x_n. See more detailed description at `infinite_polynomial_ring`

Infinite Polynomial Rings are equipped with a permutation action by permuting positive variable indices, i.e., $x_n^P = x_{P(n)}, y_n^P = y_{P(n)}, ...$ for any permutation P. Note that the variables $x_0, y_0, ...$ of index zero are invariant under that action.

A *Symmetric Ideal* is an ideal in an infinite polynomial ring X that is invariant under the permutation action. In other words, if $\mathfrak{S}_\infty$ denotes the symmetric group of $1, 2, ...$, then a Symmetric Ideal is a right $X[\mathfrak{S}_\infty]$-submodule of X.

It is known by work of Aschenbrenner and Hillar [AB2007] that an Infinite Polynomial Ring X with a single generator x_* is noetherian, in the sense that any Symmetric Ideal $I \subset X$ is finitely generated modulo addition, multiplication by elements of X, and permutation of variable indices (hence, it is a finitely generated right $X[\mathfrak{S}_\infty]$-module).

Moreover, if X is equipped with a lexicographic monomial ordering with $x_1 < x_2 < x_3...$ then there is an algorithm of Buchberger type that computes a Groebner basis G for I that allows for computation of a unique normal form, that is zero precisely for the elements of I – see [AB2008]. See `groebner_basis()` for more details.

Our implementation allows more than one generator and also provides degree lexicographic and degree reverse lexicographic monomial orderings – we do, however, not guarantee termination of the Buchberger algorithm in these cases.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: I=X*(x[1]^2+y[2]^2,x[1]*x[2]*y[3]+x[1]*y[4])
sage: I == loads(dumps(I))
True
sage: latex(I)
\left(y_{2}^{2} + x_{1}^{2}, y_{4} x_{1} + y_{3} x_{2} x_{1}\right)\Bold{Q}[x_{\ast}, y_{\ast}][
```

The default ordering is lexicographic. We now compute a Groebner basis:

```
sage: J=I.groebner_basis()
sage: J
[x1^4 + x1^3, x2*x1^2 - x1^3, x2^2 - x1^2, y1*x1^3 + y1*x1^2, y1*x2 + y1*x1^2, y1^2 + x1^2, y2*x
```

Ideal membership in `I` can now be tested by commuting symmetric reduction modulo `J`:

```
sage: I.reduce(J)
Symmetric Ideal (0, 0) of Infinite polynomial ring in x, y over Rational Field
```

Note that the Groebner basis is not point-wise invariant under permutation. However, any element of `J` has symmetric reduction zero even after applying a permutation:

```
sage: P=Permutation([1, 4, 3, 2])
sage: J[2]
x2^2 - x1^2
sage: J[2]^P
x4^2 - x1^2
sage: J.__contains__(J[2]^P)
False
sage: [[(p^P).reduce(J) for p in J] for P in Permutations(4)]
[[0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 0, 0, 0],
```

```
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0]]
```

Since `I` is not a Groebner basis, it is no surprise that it can not detect ideal membership:

```
sage: [p.reduce(I) for p in J]
[x1^4 + x1^3, x2*x1^2 - x1^3, x2^2 - x1^2, y1*x1^3 + y1*x1^2, y1*x2 + y1*x1^2, y1^2 + x1^2, y2*x
```

Note we give no guarantee that the computation of a symmetric Groebner basis will terminate in an order different from lexicographic.

When multiplying Symmetric Ideals or raising them to some integer power, the permutation action is taken into account, so that the product is indeed the product of ideals in the mathematical sense.

```
sage: I=X*(x[1])
sage: I*I
Symmetric Ideal (x1^2, x2*x1) of Infinite polynomial ring in x, y over Rational Field
sage: I^3
Symmetric Ideal (x1^3, x2*x1^2, x2^2*x1, x3*x2*x1) of Infinite polynomial ring in x, y over Rati
sage: I*I == X*(x[1]^2)
False
```

groebner_basis()

Return a symmetric Groebner basis (type `Sequence`) of self.

INPUT:

- `tailreduce` (optional, default False) - if True, use tail reduction in intermediate computations
- `reduced` (optional, default True) - return the reduced normalised Groebner basis
- `algorithm` (optional) - determine the algorithm (see below for available algorithms)
- `report` (optional) - print information on the progress of computation.
- `use_full_group` (optional, default False) - if True then proceed as originally suggested by [AB2008]. Our default method should be faster, see `symmetrisation()` for more details.

The computation of symmetric Groebner bases also involves the computation of *classical* Groebner bases, i.e., of Groebner bases for ideals in polynomial rings with finitely many variables. For these computations, Sage provides the following ALGORITHMS:

'' autoselect (default)

'singular:groebner' Singular's `groebner` command

'singular:std' Singular's `std` command

'singular:stdhilb' Singular's `stdhib` command
'singular:stdfglm' Singular's `stdfglm` command
'singular:slimgb' Singular's `slimgb` command
'libsingular:std' libSingular's `std` command
'libsingular:slimgb' libSingular's `slimgb` command
'toy:buchberger' Sage's toy/educational buchberger without strategy
'toy:buchberger2' Sage's toy/educational buchberger with strategy
'toy:d_basis' Sage's toy/educational d_basis algorithm
'macaulay2:gb' Macaulay2's `gb` command (if available)
'magma:GroebnerBasis' Magma's `Groebnerbasis` command (if available)

If only a system is given - e.g. 'magma' - the default algorithm is chosen for that system.

Note: The Singular and libSingular versions of the respective algorithms are identical, but the former calls an external Singular process while the later calls a C function, i.e. the calling overhead is smaller.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: I1 = X*(x[1]+x[2],x[1]*x[2])
sage: I1.groebner_basis()
[x1]
sage: I2 = X*(y[1]^2*y[3]+y[1]*x[3])
sage: I2.groebner_basis()
[y1*x2^2 - y1*x2*x1, y2*x2*x1 - y2*x1^2, y2*y1*x2 - y2*y1*x1, y2*y1^2 + y1*x2, y2^2*y1 + y2*
```

When using the algorithm originally suggested by Aschenbrenner and Hillar, the result is the same, but the computation takes much longer:

```
sage: I2.groebner_basis(use_full_group=True)
[y1*x2^2 - y1*x2*x1, y2*x2*x1 - y2*x1^2, y2*y1*x2 - y2*y1*x1, y2*y1^2 + y1*x2, y2^2*y1 + y2*
```

Last, we demonstrate how the report on the progress of computations looks like:

```
sage: I1.groebner_basis(report=True, reduced=True)
Symmetric interreduction
[1/2]  >
[2/2] : >
[1/2]  >
[2/2]  >
Symmetrise 2 polynomials at level 2
Apply permutations
>
>
Symmetric interreduction
[1/3]  >
[2/3]  >
[3/3] : >
-> 0
[1/2]  >
[2/2]  >
Symmetrisation done
Classical Groebner basis
-> 2 generators
Symmetric interreduction
[1/2]  >
[2/2]  >
Symmetrise 2 polynomials at level 3
Apply permutations
>
```

```
>
:>
::>
:>
::>
Symmetric interreduction
[1/4]  >
[2/4] : >
-> 0
[3/4] :: >
-> 0
[4/4] : >
-> 0
[1/1]  >
Apply permutations
:>
:>
:>
Symmetric interreduction
[1/1]  >
Classical Groebner basis
-> 1 generators
Symmetric interreduction
[1/1]  >
Symmetrise 1 polynomials at level 4
Apply permutations
>
:>
:>
>
:>
:>
Symmetric interreduction
[1/2]  >
[2/2] : >
-> 0
[1/1]  >
Symmetric interreduction
[1/1]  >
[x1]
```

interreduced_basis()

A fully symmetrically reduced generating set (type Sequence) of self.

This does essentially the same as interreduction() with the option 'tailreduce', but it returns a Sequence rather than a SymmetricIdeal.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I=X*(x[1]+x[2],x[1]*x[2])
sage: I.interreduced_basis()
[-x1^2, x2 + x1]
```

interreduction(*tailreduce=True, sorted=False, report=None, RStrat=None*)

Return symmetrically interreduced form of self

INPUT:

- tailreduce (optional) - If True, the interreduction is also performed on the non-leading monomials.
- sorted (optional) - If True, it is assumed that the generators of self are already increasingly sorted.

- report (optional) - If not None, some information on the progress of computation is printed
- RStrat (optional) - A SymmetricReductionStrategy to which the polynomials resulting from the interreduction will be added. If RStrat already contains some polynomials, they will be used in the interreduction. The effect is to compute in a quotient ring.

RETURN: A Symmetric Ideal J (sorted list of generators) coinciding with self as an ideal, so that any generator is symmetrically reduced w.r.t. the other generators. Note that the leading coefficients of the result are not necessarily 1.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I=X*(x[1]+x[2],x[1]*x[2])
sage: I.interreduction()
Symmetric Ideal (-x1^2, x2 + x1) of Infinite polynomial ring in x over Rational Field
```

Here, we show the report option:

```
sage: I.interreduction(report=True)
Symmetric interreduction
[1/2]  >
[2/2] : >
[1/2]  >
[2/2] T[1] >
>
Symmetric Ideal (-x1^2, x2 + x1) of Infinite polynomial ring in x over Rational Field
```

[m/n] indicates that polynomial number m is considered and the total number of polynomials under consideration is n. '-> 0' is printed if a zero reduction occured. The rest of the report is as described in sage.rings.polynomial.symmetric_reduction.SymmetricReductionStrategy.reduce().

Last, we demonstrate the use of the optional parameter RStrat:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: R = SymmetricReductionStrategy(X)
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in x over Rational Field
sage: I.interreduction(RStrat=R)
Symmetric Ideal (-x1^2, x2 + x1) of Infinite polynomial ring in x over Rational Field
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in x over Rational Field, modulo
    x1^2,
    x2 + x1
sage: R = SymmetricReductionStrategy(X,[x[1]^2])
sage: I.interreduction(RStrat=R)
Symmetric Ideal (x2 + x1) of Infinite polynomial ring in x over Rational Field
```

normalisation()

Return an ideal that coincides with self, so that all generators have leading coefficient 1.

Possibly occuring zeroes are removed from the generator list.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X*(1/2*x[1]+2/3*x[2], 0, 4/5*x[1]*x[2])
sage: I.normalisation()
Symmetric Ideal (x2 + 3/4*x1, x2*x1) of Infinite polynomial ring in x over Rational Field
```

reduce(*I, tailreduce=False*)

Symmetric reduction of self by another Symmetric Ideal or list of Infinite Polynomials.

INPUT:

- I – a Symmetric Ideal or a list of Infinite Polynomials

- `tailreduce` (optional) – if it is True, the non-leading terms will be reduced as well.

Reducing an element p of an Infinite Polynomial Ring X by some other element q means the following:

1. Let M and N be the leading terms of p and q.
2. Test whether there is a permutation P that does not does not diminish the variable indices occoring in N and preserves their order, so that there is some term $T \in X$ with $TN^P = M$. If there is no such permutation, return p
3. Replace p by $p - Tq^P$ and continue with step 1.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: I = X*(y[1]^2*y[3]+y[1]*x[3])
sage: I.reduce([y[2]^2*y[1]])
Symmetric Ideal (y3*y1^2 + y1*x3) of Infinite polynomial ring in x, y over Rational Field
```

The preceding is correct, since any permutation that turns `y[2]^2*y[1]` into a factor of `y[1]^2*y[3]` interchanges the variable indices 1 and 2 – which is not allowed. However, reduction by `y[1]^2*y[2]` works, since one can change variable index 1 into 2 and 2 into 3:

```
sage: I.reduce([y[1]^2*y[2]])
Symmetric Ideal (y1*x3) of Infinite polynomial ring in x, y over Rational Field
```

The next example shows that tail reduction is not done, unless it is explicitly adviced. The input can also be a symmetric ideal:

```
sage: J = (x[2])*X
sage: I.reduce(J)
Symmetric Ideal (y3*y1^2 + y1*x3) of Infinite polynomial ring in x, y over Rational Field
sage: I.reduce(J, tailreduce=True)
Symmetric Ideal (y3*y1^2) of Infinite polynomial ring in x, y over Rational Field
```

squeezed()

Reduce the variable indices occuring in self

OUTPUT: A Symmetric Ideal whose generators are the result of applying `squeezed()` to the generators of self.

NOTE: The output describes the same Symmetric Ideal as self.

EXAMPLES:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ,implementation='sparse')
sage: I = X*(x[1000]*y[100],x[50]*y[1000])
sage: I.squeezed()
Symmetric Ideal (y1*x2, y2*x1) of Infinite polynomial ring in x, y over Rational Field
```

symmetric_basis()

A symmetrised generating set (type `Sequence`) of self.

This does essentially the same as `symmetrisation()` with the option 'tailreduce', and it returns a `Sequence` rather than a `SymmetricIdeal`.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X*(x[1]+x[2], x[1]*x[2])
sage: I.symmetric_basis()
[x1^2, x2 + x1]
```

symmetrisation(*N=None, tailreduce=False, report=None, use_full_group=False*)

Apply permutations to the generators of self and interreduce

INPUT:

- N (optional) – apply permutations in Sym(N). Default is the maximal variable index occuring in the generators of `self.interreduction().squeezed()`.
- tailreduce (optional) – If True, perform tail reductions
- report (optional) – If not None, report on the progress of computations
- use_full_group (optional) – If True, apply *all* elements of Sym(N) to the generators of self (this is what [AB2008] originally suggests). The default is to apply all elementary transpositions to the generators of `self.squeezed()`, interreduce, and repeat until the result stabilises, which is often much faster than applying all of Sym(N), and we are convinced that both methods yield the same result.

OUTPUT: A symetrically interreduced symmetric ideal with respect to which any Sym(N)-translate of a generator of self is symmetrically reducible, where by default N is the maximal variable index that occurs in the generators of `self.interreduction().squeezed()`.

NOTE: If `I` is a symmetric ideal whose generators are monomials, then `I.symmetrisation()` is its reduced Groebner basis. It should be noted that without symmetrisation, monomial generators, in general, do not form a Groebner basis.

EXAMPLES:

```
sage: X.<x> = InfinitePolynomialRing(QQ)
sage: I = X*(x[1]+x[2], x[1]*x[2])
sage: I.symmetrisation()
Symmetric Ideal (-x1^2, x2 + x1) of Infinite polynomial ring in x over Rational Field
sage: I.symmetrisation(N=3)
Symmetric Ideal (-2*x1) of Infinite polynomial ring in x over Rational Field
sage: I.symmetrisation(N=3, use_full_group=True)
Symmetric Ideal (-2*x1) of Infinite polynomial ring in x over Rational Field
```

28.12 Symmetric Reduction of Infinite Polynomials

`SymmetricReductionStrategy` provides a framework for efficient symmetric reduction of Infinite Polynomials, see `infinite_polynomial_element`.

AUTHORS:

- Simon King <simon.king@uni-jena.de>

THEORY: According to M. Aschenbrenner and C. Hillar [AB2007], Symmetric Reduction of an element p of an Infinite Polynomial Ring X by some other element q means the following:

1. Let M and N be the leading terms of p and q.
2. Test whether there is a permutation P that does not does not diminish the variable indices occuring in N and preserves their order, so that there is some term $T \in X$ with $TN^P = M$. If there is no such permutation, return p
3. Replace p by $p - Tq^P$ and continue with step 1.

When reducing one polynomial p with respect to a list L of other polynomials, there usually is a choice of order on which the efficiency crucially depends. Also it helps to modify the polynomials on the list in order to simplify the basic reduction steps.

The preparation of L may be expensive. Hence, if the same list is used many times then it is reasonable to perform the preparation only once. This is the background of `SymmetricReductionStrategy`.

Our current strategy is to keep the number of terms in the polynomials as small as possible. For this, we sort L by increasing number of terms. If several elements of L allow for a reduction of p, we chose the one with the smallest number of terms. Later on, it should be possible to implement further strategies for choice.

When adding a new polynomial q to L, we first reduce q with respect to L. Then, we test heuristically whether it is possible to reduce the number of terms of the elements of L by reduction modulo q. That way, we see best chances to keep the number of terms in intermediate reduction steps relatively small.

EXAMPLES:

First, we create an infinite polynomial ring and one of its elements:

```
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: p = y[1]^2*y[3]+y[1]*x[3]
```

We want to symmetrically reduce it by another polynomial. So, we put this other polynomial into a list and create a Symmetric Reduction Strategy object:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: S = SymmetricReductionStrategy(X, [y[2]^2*y[1]])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    y2^2*y1
sage: S.reduce(p)
y3*y1^2 + y1*x3
```

The preceding is correct, since any permutation that turns `y[2]^2*y[1]` into a factor of `y[1]^2*y[3]` interchanges the variable indices 1 and 2 – which is not allowed in a symmetric reduction. However, reduction by `y[1]^2*y[2]` works, since one can change variable index 1 into 2 and 2 into 3. So, we add this to `S`:

```
sage: S.add_generator(y[1]^2*y[2])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    y2*y1^2,
    y2^2*y1
sage: S.reduce(p)
y1*x3
```

The next example shows that tail reduction is not done, unless it is explicitly adviced:

```
sage: S.reduce(y[3] + 2*y[2]*y[1]^2 + 3*y[2]^2*y[1])
y3 + 3*y2^2*y1 + 2*y2*y1^2
sage: S.tailreduce(y[3] + 2*y[2]*y[1]^2 + 3*y[2]^2*y[1])
y3
```

However, it is possible to ask for tailreduction already when the Symmetric Reduction Strategy is created:

```
sage: S2 = SymmetricReductionStrategy(X, [y[2]^2*y[1],y[1]^2*y[2]], tailreduce=True)
sage: S2
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    y2*y1^2,
    y2^2*y1
with tailreduction
sage: S2.reduce(y[3] + 2*y[2]*y[1]^2 + 3*y[2]^2*y[1])
y3
```

class SymmetricReductionStrategy()

A framework for efficient symmetric reduction of InfinitePolynomial, see `infinite_polynomial_element`.

INPUT:

- Parent, an Infinite Polynomial Ring, see infinite_polynomial_element.
- L, a list of elements of Parent (default: L=[]) with respect to which reduction will be done
- good_input: If this optional parameter is true, it is assumed that each element of L is symmetrically reduced with respect to the previous elements of L

EXAMPLES:

```
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: S = SymmetricReductionStrategy(X, [y[2]^2*y[1],y[1]^2*y[2]], good_input=True)
sage: S.reduce(y[3] + 2*y[2]*y[1]^2 + 3*y[2]^2*y[1])
y3 + 3*y2^2*y1 + 2*y2*y1^2
sage: S.tailreduce(y[3] + 2*y[2]*y[1]^2 + 3*y[2]^2*y[1])
y3
```

add_generator()

Add another polynomial to self

NOTE: Previously added polynomials may be modified. All input is prepared in view of an efficient symmetric reduction.

EXAMPLES:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X)
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field
sage: S.add_generator(y[3] + y[1]*(x[3]+x[1]))
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    y3 + y1*x3 + y1*x1
```

Note that the first added polynomial will be simplified when adding a suitable second polynomial:

```
sage: S.add_generator(x[2]+x[1])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    y3,
    x2 + x1
```

By default, reduction is applied to any newly added polynomial. This can be avoided by specifying the optional parameter 'good_input':

```
sage: S.add_generator(y[2]+y[1]*x[2])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    y3,
    y2 + y1*x2,
    x2 + x1
sage: S.reduce(x[3]+x[2])
-2*x1
sage: S.add_generator(x[3]+x[2], good_input=True)
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    y3,
    x3 + x2,
    y2 + y1*x2,
    x2 + x1
```

In the previous example, x[3] + x[2] is added without being reduced to zero.

gens()

Return the list of Infinite Polynomials modulo which self reduces

EXAMPLES:

```
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: S = SymmetricReductionStrategy(X, [y[2]^2*y[1],y[1]^2*y[2]])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field, modulo
    y2*y1^2,
    y2^2*y1
sage: S.gens()
[y2*y1^2, y2^2*y1]
```

reduce()

Symmetric reduction of an infinite polynomial

By default, tail reduction is done if and only if it was specified when creating self. If the optional parameter 'notail' is True, tail reduction is avoided. However, we do *not* guarantee that there will be no tail reduction at all in that case.

If tail reduction shall be forced, use `tailreduce()`.

Information on the progress of computation can be obtained by specifying the optional parameter `report`.

EXAMPLES:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X, [x[3]], tailreduce=True)
sage: S.reduce(y[4]*x[1] + y[1]*x[4])
y4*x1
sage: S.reduce(y[4]*x[1] + y[1]*x[4], notail=True)
y4*x1 + y1*x4
```

Last, we demonstrate the 'report' option:

```
sage: S = SymmetricReductionStrategy(X, [y[2]+x[1],y[2]*y[3]+y[1]*x[2]+x[4],x[3]+x[2]])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    x3 + x2,
    y2 + x1,
    y1*x2 + x4 + x1^2
sage: S.reduce(y[3] + y[1]*x[3] + y[1]*x[1],report=True)
:::>
y1*x1 + x4 + x1^2 - x1
```

Each ':' indicates that one reduction of the leading monomial was performed. Eventually, the '>' indicates that the computation is finished.

reset()

Remove all polynomials from self

EXAMPLES:

```
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: S = SymmetricReductionStrategy(X, [y[2]^2*y[1],y[1]^2*y[2]])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field, modulo
    y2*y1^2,
    y2^2*y1
sage: S.reset()
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field
```

setgens()

Define the list of Infinite Polynomials modulo which self reduces

INPUT:

L, a list

NOTE:

It is not tested if L is a good input. That method simply assigns a *copy* of L to the generators of self.

EXAMPLES:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X, [y[2]^2*y[1],y[1]^2*y[2]])
sage: R = SymmetricReductionStrategy(X)
sage: R.setgens(S.gens())
sage: R
Symmetric Reduction Strategy in Infinite polynomial ring in y over Rational Field, modulo
    y2*y1^2,
    y2^2*y1
sage: R.gens() is S.gens()
False
sage: R.gens() == S.gens()
True
```

tailreduce()

Symmetric reduction of an infinite polynomial, with forced tail reduction

Information on the progress of computation can be obtained by specifying the optional parameter 'report'.

EXAMPLES:

```
sage: from sage.rings.polynomial.symmetric_reduction import SymmetricReductionStrategy
sage: X.<x,y> = InfinitePolynomialRing(QQ)
sage: S = SymmetricReductionStrategy(X, [x[3]])
sage: S.reduce(y[4]*x[1] + y[1]*x[4])
y4*x1 + y1*x4
sage: S.tailreduce(y[4]*x[1] + y[1]*x[4])
y4*x1
```

Last, we demonstrate the 'report' option:

```
sage: S = SymmetricReductionStrategy(X, [y[2]+x[1],y[2]*y[3]+y[1]*x[2]+x[4],x[3]+x[2]])
sage: S
Symmetric Reduction Strategy in Infinite polynomial ring in x, y over Rational Field, modulo
    x3 + x2,
    y2 + x1,
    y1*x2 + x4 + x1^2
sage: S.tailreduce(y[3] + y[1]*x[3] + y[1]*x[1],report=True)
T[3]:::>
T[3]:>
y1*x1 - x2 + x1^2 - x1
```

The protocol means the following.

- 'T[3]' means that we currently do tail reduction for a polynomial with three terms.
- ':::>' means that there were three reductions of leading terms.
- The tail of the result of the preceding reduction still has three terms. One reduction of leading terms was possible, and then the final result was obtained.

28.13 Educational Versions of Groebner Basis Algorithms: Triangular Factorization.

In this file is the implementation of two algorithms in [Laz92].

The main algorithm is `Triangular`; a secondary algorithm, necessary for the first, is `ElimPolMin`. As per Lazard's formulation, the implementation works with any term ordering, not only lexicographic.

Lazard does not specify a few of the subalgorithms implemented as the functions

- `is_triangular`,
- `is_linearly_dependent`, and
- `linear_representation`.

The implementations are not hard, and the choice of algorithm is described with the relevant function.

No attempt was made to optimize these algorithms as the emphasis of this implementation is a clean and easy presentation.

Examples appear with the appropriate function.

AUTHORS:

- John Perry (2009-02-24): initial version, but some words of documentation were stolen shamelessly from Martin Albrecht's `toy_buchberger.py`.

REFERENCES:

coefficient_matrix(*polys*)

Generates the matrix `M` whose entries are the coefficients of `polys`. The entries of row `i` of `M` consist of the coefficients of `polys[i]`.

INPUT:

•`polys` - a list/tuple of polynomials

OUTPUT:

A matrix `M` of the coefficients of `polys`.

EXAMPLE:

```
sage: from sage.rings.polynomial.toy_variety import coefficient_matrix
sage: R.<x,y> = PolynomialRing(QQ)
sage: coefficient_matrix([x^2 + 1, y^2 + 1, x*y + 1])
[1 0 0 1]
[0 0 1 1]
[0 1 0 1]
```

Note: This function may be merged with `sage.crypto.mq.MPolynomialSystem_generic.coefficient_matrix` in the future.

elim_pol(*B, n=-1*)

Finds the unique monic polynomial of lowest degree and lowest variable in the ideal described by `B`.

For the purposes of the triangularization algorithm, it is necessary to preserve the ring, so `n` specifies which variable to check. By default, we check the last one, which should also be the smallest.

The algorithm may not work if you are trying to cheat: `B` should describe the Groebner basis of a zero-dimensional ideal. However, it is not necessary for the Groebner basis to be lexicographic.

The algorithm is taken from a 1993 paper by Lazard [Laz92].

INPUT:

- `B` - a list/tuple of polynomials or a multivariate polynomial ideal
- `n` - the variable to check (see above) (default: `-1`)

EXAMPLE:

```
sage: set_verbose(0)
sage: from sage.rings.polynomial.toy_variety import elim_pol
sage: R.<x,y,z> = PolynomialRing(GF(32003))
sage: p1 = x^2*(x-1)^3*y^2*(z-3)^3
sage: p2 = z^2 - z
sage: p3 = (x-2)^2*(y-1)^3
sage: I = R.ideal(p1,p2,p3)
sage: elim_pol(I.groebner_basis())
z^2 - z
```

is_linearly_dependent(*polys*)

Decides whether the polynomials of `polys` are linearly dependent. Here `polys` is a collection of polynomials.

The algorithm creates a matrix of coefficients of the monomials of `polys`. It computes the echelon form of the matrix, then checks whether any of the rows is the zero vector.

Essentially this relies on the fact that the monomials are linearly independent, and therefore is building a linear map from the vector space of the monomials to the canonical basis of `R^n`, where `n` is the number of distinct monomials in `polys`. There is a zero vector iff there is a linear dependence among `polys`.

The case where `polys=[]` is considered to be not linearly dependent.

INPUT:

- `polys` - a list/tuple of polynomials

OUTPUT:

`True` if the elements of `polys` are linearly dependent; `False` otherwise.

EXAMPLE:

```
sage: from sage.rings.polynomial.toy_variety import is_linearly_dependent
sage: R.<x,y> = PolynomialRing(QQ)
sage: B = [x^2 + 1, y^2 + 1, x*y + 1]
sage: p = 3*B[0] - 2*B[1] + B[2]
sage: is_linearly_dependent(B + [p])
True
sage: p = x*B[0]
sage: is_linearly_dependent(B + [p])
False
sage: is_linearly_dependent([])
False
```

is_triangular(*B*)

Check whether the basis `B` of an ideal is triangular. That is: check whether the largest variable in `B[i]` with respect to the ordering of the base ring `R` is `R.gens()[i]`.

The algorithm is based on the definition of a triangular basis, given by Lazard in 1992 in [Laz92].

INPUT:

- `B` - a list/tuple of polynomials or a multivariate polynomial ideal

OUTPUT:

`True` if the basis is triangular; `False` otherwise.

EXAMPLE:

```
sage: from sage.rings.polynomial.toy_variety import is_triangular
sage: R.<x,y,z> = PolynomialRing(QQ)
sage: p1 = x^2*y + z^2
sage: p2 = y*z + z^3
sage: p3 = y+z
sage: is_triangular(R.ideal(p1,p2,p3))
False
sage: p3 = z^2 - 3
sage: is_triangular(R.ideal(p1,p2,p3))
True
```

linear_representation(*p, polys*)

Assuming that `p` is a linear combination of `polys`, determines coefficients that describe the linear combination. This probably doesn't work for any inputs except `p`, a polynomial, and `polys`, a sequence of polynomials. If `p` is not in fact a linear combination of `polys`, the function raises an exception.

The algorithm creates a matrix of coefficients of the monomials of `polys` and `p`, with the coefficients of `p` in the last row. It augments this matrix with the appropriate identity matrix, then computes the echelon form of the augmented matrix. The last row should contain zeroes in the first columns, and the last columns contain a linear dependence relation. Solving for the desired linear relation is straightforward.

INPUT:

- `p` - a polynomial
- `polys` - a list/tuple of polynomials

OUTPUT:

If `n == len(polys)`, returns `[a[0],a[1],...,a[n-1]]` such that `p == a[0]*poly[0] + ... + a[n-1]*poly[n-1]`.

EXAMPLE:

```
sage: from sage.rings.polynomial.toy_variety import linear_representation
sage: R.<x,y> = PolynomialRing(GF(32003))
sage: B = [x^2 + 1, y^2 + 1, x*y + 1]
sage: p = 3*B[0] - 2*B[1] + B[2]
sage: linear_representation(p, B)
[3, 32001, 1]
```

triangular_factorization(*B, n=-1, proof=True*)

Compute the triangular factorization of the Groebner basis `B` of an ideal.

This will not work properly if `B` is not a Groebner basis!

The algorithm used is that described in a 1992 paper by Daniel Lazard [Laz92]. It is not necessary for the term ordering to be lexicographic.

INPUT:

- `B` - a list/tuple of polynomials or a multivariate polynomial ideal
- `n` - the recursion parameter (default: `-1`)

OUTPUT:

A list `T` of triangular sets `T_0`, `T_1`, etc.

EXAMPLE:

```
sage: set_verbose(0)
sage: from sage.rings.polynomial.toy_variety import triangular_factorization
sage: R.<x,y,z> = PolynomialRing(GF(32003))
sage: p1 = x^2*(x-1)^3*y^2*(z-3)^3
sage: p2 = z^2 - z
sage: p3 = (x-2)^2*(y-1)^3
sage: I = R.ideal(p1,p2,p3)
sage: triangular_factorization(I.groebner_basis(), proof=True)
...
NotImplementedError: proof = True factorization not implemented.  Call factor with proof=False.

sage: triangular_factorization(I.groebner_basis(), proof=False)
[[x^2 - 4*x + 4, y, z],
 [x^5 - 3*x^4 + 3*x^3 - x^2, y - 1, z],
 [x^2 - 4*x + 4, y, z - 1],
 [x^5 - 3*x^4 + 3*x^3 - x^2, y - 1, z - 1]]
```

28.14 Boolean Polynomials.

Elements of the quotient ring

$$\mathbf{F}_2[x_1, ..., x_n]/ < x_1^2 + x_1, ..., x_n^2 + x_n > .$$

are called boolean polynomials. Boolean polynomials arise naturally in cryptography, coding theory, formal logic, chip design and other areas. This implementation is a thin wrapper around the PolyBoRi library by Michael Brickenstein and Alexander Dreyer.

"Boolean polynomials can be modelled in a rather simple way, with both coefficients and degree per variable lying in `{0, 1}`. The ring of Boolean polynomials is, however, not a polynomial ring, but rather the quotient ring of the polynomial ring over the field with two elements modulo the field equations $x^2 = x$ for each variable x. Therefore, the usual polynomial data structures seem not to be appropriate for fast Groebner basis computations. We introduce a specialised data structure for Boolean polynomials based on zero-suppressed binary decision diagrams (ZDDs), which is capable of handling these polynomials more efficiently with respect to memory consumption and also computational speed. Furthermore, we concentrate on high-level algorithmic aspects, taking into account the new data structures as well as structural properties of Boolean polynomials." - [BD07]

AUTHORS:

- Michael Brickenstein: PolyBoRi author
- Alexander Dreyer: PolyBoRi author
- Burcin Erocal <burcin@erocal.org>: main Sage wrapper author
- Martin Albrecht <malb@informatik.uni-bremen.de>: some contributions to the Sage wrapper

EXAMPLES:

Consider the ideal

$$< ab + cd + 1, ace + de, abe + ce, bc + cde + 1 > .$$

First, we compute the lexicographical Groebner basis in the polynomial ring

$$R = \mathbf{F}_2[a, b, c, d, e].$$

```
sage: P.<a,b,c,d,e> = PolynomialRing(GF(2), 5, order='lex')
sage: I1 = ideal([a*b + c*d + 1, a*c*e + d*e, a*b*e + c*e, b*c + c*d*e + 1])
sage: for f in I1.groebner_basis():
...       f
a + c^2*d + c + d^2*e
b*c + d^3*e^2 + d^3*e + d^2*e^2 + d*e + e + 1
b*e + d*e^2 + d*e + e
c*e + d^3*e^2 + d^3*e + d^2*e^2 + d*e
d^4*e^2 + d^4*e + d^3*e + d^2*e^2 + d^2*e + d*e + e
```

If one wants to solve this system over the algebraic closure of $\mathbf{F}_2$ then this Groebner basis was the one to consider. If one wants solutions over $\mathbf{F}_2$ only then one adds the field polynomials to the ideal to force the solutions in $\mathbf{F}_2$.

```
sage: J = I1 + sage.rings.ideal.FieldIdeal(P)
sage: for f in J.groebner_basis():
...       f
a + d + 1
b + 1
c + 1
d^2 + d
e
```

So the solutions over $\mathbf{F}_2$ are $\{e = 0, d = 1, c = 1, b = 1, a = 0\}$ and $\{e = 0, d = 0, c = 1, b = 1, a = 1\}$.

We can express the restriction to $\mathbf{F}_2$ by considering the quotient ring. If I is an ideal in $\mathbb{F}[x_1, ..., x_n]$ then the ideals in the quotient ring $\mathbb{F}[x_1, ..., x_n]/I$ are in one-to-one correspondence with the ideals of $\mathbb{F}[x_0, ..., x_n]$ containing I (that is, the ideals J satisfying $I \subset J \subset P$).

```
sage: Q = P.quotient( sage.rings.ideal.FieldIdeal(P) )
sage: I2 = ideal([Q(f) for f in I1.gens()])
sage: for f in I2.groebner_basis():
...       f
abar + dbar + 1
bbar + 1
cbar + 1
ebar
```

This quotient ring is exactly what PolyBoRi handles well:

```
sage: B.<a,b,c,d,e> = BooleanPolynomialRing(5, order='lex')
sage: I2 = ideal([B(f) for f in I1.gens()])
sage: for f in I2.groebner_basis():
...       f
a + d + 1
b + 1
c + 1
e
```

Note that $d^2 + d$ is not representable in $B == Q$. Also note, that PolyBoRi cannot play out its strength in such small examples, i.e. working in the polynomial ring might be faster for small examples like this.

28.14.1 Implementation specific notes

PolyBoRi comes with a Python wrapper. However this wrapper does not match Sage's style and is written using Boost. Thus Sage's wrapper is a reimplementation of Python bindings to PolyBoRi's C++ library. This interface is written in

Cython like all of Sage's C/C++ library interfaces. An interface in PolyBoRi style is also provided which is effectively a reimplementation of the official Boost wrapper in Cython. This means that some functionality of the official wrapper might be missing from this wrapper and this wrapper might have bugs not present in the official Python interface.

28.14.2 Access to the original PolyBoRi interface

The re-implementation PolyBoRi's native wrapper is available to the user too:

```
sage: from polybori import *
sage: declare_ring([Block('x',2),Block('y',3)],globals())
Boolean PolynomialRing in x(0), x(1), y(0), y(1), y(2)
sage: r
Boolean PolynomialRing in x(0), x(1), y(0), y(1), y(2)


sage: [Variable(i) for i in xrange(r.ngens())]
[x(0), x(1), y(0), y(1), y(2)]
```

For details on this interface see:

http://polybori.sourceforge.net/doc/tutorial/tutorial.html.

Note that due to this duality of this interface some functions do not obey Sage's naming convention. For instance, `f.zerosIn` should be called `f.zeros_in` by Sage's standards. However, PolyBoRi expects the function `f.zerosIn`.

Also, the interface provides functions for compatibility with Sage accepting convenient Sage data types which are slower than their native PolyBoRi counterparts. For instance, sets of points can be represented as tuples of tuples (Sage) or as `BooleSets` (PolyBoRi) and naturally the second option is faster.

REFERENCES:

class BooleSet()

Return a new set of boolean monomials. This data type is also implemented on the top of ZDDs and allows to see polynomials from a different angle. Also, it makes high-level set operations possible, which are in most cases faster than operations handling individual terms, because the complexity of the algorithms depends only on the structure of the diagrams.

`BooleanPolynomials` can easily be converted to `BooleSets` by using the member function `set()`.

INPUT:

- `param` - either a `CCuddNavigator`, a `BooleSet` or `None`.
- `ring` - a boolean polynomial ring.

EXAMPLE:

```
sage: from polybori import BooleSet
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: BS = BooleSet(B)
sage: BS
{}
```

Note: `BooleSet` prints as '{}' but are not Python dictionaries.

cartesianProduct()

Return the cartesian product of this set and the set `rhs`.

The Cartesian product of two sets X and Y is the set of all possible ordered pairs whose first component is a member of X and whose second component is a member of Y.

$$X \times Y = \{(x, y)|x \in X \text{ and } y \in Y\}.$$

EXAMPLE:

```
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: g = x4 + 1
sage: t = g.set(); t
{{x4}, {}}
sage: s.cartesianProduct(t)
{{x1,x2,x4}, {x1,x2}, {x2,x3,x4}, {x2,x3}}
```

change()

diff()

Return the set theoretic difference of this set and the set `rhs`.

The difference of two sets X and Y is defined as:

$$X \ Y = \{x|x \in X \text{ and } x \notin Y\}.$$

EXAMPLE:

```
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: g = x2*x3 + 1
sage: t = g.set(); t
{{x2,x3}, {}}
sage: s.diff(t)
{{x1,x2}}
```

divide()

empty()

Return `True` if this set is empty.

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: BS = (a*b + c).set()
sage: BS.empty()
False

sage: BS = B(0).set()
sage: BS.empty()
True
```

includeDivisors()

minimalElements()

nNodes()

Return the number of nodes in the ZDD.

EXAMPLE:

```
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: s.nNodes()
4
```

nSupport()

navigation()

Navigators provide an interface to diagram nodes, accessing their index as well as the corresponding then- and else-branches.

You should be very careful and always keep a reference to the original object, when dealing with navigators, as navigators contain only a raw pointer as data. For the same reason, it is necessary to supply the ring as argument, when constructing a set out of a navigator.

EXAMPLE:

```
sage: from polybori import BooleSet
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3*x4+x2*x4+x3+x4+1
sage: s = f.set(); s
{{x1,x2}, {x2,x3,x4}, {x2,x4}, {x3}, {x4}, {}}

sage: nav = s.navigation()
sage: BooleSet(nav,s.ring())
{{x1,x2}, {x2,x3,x4}, {x2,x4}, {x3}, {x4}, {}}

sage: nav.value()
1

sage: nav_else = nav.elseBranch()

sage: BooleSet(nav_else,s.ring())
{{x2,x3,x4}, {x2,x4}, {x3}, {x4}, {}}

sage: nav_else.value()
2
```

ring()

Return the parent ring.

EXAMPLE:

```
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3*x4+x2*x4+x3+x4+1
sage: f.set().ring() is B
True
```

set()

Return `self`.

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: BS = (a*b + c).set()
sage: BS.set() is BS
True
```

stableHash()

subset0()

subset1()

union()

Return the set theoretic union of this set and the set `rhs`.

The union of two sets X and Y is defined as:

$$X \cup Y = \{x | x \in X \text{ or } x \in Y\}.$$

EXAMPLE:

```
sage: B = BooleanPolynomialRing(5,'x')
sage: x0,x1,x2,x3,x4 = B.gens()
sage: f = x1*x2+x2*x3
sage: s = f.set(); s
{{x1,x2}, {x2,x3}}
sage: g = x2*x3 + 1
sage: t = g.set(); t
{{x2,x3}, {}}
sage: s.diff(t)
{{x1,x2}}
```

vars()

class **BooleSetIterator**()

next()

x.next() -> the next value, or raise StopIteration

class **BooleVariable**()

class **BooleanMonomial**()

Construct a boolean monomial object.

INPUT:

- `parent` - parent monoid this element lies in

EXAMPLE:

```
sage: from polybori import BooleanMonomialMonoid, BooleanMonomial
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: BooleanMonomial(M)
1
```

Note: Use the `__call__` method of `BooleanMonomialMonoid` and not this constructor to construct these objects.

deg()

Return degree of this monomial.

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M(x*y).deg()
2
```

```
sage: M(x*x*y*z).deg()
3
```

degree()

Return degree of this monomial.

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M(x*y).degree()
2
```

divisors()

Return a set of boolean monomials with all divisors of this monomial.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.divisors()
{{x,y}, {x}, {y}, {}}
```

index()

Return the variable index of the first variable in this monomial.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.index()
0
```

iterindex()

Return an iterator over the indicies of the variables in self.

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: list(M(x*z).iterindex())
[0, 2]
```

multiples()

Return a set of boolean monomials with all multiples of this monomial up the bound `rhs`.

INPUT:

- `rhs` - a boolean monomial

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x
sage: m = f.lm()
sage: g = x*y*z
sage: n = g.lm()
sage: m.multiples(n)
{{x,y,z}, {x,y}, {x,z}, {x}}
sage: n.multiples(m)
{{x,y,z}}
```

Note: The returned set always contains `self` even if the bound `rhs` is smaller than `self`.

reducibleBy()

Return `True` if `self` is reducible by `rhs`.

INPUT:

- `rhs` - a boolean monomial

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.reducibleBy((x*y).lm())
True
sage: m.reducibleBy((x*z).lm())
False
```

set()

Return a boolean set of variables in this monomials.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y
sage: m = f.lm()
sage: m.set()
{{x,y}}
```

stableHash()

variables()

Return a list of the variables in this monomial.

EXAMPLE:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M(x*z).variables() # indirect doctest
[x, z]
```

class **BooleanMonomialIterator**()

An iterator over the variable indices of a monomial.

next()

x.next() -> the next value, or raise StopIteration

class **BooleanMonomialMonoid**()

Construct a boolean monomial monoid given a boolean polynomial ring.

This object provides a parent for boolean monomials.

INPUT:

- `polring` - the polynomial ring our monomials lie in

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: M = BooleanMonomialMonoid(P)
sage: M
MonomialMonoid of Boolean PolynomialRing in x, y
```

```
sage: M.gens()
(x, y)
sage: type(M.gen(0))
<type 'sage.rings.polynomial.pbori.BooleanMonomial'>
```

gen()
Return the i-th generator of self.
INPUT:
- i - an integer

EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M.gen(0)
x
sage: M.gen(2)
z

sage: P = BooleanPolynomialRing(1000, 'x')
sage: M = BooleanMonomialMonoid(P)
sage: M.gen(50)
x50
```

gens()
Return the tuple of generators of this monoid.
EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: M = BooleanMonomialMonoid(P)
sage: M.gens()
(x, y, z)
```

ngens()
Returns the number of variables in this monoid.
EXAMPLES:

```
sage: from polybori import BooleanMonomialMonoid
sage: P = BooleanPolynomialRing(100, 'x')
sage: M = BooleanMonomialMonoid(P)
sage: M.ngens()
100
```

class **BooleanMonomialVariableIterator**()

next()
x.next() -> the next value, or raise StopIteration

class **BooleanMulAction**()

class **BooleanPolynomial**()
Construct a boolean polynomial object in the given boolean polynomial ring.
INPUT:
- parent - a boolean polynomial ring

TEST:

```
sage: from polybori import BooleanPolynomial
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: BooleanPolynomial(B)
0
```

Note: Do not use this method to construct boolean polynomials, but use the appropriate __call__ method in the parent.

constant()

Return True if this element is constant.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: x.constant()
False

sage: B(1).constant()
True
```

constant_coefficient()

Returns the constant coefficient of this boolean polynomial.

EXAMPLE: sage: B.<a,b> = BooleanPolynomialRing() sage: a.constant_coefficient() 0 sage: (a+1).constant_coefficient() 1

deg()

Return the degree of self. This is usually equivalent to the total degree except for weighted term orderings which are not implemented yet.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: (x+y).degree()
1

sage: P(1).degree()
0

sage: (x*y + x + y + 1).degree()
2
```

degree()

Return the total degree of self.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: (x+y).degree()
1

sage: P(1).degree()
0

sage: (x*y + x + y + 1).degree()
2
```

elength()

Return elimination length as used in the SlimGB algorithm.

EXAMPLE:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: x.elength()
```

```
1
sage: f = x*y + 1
sage: f.elength()
2
```

REFERENCES:

- Michael Brickenstein; SlimGB: Groebner Bases with Slim Polynomials http://www.mathematik.uni-kl.de/~zca/Reports_on_ca/35/paper_35_full.ps.gz

firstTerm()

Return the first term with respect to the lexicographical term ordering.

EXAMPLE:

```
sage: B.<a,b,z> = BooleanPolynomialRing(3,order='lex')
sage: f = b*z + a + 1
sage: f.firstTerm()
a
```

gradedPart()

Return graded part of this boolean polynomial of degree `deg`.

INPUT:

- `deg` - a degree

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + c*d + a*b + 1
sage: f.gradedPart(2)
a*b + c*d

sage: f.gradedPart(0)
1
```

TESTS:

```
sage: f.gradedPart(-1)
0
```

hasConstantPart()

Return `True` if this boolean polynomial has a constant part, i.e. if 1 is a term.

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + c*d + a*b + 1
sage: f.hasConstantPart()
True

sage: f = a*b*c + c*d + a*b
sage: f.hasConstantPart()
False
```

isOne()

Return `True` if this element is one.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: x.isOne()
False
sage: B(0).isOne()
False
```

```
sage: B(1).isOne()
True
```

isZero()

Return `True` if this element is zero.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: x.isZero()
False
sage: B(0).isZero()
True
sage: B(1).isZero()
False
```

is_constant()

Check if `self` is constant.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(1).is_constant()
True

sage: P(0).is_constant()
True

sage: x.is_constant()
False

sage: (x*y).is_constant()
False
```

is_equal()

EXAMPLE:

```
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*z + b + 1
sage: g = b + z
sage: f.is_equal(g)
False

sage: f.is_equal( (f + 1) - 1 )
True
```

is_homogeneous()

Return `True` if this element is a homogeneous polynomial.

EXAMPLES:

```
sage: P.<x, y> = BooleanPolynomialRing()
sage: (x+y).is_homogeneous()
True
sage: P(0).is_homogeneous()
True
sage: (x+1).is_homogeneous()
False
```

is_one()

Check if self is 1.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(1).is_one()
True

sage: P.one_element().is_one()
True

sage: x.is_one()
False

sage: P(0).is_one()
False
```

is_unit()

Check if `self` is invertible in the parent ring.

Note that this condition is equivalent to being 1 for boolean polynomials.

EXAMPLE:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P.one_element().is_unit()
True

sage: x.is_unit()
False
```

is_zero()

Check if `self` is zero.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P(0).is_zero()
True

sage: x.is_zero()
False

sage: P(1).is_zero()
False
```

lead()

Return the leading monomial of boolean polynomial, with respect to to the order of parent ring.

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lead()
x

sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lead()
y*z
```

lexLead()

Return the leading monomial of boolean polynomial, with respect to the lexicographical term ordering.

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lexLead()
x
```

```
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lexLead()
x
```

lexLmDeg()

Return degree of leading monomial with respect to the lexicographical ordering.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3,order='lex')
sage: f = x + y*z
sage: f
x + y*z
sage: f.lexLmDeg()
1

sage: B.<x,y,z> = BooleanPolynomialRing(3,order='deglex')
sage: f = x + y*z
sage: f
y*z + x
sage: f.lexLmDeg()
1
```

lm()

Return the leading monomial of this boolean polynomial, with respect to the order of parent ring.

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lm()
x

sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lm()
y*z
```

lmDeg()

Return the degree of the leading monomial with respect to the lexicographical orderings.

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3,order='lex')
sage: f = x + y*z
sage: f
x + y*z
sage: f.lmDeg()
1

sage: B.<x,y,z> = BooleanPolynomialRing(3,order='deglex')
sage: f = x + y*z
sage: f
y*z + x
sage: f.lmDeg()
2
```

lmDivisors()

Return a `BooleSet` of all divisors of the leading monomial.

EXAMPLE:

```
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*b + z + 1
```

```
sage: f.lmDivisors()
{{a,b}, {a}, {b}, {}}
```

lm_degree()

Returns the total degree of the leading monomial of `self`.

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: p = x + y*z
sage: p.lm_degree()
1

sage: P.<x,y,z> = BooleanPolynomialRing(3,order='deglex')
sage: p = x + y*z
sage: p.lm_degree()
2

sage: P(0).lm_degree()
0
```

lt()

Return the leading term of this boolean polynomial, with respect to the order of the parent ring.

Note that for boolean polynomials this is equivalent to returning leading monomials.

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x+y+y*z).lt()
x

sage: P.<x,y,z> = BooleanPolynomialRing(3, order='deglex')
sage: (x+y+y*z).lt()
y*z
```

mapEveryXToXPlusOne()

Map every variable `x_i` in this polynomial to $x_i + 1$.

EXAMPLE:

```
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: f = a*b + z + 1; f
a*b + z + 1
sage: f.mapEveryXToXPlusOne()
a*b + a + b + z + 1
sage: f(a+1,b+1,z+1)
a*b + a + b + z + 1
```

monomial_coefficient()

Return the coefficient of the monomial `mon` in `self`, where `mon` must have the same parent as `self`.

INPUT:

- `mon` - a monomial

EXAMPLE:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: x.monomial_coefficient(x)
1
sage: x.monomial_coefficient(y)
0
sage: R.<x,y,z,a,b,c>=BooleanPolynomialRing(6)
```

```
sage: f=(1-x)*(1+y); f
x*y + x + y + 1

sage: f.monomial_coefficient(1)
1

sage: f.monomial_coefficient(0)
0
```

monomials()

Return a list of monomials appearing in `self` ordered largest to smallest.

EXAMPLE:

```
sage: P.<a,b,c> = BooleanPolynomialRing(3,order='lex')
sage: f = a + c*b
sage: f.monomials()
[a, b*c]

sage: P.<a,b,c> = BooleanPolynomialRing(3,order='degrevlex')
sage: f = a + c*b
sage: f.monomials()
[c*b, a]
```

nNodes()

nVars()

Return the number of variables used to form this boolean polynomial.

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + 1
sage: f.nVars()
3
```

navigation()

nvariables()

Return the number of variables used to form this boolean polynomial.

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + 1
sage: f.nvariables()
3
```

reduce()

Return the normal form of `self` w.r.t. `I`, i.e. return the remainder of `self` with respect to the polynomials in `I`. If the polynomial set/list `I` is not a Groebner basis the result is not canonical.

INPUT:

- `I` - a list/set of polynomials in self.parent(). If I is an ideal, the generators are used.

EXAMPLE:

```
sage: B.<x0,x1,x2,x3> = BooleanPolynomialRing(4)
sage: I = B.ideal((x0 + x1 + x2 + x3, \
                   x0*x1 + x1*x2 + x0*x3 + x2*x3, \
                   x0*x1*x2 + x0*x1*x3 + x0*x2*x3 + x1*x2*x3, \
                   x0*x1*x2*x3 + 1))
sage: gb = I.groebner_basis()
sage: f,g,h,i = I.gens()
```

```
sage: f.reduce(gb)
0
sage: p = f*g + x0*h + x2*i
sage: p.reduce(gb)
0
sage: p.reduce(I)
x1*x2*x3 + x2
```

Note: If this function is called repeatedly with the same I then it is advised to use PolyBoRi's `GroebnerStrategy` object directly, since that will be faster. See the source code of this function for details.

reducibleBy()

Return `True` if this boolean polynomial is reducible by the polynomial `rhs`.

INPUT:

- `rhs` - a boolean polynomial

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4,order='degrevlex')
sage: f = (a*b + 1)*(c + 1)
sage: f.reducibleBy(d)
False
sage: f.reducibleBy(c)
True
sage: f.reducibleBy(c + 1)
True
```

ring()

Return the parent of this boolean polynomial.

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: a.ring() is B
True
```

set()

Return a `BooleSet` with all monomials appearing in this polynomial.

EXAMPLE:

```
sage: B.<a,b,z> = BooleanPolynomialRing(3)
sage: (a*b+z+1).set()
{{a,b}, {z}, {}}
```

spoly()

Return the S-Polynomial of this boolean polynomial and the other boolean polynomial `rhs`.

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + c*d + a*b + 1
sage: g = c*d + b
sage: f.spoly(g)
a*b + a*c*d + c*d + 1
```

stableHash()

subs()

Fixes some given variables in a given boolean polynomial and returns the changed boolean polynomials. The polynomial itself is not affected. The variable,value pairs for fixing are to be provided as dictionary of the form {variable:value} or named parameters (see examples below).

INPUT:

- `in_dict` - (optional) dict with variable:value pairs
- `**kwds` - names parameters

EXAMPLE:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: f = x*y + z + y*z + 1
sage: f.subs(x=1)
y*z + y + z + 1
sage: f.subs(x=0)
y*z + z + 1

sage: f.subs(x=y)
y*z + y + z + 1

sage: f.subs({x:1},y=1)
0
sage: f.subs(y=1)
x + 1
sage: f.subs(y=1,z=1)
x + 1
sage: f.subs(z=1)
x*y + y
sage: f.subs({'x':1},y=1)
0
```

This method can work fully symbolic:

```
sage: f.subs(x=var('a'),y=var('b'),z=var('c'))
a*b + b*c + c + 1
sage: f.subs({'x':var('a'),'y':var('b'),'z':var('c')})
a*b + b*c + c + 1
```

totalDegree()

Return total degree of this boolean polynomial.

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b*c + 1
sage: f.totalDegree()
3
```

total_degree()

Return the total degree of `self`.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: (x+y).total_degree()
1

sage: P(1).total_degree()
0

sage: (x*y + x + y + 1).total_degree()
2
```

variables()

Return a list of all variables appearing in `self`.

EXAMPLE:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x + y).variables()
[x, y]

sage: (x*y + z).variables()
[x, y, z]

sage: P.zero_element().variables()
[]

sage: P.one_element().variables()
[1]
```

varsAsMonomial()

Return a boolean monomial with all the variables appearing in `self`.

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: (x + y).varsAsMonomial()
x*y

sage: (x*y + z).varsAsMonomial()
x*y*z

sage: P.zero_element().varsAsMonomial()
1

sage: P.one_element().varsAsMonomial()
1
```

TESTS:

```
sage: R = BooleanPolynomialRing(1, 'y')
sage: y.varsAsMonomial()
y
sage: R
Boolean PolynomialRing in y
```

zerosIn()

Return a set containing all elements of `s` where this boolean polynomial evaluates to zero.

If `s` is given as a `BooleSet`, then the return type is also a `BooleSet`. If `s` is a set/list/tuple of tuple this function returns a tuple of tuples.

INPUT:

- `s` - candidate points for evaluation to zero

EXAMPLE:

```
sage: B.<a,b,c,d> = BooleanPolynomialRing(4)
sage: f = a*b + c + d + 1
```

Now we create a set of points:

```
sage: s = a*b + a*b*c + c*d + 1
sage: s = s.set(); s
{{a,b,c}, {a,b}, {c,d}, {}}
```

This encodes the points (1,1,1,0), (1,1,0,0), (0,0,1,1) and (0,0,0,0). But of these only (1,1,0,0) evaluates to zero.

```
sage: f.zerosIn(s)
{{a,b}}

sage: f.zerosIn([(1,1,1,0), (1,1,0,0), (0,0,1,1), (0,0,0,0)])
((1, 1, 0, 0),)
```

class BooleanPolynomialIdeal()

groebner_basis()

Return a Groebner basis of this ideal.

INPUT:

- **red_tail - tail reductions in intermediate polynomials,** this options affects mainly heuristics. The reducedness of the output polynomials can only be guaranteed by the option redsb (default: `True`)
- `minsb` - return a minimal Groebner basis (default: `True`)
- `redsb` - return a minimal Groebner basis and all tails are reduced (default: `True`)
- `deg_bound` - only compute Groebner basis up to a given degree bound (default: `False`)
- `faugere` - turn off or on the linear algebra (default: `False`)
- `linear_algebra_in_last_block` - this affects the last block of block orderings and degree orderings. If it is set to `True` linear algebra takes affect in this block. (default: `True`)
- `selection_size` - maximum number of polynomials for parallel reductions (default: `1000`)
- `heuristic` - Turn off heuristic by setting `heuristic=False` (default: `True`)
- `lazy` - (default: `True`)
- `invert` - setting `invert=True` input and output get a transformation `x+1` for each variable `x`, which shouldn't effect the calculated GB, but the algorithm.
- `other_ordering_first` - possible values are `False` or an ordering code. In practice, many Boolean examples have very few solutions and a very easy Groebner basis. So, a complex walk algorithm (which cannot be implemented using the data structures) seems unnecessary, as such Groebner bases can be converted quite fast by the normal Buchberger algorithm from one ordering into another ordering. (default: `False`)
- `prot` - show protocol (default: `False`)
- `full_prot` - show full protocol (default: `False`)

EXAMPLES:

```
sage: P.<x0, x1, x2, x3> = BooleanPolynomialRing(4)
sage: I = P.ideal(x0*x1*x2*x3 + x0*x1*x3 + x0*x1 + x0*x2 + x0)
sage: I.groebner_basis()
[x0*x1 + x0*x2 + x0, x0*x2*x3 + x0*x3]
```

interreduced_basis()

If this ideal is spanned by `(f_1, ..., f_n)` this method returns `(g_1, ..., g_s)` such that:

- `(f_1,...,f_n) = (g_1,...,g_s)`
- `LT(g_i) != LT(g_j)` for all `i != j`'
- `LT(g_i)` does not divide `m` for all monomials `m` of `{g_1,...,g_{i-1},g_{i+1},...,g_s}`

EXAMPLE:

```
sage: sr = mq.SR(1, 1, 1, 4, gf2=True, polybori=True)
sage: F,s = sr.polynomial_system()
sage: I = F.ideal()
sage: I.interreduced_basis()
[k000*s000 + k002*s000 + k002*s001 + k002*s003 + k003*s001 + k003*s002 + k003*s003 + s001 +
 k000*s001 + k001*s003 + k002*s000 + k002*s001 + k002*s003 + k003*s000 + s001,
```

```
k000*s002 + k001*s000 + k001*s001 + k001*s003 + k002*s001 + k003*s000 + k003*s002 + s001,
k000*s003 + k001*s001 + k001*s002 + k001*s003 + k002*s001 + k003*s000 + k003*s001 + s001 +
k001*s000 + k001*s003 + k002*s000 + k002*s001 + k003*s000 + k003*s002 + k003*s003 + s001 +
k001*s001 + k001*s003 + k002*s000 + k002*s001 + k002*s003 + k003*s002 + k003*s003 + k003,
k001*s002 + k002*s000 + k002*s003 + k003*s001 + k003*s002 + s003 + 1,
k001*s003 + k002*s000 + k002*s001 + k002*s002 + s001 + k001 + k003 + 1,
k001*k000 + k002*k000 + k003*s002 + k003,
k002*s000 + k002*s002 + k003*s001 + k003*s002 + s001 + k000 + k001 + k002,
k002*s001 + k002*s003 + k003*s000 + k003*s001 + s003 + k000 + k001 + k003,
k002*s002 + k003*s003 + s000 + s001 + k000 + k001 + 1,
k002*s003 + k003*s000 + k003*s001 + k003*s002 + k003*s003 + s002 + s003 + k001,
k002*k000 + k000 + k002 + 1,
k002*k001 + k003*s002 + k003*k002 + k002 + k003,
k003*s001 + k003*s002 + k003*s003 + k001,
k003*k000 + s001 + 1,
k003*k001 + s001 + k000 + k003,
k003*k002 + s001 + k002 + 1,
k100 + x103 + s001 + s002 + 1,
k101 + x102 + x103 + s002,
k102 + x100 + x103 + s001 + s002 + s003,
k103 + x101 + x102 + x103,
x100 + x103 + s001 + s003 + 1,
x101 + x102 + x103 + s001 + s002 + s003 + 1,
x102 + x103 + s002 + s003 + 1,
x103 + s003 + 1,
w100 + k000 + 1,
w101,
w102 + k002 + 1,
w103 + k003 + 1,
s000 + s001,
s001 + k002 + k003 + 1,
s002 + k002 + k003 + 1,
s003 + k002 + 1,
k001]
```

reduce()

Reduce an element modulo the reduced Groebner basis for this ideal. This returns 0 if and only if the element is in this ideal. In any case, this reduction is unique up to monomial orders.

EXAMPLE:

```
sage: P = PolynomialRing(GF(2),10, 'x')
sage: B = BooleanPolynomialRing(10,'x')
sage: I = sage.rings.ideal.Cyclic(P)
sage: I = B.ideal([B(f) for f in I.gens()])
sage: gb = I.groebner_basis()
sage: I.reduce(gb[0])
0
sage: I.reduce(gb[0] + 1)
1
sage: I.reduce(gb[0]*gb[1])
0
sage: I.reduce(gb[0]*B.gen(1))
0
```

class **BooleanPolynomialIterator**()

Iterator over the monomials of a boolean polynomial.

next()

x.next() -> the next value, or raise StopIteration

class BooleanPolynomialRing()

Construct a boolean polynomial ring with the following parameters:

INPUT:

- n - number of variables (an integer > 1)
- **names - names of ring variables, may be a string or** list/tuple
- order - term order (default: lex)

EXAMPLES:

```
sage: R.<x, y, z> = BooleanPolynomialRing()
sage: R
Boolean PolynomialRing in x, y, z

sage: p = x*y + x*z + y*z
sage: x*p
x*y*z + x*y + x*z

sage: R.term_order()
Lexicographic term order

sage: R = BooleanPolynomialRing(5,'x',order='deglex(3),deglex(2)')
sage: R.term_order()
deglex(3),deglex(2) term order

sage: R = BooleanPolynomialRing(3,'x',order='degrevlex')
sage: R.term_order()
Degree reverse lexicographic term order
```

TESTS:

```
sage: P.<x,y> = BooleanPolynomialRing(2,order='degrevlex')
sage: x > y
True

sage: P.<x0, x1, x2, x3> = BooleanPolynomialRing(4,order='degrevlex(2),degrevlex(2)')
sage: x0 > x1
True
sage: x2 > x3
True
```

cover_ring()

Return $R = \mathbf{F}_2[vars]$ if $vars$ is the ordered list of variable names of this ring. R also has the same term ordering as this ring.

EXAMPLE:

```
sage: B.<x,y> = BooleanPolynomialRing(2)
sage: R = B.cover_ring(); R
Multivariate Polynomial Ring in x, y over Finite Field of size 2

sage: B.term_order() == R.term_order()
True
```

The cover ring is cached:

```
sage: B.cover_ring() is B.cover_ring()
True
```

defining_ideal()

Return $I =< x_i^2 + x_i >\subset R$ where $R = \mathbf{F}_2[vars]$, $vars$ the ordered list of variables/variable names of this ring and x_i any element in $vars$.

EXAMPLE:

```
sage: B.<x,y> = BooleanPolynomialRing(2)
sage: I = B.defining_ideal(); I
Ideal (x^2 + x, y^2 + y) of Multivariate Polynomial Ring
in x, y over Finite Field of size 2
```

gen()

Returns the i-th generator of this boolean polynomial ring.

INPUT:

- `i` - an integer or a boolean monomial in one variable

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.gen()
x
sage: P.gen(2)
z
sage: m = x.monomials()[0]
sage: P.gen(m)
x
```

TESTS:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3, order='dp')
sage: P.gen(0)
x
```

gens()

Return the tuple of variables in this ring.

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.gens()
(x, y, z)

sage: P = BooleanPolynomialRing(10,'x')
sage: P.gens()
(x0, x1, x2, x3, x4, x5, x6, x7, x8, x9)
```

TESTS:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3,order='degrevlex')
sage: P.gens()
(x, y, z)
```

ideal()

Create an ideal in this ring.

INPUT:

- `gens` - list or tuple of generators
- `coerce` - bool (default: True) automatically coerce the given polynomials to this ring to form the ideal

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.ideal(x+y)
Ideal (x + y) of Boolean PolynomialRing in x, y, z

sage: P.ideal(x*y, y*z)
Ideal (x*y, y*z) of Boolean PolynomialRing in x, y, z

sage: P.ideal([x+y, z])
Ideal (x + y, z) of Boolean PolynomialRing in x, y, z
```

interpolation_polynomial()

Return the lexicographically minimal boolean polynomial for the given sets of points.

Given two sets of points `zeros` - evaluating to zero - and `ones` - evaluating to one -, compute the lexicographically minimal boolean polynomial satisfying these points.

INPUT:

- `zeros` - the set of interpolation points mapped to zero
- `ones` - the set of interpolation points mapped to one

EXAMPLE:

First we create a random-ish boolean polynomial.

```
sage: B.<a,b,c,d,e,f> = BooleanPolynomialRing(6)
sage: f = a*b*c*e + a*d*e + a*f + b + c + e + f + 1
```

Now we find interpolation points mapping to zero and to one.

```
sage: zeros = set([(1, 0, 1, 0, 0, 0), (1, 0, 0, 0, 1, 0), \
                   (0, 0, 1, 1, 1, 1), (1, 0, 1, 1, 1, 1), \
                   (0, 0, 0, 0, 1, 0), (0, 1, 1, 1, 1, 0), \
                   (1, 1, 0, 0, 0, 1), (1, 1, 0, 1, 0, 1)])
sage: ones = set([(0, 0, 0, 0, 0, 0), (1, 0, 1, 0, 1, 0), \
                  (0, 0, 0, 1, 1, 1), (1, 0, 0, 1, 0, 1), \
                  (0, 0, 0, 0, 1, 1), (0, 1, 1, 0, 1, 1), \
                  (0, 1, 1, 1, 1, 1), (1, 1, 1, 0, 1, 0)])
sage: [f(*p) for p in zeros]
[0, 0, 0, 0, 0, 0, 0, 0]
sage: [f(*p) for p in ones]
[1, 1, 1, 1, 1, 1, 1, 1]
```

Finally, we find the lexicographically smallest interpolation polynomial using PolyBoRi .

```
sage: g = B.interpolation_polynomial(zeros, ones); g
b*f + c + d*f + d + e*f + e + 1

sage: [g(*p) for p in zeros]
[0, 0, 0, 0, 0, 0, 0, 0]
sage: [g(*p) for p in ones]
[1, 1, 1, 1, 1, 1, 1, 1]
```

Alternatively, we can work with PolyBoRi's native `BooleSet`'s. This example is from the PolyBoRi tutorial:

```
sage: B = BooleanPolynomialRing(4,"x0,x1,x2,x3")
sage: x = B.gen
sage: V=(x(0)+x(1)+x(2)+x(3)+1).set(); V
{{x0}, {x1}, {x2}, {x3}, {}}
sage: f=x(0)*x(1)+x(1)+x(2)+1
sage: z = f.zerosIn(V); z
```

```
{{x1}, {x2}}
sage: o = V.diff(z); o
{{x0}, {x3}, {}}
sage: B.interpolation_polynomial(z,o)
x1 + x2 + 1
```

ALGORITHM: Calls `interpolate_smallest_lex` as described in the PolyBoRi tutorial.

ngens()

Returns the number of variables in this boolean polynomial ring.

EXAMPLES:

```
sage: P.<x,y> = BooleanPolynomialRing(2)
sage: P.ngens()
2

sage: P = BooleanPolynomialRing(1000, 'x')
sage: P.ngens()
1000
```

random_element()

Return a random boolean polynomial. Generated polynomial has the given number of terms, and at most given degree.

INPUT:

- `degree` - maximum degree (default: 2)
- `terms` - number of terms (default: 5)
- `choose_degree` - choose degree of monomials randomly first, rather than monomials uniformly random
- `vars_set` - list of integer indicies of generators of self to use in the generated polynomial

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: P.random_element(degree=3, terms=4)
x*y*z + x*z + y*z + z

sage: P.random_element(degree=1, terms=2)
z + 1
```

TESTS:

```
sage: P.random_element(degree=4)
...
ValueError: Given degree should be less than or equal to number of variables (3)

sage: t = P.random_element(degree=1, terms=5)
...
ValueError: Cannot generate random polynomial with 5 terms and maximum degree 1 using 3 vari

sage: t = P.random_element(degree=2,terms=5,vars_set=(0,1))
...
ValueError: Cannot generate random polynomial with 5 terms using 2 variables
```

class **BooleanPolynomialVector**()

append()

class **BooleanPolynomialVectorIterator**()

next()
x.next() -> the next value, or raise StopIteration

class **CCuddNavigator**()

constant()
elseBranch()
terminalOne()
thenBranch()
value()

class **DD**()

empty()
navigation()
subset0()
subset1()
union()

class **GroebnerStrategy**()

addAsYouWish()
addGenerator()
addGeneratorDelayed()
allGenerators()
allSpolysInNextDegree()
cleanTopByChainCriterion()
containsOne()
faugereStepDense()
implications()
llReduceAll()
minimalize()
minimalizeAndTailReduce()
nextSpoly()
nf()
npairs()
select()
smallSpolysInNextDegree()
someSpolysInNextDegree()
suggestPluginVariable()
symmGB_F2()
topSugar()
variableHasValue()

VariableBlock()

class VariableBlockFalse()

class VariableBlockTrue()

class VariableBlock_base()

add_up_polynomials()

append_ring_block()

change_ordering()

contained_vars()

free_m4ri()

get_cring()
Return the currently active global ring, this is only relevant for the native PolyBoRi interface.

```
sage: from polybori import declare_ring, get_cring, Block
sage: R = declare_ring([Block('x',2),Block('y',3)],globals())
sage: Q = get_cring(); Q
Boolean PolynomialRing in x(0), x(1), y(0), y(1), y(2)
sage: R is Q
True
```

get_order_code()

get_var_mapping()
Return a variable mapping between variables of `other` and `ring`. When other is a parent object, the mapping defines images for all variables of other. If it is an element, only variables occurring in other are mapped.

Raises `NameError` if no such mapping is possible.

EXAMPLES:

```
sage: P.<x,y,z> = BooleanPolynomialRing(3)
sage: R.<z,y> = QQ[]
sage: sage.rings.polynomial.pbori.get_var_mapping(P,R)
[z, y]
sage: sage.rings.polynomial.pbori.get_var_mapping(P, z^2)
[z, None]


sage: R.<z,x> = BooleanPolynomialRing(2)
sage: sage.rings.polynomial.pbori.get_var_mapping(P,R)
[z, x]
sage: sage.rings.polynomial.pbori.get_var_mapping(P, x^2)
[None, x]
```

have_degree_order()

if_then_else()
The opposite of navigating down a ZDD using navigators is to construct new ZDDs in the same way, namely giving their else- and then-branch as well as the index value of the new node.

INPUT:

- `root` - a variable
- `a` - the if branch, a `BooleSet` or a `BoolePolynomial`
- `b` - the else branch, a `BooleSet` or a `BoolePolynomial`

EXAMPLE:

```
sage: from polybori import if_then_else
sage: B = BooleanPolynomialRing(6,'x')
sage: x0,x1,x2,x3,x4,x5 = B.gens()
sage: f0 = x2*x3+x3
sage: f1 = x4
sage: if_then_else(x1, f0, f1)
{{x1,x2,x3}, {x1,x3}, {x4}}

sage: if_then_else(x1.lm().index(),f0,f1)
{{x1,x2,x3}, {x1,x3}, {x4}}

sage: if_then_else(x5, f0, f1)
...
IndexError: index of root must be less than the values of roots of the branches.
```

interpolate()

interpolate_smallest_lex()

ll_red_nf()

ll_red_nf_noredsb()

map_every_x_to_x_plus_one()

mod_mon_set()

mod_var_set()

mult_fact_sim_C()

nf3()

parallel_reduce()

recursively_insert()

red_tail()

set_cring()

Set the currently active global ring, this is only relevant for the native PolyBoRi interface.

```
sage: from polybori import *
sage: declare_ring([Block('x',2),Block('y',3)],globals())
Boolean PolynomialRing in x(0), x(1), y(0), y(1), y(2)
sage: R = get_cring(); R
Boolean PolynomialRing in x(0), x(1), y(0), y(1), y(2)

sage: declare_ring([Block('x',2),Block('y',2)],globals())
Boolean PolynomialRing in x(0), x(1), y(0), y(1)

sage: get_cring()
Boolean PolynomialRing in x(0), x(1), y(0), y(1)

sage: set_cring(R)
sage: get_cring()
Boolean PolynomialRing in x(0), x(1), y(0), y(1), y(2)
```

set_variable_name()

top_index()
Return the highest index in the parameter s.

INPUT:

- s - BooleSet, BooleMonomial, BoolePolynomial

EXAMPLE:

```
sage: B.<x,y,z> = BooleanPolynomialRing(3)
sage: from polybori import top_index
sage: top_index(x.lm())
0
sage: top_index(y*z)
1
sage: top_index(x + 1)
0
```

unpickle_BooleanPolynomial()
Unpickle boolean polynomials

EXAMPLE:

```
sage: T = TermOrder('deglex',2)+TermOrder('deglex',2)
sage: P.<a,b,c,d> = BooleanPolynomialRing(4,order=T)
sage: loads(dumps(a+b)) == a+b # indirect doctest
True
```

unpickle_BooleanPolynomialRing()
Unpickle boolean polynomial rings.

EXAMPLE:

```
sage: T = TermOrder('deglex',2)+TermOrder('deglex',2)
sage: P.<a,b,c,d> = BooleanPolynomialRing(4,order=T)
sage: loads(dumps(P)) == P  # indirect doctest
True
```

zeros()

28.15 Generic Convolution.

Asymptotically fast convolution of lists over any commutative ring in which the multiply-by-two map is injective. (More precisely, if $x \in R$, and $x = 2^k * y$ for some $k \geq 0$, we require that $R(x/2^k)$ returns y.)

The main function to be exported is convolution().

EXAMPLES:

```
sage: convolution([1, 2, 3, 4, 5], [6, 7])
[6, 19, 32, 45, 58, 35]
```

The convolution function is reasonably fast, even though it is written in pure Python. For example, the following takes less than a second:

```
sage: v=convolution(range(1000), range(1000))
```

ALGORITHM: Converts the problem to multiplication in the ring $S[x]/(x^M - 1)$, where $S = R[y]/(y^K + 1)$ (where R is the original base ring). Performs FFT with respect to the roots of unity $1, y, y^2, \ldots, y^{2K-1}$ in S. The FFT/IFFT are accomplished with just additions and subtractions and rotating python lists. (I think this algorithm is essentially due to Schonhage, not completely sure.) The pointwise multiplications are handled recursively, switching to a classical algorithm at some point.

Complexity is O(n log(n) log(log(n))) additions/subtractions in R and O(n log(n)) multiplications in R.

AUTHORS:

- David Harvey (2007-07): first implementation
- William Stein: editing the docstrings for inclusion in Sage.

convolution(*L1, L2*)

Returns convolution of non-empty lists L1 and L2. L1 and L2 may have arbitrary lengths.

EXAMPLES:

```
sage: convolution([1, 2, 3], [4, 5, 6, 7])
[4, 13, 28, 34, 32, 21]
```

```
sage: R = Integers(47)
sage: L1 = [R.random_element() for _ in range(1000)]
sage: L2 = [R.random_element() for _ in range(3756)]
sage: L3 = convolution(L1, L2)
sage: L3[2000] == sum([L1[i] * L2[2000-i] for i in range(1000)])
True
sage: len(L3) == 1000 + 3756 - 1
True
```

CHAPTER

TWENTYNINE

POWER SERIES RINGS

29.1 Univariate Power Series Rings

EXAMPLES: Power series rings are constructed in the standard Sage fashion.

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: R.random_element(6)
-4 - 1/2*t^2 - 1/95*t^3 + 1/2*t^4 - 12*t^5 + O(t^6)
```

The default precision is specified at construction, but does not bound the precision of created elements.

```
sage: R.<t> = PowerSeriesRing(QQ, default_prec=5)
sage: R.random_element(6)
1/2 - 1/4*t + 2/3*t^2 - 5/2*t^3 + 2/3*t^5 + O(t^6)


sage: S = R([1, 3, 5, 7]); S  # XXX + O(t^5)
1 + 3*t + 5*t^2 + 7*t^3


sage: S.truncate(3)
5*t^2 + 3*t + 1


sage: S.<w> = PowerSeriesRing(QQ)
sage: S.base_ring()
Rational Field
```

An iterated example:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: S.<t2> = PowerSeriesRing(R)
sage: S
Power Series Ring in t2 over Power Series Ring in t over Integer Ring
sage: S.base_ring()
Power Series Ring in t over Integer Ring
```

We compute with power series over the symbolic ring.

```
sage: K.<t> = PowerSeriesRing(SR, 5)
sage: a, b, c = var('a,b,c')
sage: f = a + b*t + c*t^2 + O(t^3)
sage: f*f
```

```
a^2 + 2*a*b*t + (2*a*c + b^2)*t^2 + O(t^3)
sage: f = sqrt(2) + sqrt(3)*t + O(t^3)
sage: f^2
2 + 2*sqrt(2)*sqrt(3)*t + 3*t^2 + O(t^3)
```

Elements are first coerced to constants in base_ring, then coerced into the PowerSeriesRing:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: f = Mod(2, 3) * t; (f, f.parent())
(2*t, Power Series Ring in t over Ring of integers modulo 3)
```

We make a sparse power series.

```
sage: R.<x> = PowerSeriesRing(QQ, sparse=True); R
Sparse Power Series Ring in x over Rational Field
sage: f = 1 + x^1000000
sage: g = f*f
sage: g.degree()
2000000
```

We make a sparse Laurent series from a power series generator:

```
sage: R.<t> = PowerSeriesRing(QQ, sparse=True)
sage: latex(-2/3*(1/t^3) + 1/t + 3/5*t^2 + O(t^5))
\frac{-\frac{2}{3}}{t^{3}} + \frac{1}{t} + \frac{3}{5}t^{2} + O(t^{5})
sage: S = parent(1/t); S
Sparse Laurent Series Ring in t over Rational Field
```

AUTHORS:

- William Stein: the code
- Jeremy Cho (2006-05-17): some examples (above)

TESTS:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: R == loads(dumps(R))
True
```

```
sage: R.<x> = PowerSeriesRing(QQ, sparse=True)
sage: R == loads(dumps(R))
True
```

PowerSeriesRing(*base_ring, name=None, default_prec=20, names=None, sparse=False*)

Create a power series ring.

INPUT:

- `base_ring` - a commutative ring
- `name` - name of the indeterminate
- `default_prec` - (default: 20) the default precision used if an exact object must be changed to an approximate object in order to do an arithmetic operation.
- `sparse` - (default: False) whether power series are represented as sparse objects.

There is a unique power series ring over each base ring with given variable name. Two power series over the same base ring with different variable names are not equal or isomorphic.

EXAMPLES:

```
sage: R = PowerSeriesRing(QQ, 'x'); R
Power Series Ring in x over Rational Field

sage: S = PowerSeriesRing(QQ, 'y'); S
Power Series Ring in y over Rational Field

sage: R = PowerSeriesRing(QQ, 10)
...
ValueError: first letter of variable name must be a letter

sage: S = PowerSeriesRing(QQ, 'x', default_prec = 15); S
Power Series Ring in x over Rational Field
sage: S.default_prec()
15
```

class **PowerSeriesRing_domain**(*base_ring, name=None, default_prec=20, sparse=False, use_lazy_mpoly_ring=False*)

class **PowerSeriesRing_generic**(*base_ring, name=None, default_prec=20, sparse=False, use_lazy_mpoly_ring=False*)

A power series ring.

base_extend(*R*)

Returns the power series ring over R in the same variable as self, assuming there is a canonical coerce map from the base ring of self to R.

EXAMPLES:

```
sage: R.<T> = GF(7)[[]]; R
Power Series Ring in T over Finite Field of size 7
sage: R.change_ring(ZZ)
Power Series Ring in T over Integer Ring
sage: R.base_extend(ZZ)
...
TypeError: no base extension defined
```

change_ring(*R*)

Returns the power series ring over R in the same variable as self.

EXAMPLES:

```
sage: R.<T> = QQ[[]]; R
Power Series Ring in T over Rational Field
sage: R.change_ring(GF(7))
Power Series Ring in T over Finite Field of size 7
sage: R.base_extend(GF(7))
...
TypeError: no base extension defined
sage: R.base_extend(QuadraticField(3,'a'))
Power Series Ring in T over Number Field in a with defining polynomial x^2 - 3
```

characteristic()

Return the characteristic of this power series ring, which is the same as the characteristic of the base ring of the power series ring.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: R.characteristic()
0
sage: R.<w> = Integers(2^50)[[]]; R
Power Series Ring in w over Ring of integers modulo 1125899906842624
sage: R.characteristic()
1125899906842624
```

construction()

Returns the functorial construction of self, namely, completion of the univariate polynomial ring with respect to the indeterminate (to a given precision).

EXAMPLE:

```
sage: R = PowerSeriesRing(ZZ, 'x')
sage: c, S = R.construction(); S
Univariate Polynomial Ring in x over Integer Ring
sage: R == c(S)
True
```

gen(*n=0*)

Return the generator of this power series ring.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: R.gen()
t
sage: R.gen(3)
...
IndexError: generator n>0 not defined
```

is_atomic_repr()

Return False since power objects do not appear atomically, i.e., they have plus and spaces.

is_dense()

is_exact()

is_field()

Return False since the ring of power series over any ring is never a field.

is_finite()

Return False since the ring of power series over any ring is never finite.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: R.is_finite()
False
```

is_sparse()

laurent_series_ring()

If this is the power series ring $R[[t]]$, this function returns the Laurent series ring $R((t))$.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: R.laurent_series_ring()
Laurent Series Ring in t over Integer Ring
```

ngens()

Return the number of generators of this power series ring.

This is always 1.

EXAMPLES:

```
sage: R.<t> = ZZ[[]]
sage: R.ngens()
1
```

random_element(*prec, bound=None*)

Return a random power series.

INPUT:

- `prec` - an integer
- `bound` - an integer (default: None, which tries to spread choice across ring, if implemented)

OUTPUT:

- `power series` - a power series such that the coefficient of x^i, for i up to `degree`, are coercions to the base ring of random integers between -`bound` and `bound`.

IMPLEMENTATION: Call the random_element method on the underlying polynomial ring.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: R.random_element(5)
-4 - 1/2*t^2 - 1/95*t^3 + 1/2*t^4 + O(t^5)
sage: R.random_element(5,20)
1/15 + 19/17*t + 10/3*t^2 + 5/2*t^3 + 1/2*t^4 + O(t^5)
```

class **PowerSeriesRing_over_field**(*base_ring, name=None, default_prec=20, sparse=False, use_lazy_mpoly_ring=False*)

fraction_field()

Return the fraction field of this power series ring, which is defined since this is over a field.

This fraction field is just the Laurent series ring over the base field.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(GF(7))
sage: R.fraction_field()
Laurent Series Ring in t over Finite Field of size 7
sage: Frac(R)
Laurent Series Ring in t over Finite Field of size 7
```

is_PowerSeriesRing(*R*)

Return True if R is a power series ring.

EXAMPLES:

```
sage: from sage.rings.power_series_ring import is_PowerSeriesRing
sage: is_PowerSeriesRing(10)
False
sage: is_PowerSeriesRing(QQ[['x']])
True
```

unpickle_power_series_ring_v0(*base_ring, name, default_prec, sparse*)

29.2 Power Series

Sage provides an implementation of dense and sparse power series over any Sage base ring.

AUTHORS:

- William Stein

- David Harvey (2006-09-11): added solve_linear_de() method
- Robert Bradshaw (2007-04): sqrt, rmul, lmul, shifting
- Robert Bradshaw (2007-04): Cython version

EXAMPLE:

```
sage: R.<x> = PowerSeriesRing(ZZ)
sage: R([1,2,3])
1 + 2*x + 3*x^2
sage: R([1,2,3], 10)
1 + 2*x + 3*x^2 + O(x^10)
sage: f = 1 + 2*x - 3*x^3 + O(x^4); f
1 + 2*x - 3*x^3 + O(x^4)
sage: f^10
1 + 20*x + 180*x^2 + 930*x^3 + O(x^4)
sage: g = 1/f; g
1 - 2*x + 4*x^2 - 5*x^3 + O(x^4)
sage: g * f
1 + O(x^4)
```

In Python (as opposed to Sage) create the power series ring and its generator as follows:

```
sage: R, x = objgen(PowerSeriesRing(ZZ, 'x'))
sage: parent(x)
Power Series Ring in x over Integer Ring
```

EXAMPLE:

This example illustrates that coercion for power series rings is consistent with coercion for polynomial rings.

```
sage: poly_ring1.<gen1> = PolynomialRing(QQ)
sage: poly_ring2.<gen2> = PolynomialRing(QQ)
sage: huge_ring.<x> = PolynomialRing(poly_ring1)
```

The generator of the first ring gets coerced in as itself, since it is the base ring.

```
sage: huge_ring(gen1)
gen1
```

The generator of the second ring gets mapped via the natural map sending one generator to the other.

```
sage: huge_ring(gen2)
x
```

With power series the behavior is the same.

```
sage: power_ring1.<gen1> = PowerSeriesRing(QQ)
sage: power_ring2.<gen2> = PowerSeriesRing(QQ)
sage: huge_power_ring.<x> = PowerSeriesRing(power_ring1)
sage: huge_power_ring(gen1)
gen1
sage: huge_power_ring(gen2)
x
```

TODO: Rewrite valuation so it is *carried* along after any calculation, so in almost all cases f.valuation() is instant. Also, if you add f and g and their valuations are the same, note that we only have to look at terms at positions = f.valuation().

class PowerSeries()

A power series.

O()

Return this series plus $O(x^{\text{prec}})$. Does not change self.

V()

If $f = \sum a_m x^m$, then this function returns $\sum a_m x^{nm}$.

add_bigoh()

Returns the power series of precision at most prec got by adding $O(q^{\text{prec}})$ to f, where q is the variable.

EXAMPLES:

```
sage: R.<A> = RDF[[]]
sage: f = (1+A+O(A^5))^5; f
1.0 + 5.0*A + 10.0*A^2 + 10.0*A^3 + 5.0*A^4 + O(A^5)
sage: f.add_bigoh(3)
1.0 + 5.0*A + 10.0*A^2 + O(A^3)
sage: f.add_bigoh(5)
1.0 + 5.0*A + 10.0*A^2 + 10.0*A^3 + 5.0*A^4 + O(A^5)
```

base_extend()

Return a copy of this power series but with coefficients in R.

The following coercion uses base_extend implicitly:

```
sage: R.<t> = ZZ[['t']]
sage: (t - t^2) * Mod(1, 3)
t + 2*t^2
```

base_ring()

Return the base ring that this power series is defined over.

EXAMPLES:

```
sage: R.<t> = GF(49,'alpha')[[]]
sage: (t^2 + O(t^3)).base_ring()
Finite Field in alpha of size 7^2
```

change_ring()

Change if possible the coefficients of self to lie in R.

EXAMPLES:

```
sage: R.<T> = QQ[[]]; R
Power Series Ring in T over Rational Field
sage: f = 1 - 1/2*T + 1/3*T^2 + O(T^3)
sage: f.base_extend(GF(5))
...
TypeError: no base extension defined
sage: f.change_ring(GF(5))
1 + 2*T + 2*T^2 + O(T^3)
sage: f.change_ring(GF(3))
...
ZeroDivisionError: Inverse does not exist.
```

We can only change ring if there is a __call__ coercion defined. The following succeeds because ZZ(K(4)) is defined.

```
sage: K.<a> = NumberField(cyclotomic_polynomial(3), 'a')
sage: R.<t> = K[['t']]
sage: (4*t).change_ring(ZZ)
4*t
```

This does not succeed because ZZ(K(a+1)) is not defined.

```
sage: K.<a> = NumberField(cyclotomic_polynomial(3), 'a')
sage: R.<t> = K[['t']]
sage: ((a+1)*t).change_ring(ZZ)
...
TypeError: Unable to coerce a + 1 to an integer
```

coefficients()

Return the nonzero coefficients of self.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: f = t + t^2 - 10/3*t^3
sage: f.coefficients()
[1, 1, -10/3]
```

common_prec()

Returns minimum precision of f and self.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)

sage: f = t + t^2 + O(t^3)
sage: g = t + t^3 + t^4 + O(t^4)
sage: f.common_prec(g)
3
sage: g.common_prec(f)
3

sage: f = t + t^2 + O(t^3)
sage: g = t^2
sage: f.common_prec(g)
3
sage: g.common_prec(f)
3

sage: f = t + t^2
sage: f = t^2
sage: f.common_prec(g)
+Infinity
```

degree()

Return the degree of this power series, which is by definition the degree of the underlying polynomial.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ, sparse=True)
sage: f = t^100000 + O(t^10000000)
sage: f.degree()
100000
```

derivative()

The formal derivative of this power series, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See Also:

`_derivative()`

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(QQ)
sage: g = -x + x^2/2 - x^4 + O(x^6)
sage: g.derivative()
-1 + x - 4*x^3 + O(x^5)
sage: g.derivative(x)
-1 + x - 4*x^3 + O(x^5)
sage: g.derivative(x, x)
1 - 12*x^2 + O(x^4)
sage: g.derivative(x, 2)
1 - 12*x^2 + O(x^4)
```

egf()

Returns the exponential generating function associated to self.

This function is known as serlaplace in GP/PARI.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: f = t + t^2 + 2*t^3
sage: f.egf()
t + 1/2*t^2 + 1/3*t^3
```

exp()

Returns exp of this power series to the indicated precision.

INPUT:

- `prec` - integer; default is self.parent().default_prec

ALGORITHM: See PowerSeries.solve_linear_de().

Note:

- Screwy things can happen if the coefficient ring is not a field of characteristic zero. See PowerSeries.solve_linear_de().

AUTHORS:

- David Harvey (2006-09-08): rewrote to use simplest possible "lazy" algorithm.
- David Harvey (2006-09-10): rewrote to use divide-and-conquer strategy.
- David Harvey (2006-09-11): factored functionality out to solve_linear_de().
- Sourav Sen Gupta, David Harvey (2008-11): handle constant term

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ, default_prec=10)
```

Check that $\exp(t)$ is, well, $\exp(t)$:

```
sage: (t + O(t^10)).exp()
1 + t + 1/2*t^2 + 1/6*t^3 + 1/24*t^4 + 1/120*t^5 + 1/720*t^6 + 1/5040*t^7 + 1/40320*t^8 + 1/
```

Check that $\exp(\log(1+t))$ is $1+t$:

```
sage: (sum([-(-t)^n/n for n in range(1, 10)]) + O(t^10)).exp()
1 + t + O(t^10)
```

Check that $\exp(2t + t^2 - t^5)$ is whatever it is:

```
sage: (2*t + t^2 - t^5 + O(t^10)).exp()
1 + 2*t + 3*t^2 + 10/3*t^3 + 19/6*t^4 + 8/5*t^5 - 7/90*t^6 - 538/315*t^7 - 425/168*t^8 - 306
```

Check requesting lower precision:

```
sage: (t + t^2 - t^5 + O(t^10)).exp(5)
1 + t + 3/2*t^2 + 7/6*t^3 + 25/24*t^4 + O(t^5)
```

Can't get more precision than the input:

```
sage: (t + t^2 + O(t^3)).exp(10)
1 + t + 3/2*t^2 + O(t^3)
```

Check some boundary cases:

```
sage: (t + O(t^2)).exp(1)
1 + O(t)
sage: (t + O(t^2)).exp(0)
O(t^0)
```

Handle nonzero constant term (fixes trac #4477):

```
sage: R.<x> = PowerSeriesRing(RR)
sage: (1 + x + x^2 + O(x^3)).exp()
2.71828182845905 + 2.71828182845905*x + 4.07742274268857*x^2 + O(x^3)

sage: R.<x> = PowerSeriesRing(ZZ)
sage: (1 + x + O(x^2)).exp()
...
ArithmeticError: exponential of constant term does not belong to coefficient ring (consider

sage: R.<x> = PowerSeriesRing(GF(5))
sage: (1 + x + O(x^2)).exp()
...
ArithmeticError: constant term of power series does not support exponentiation
```

exponents()

Return the exponents appearing in self with nonzero coefficients.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: f = t + t^2 - 10/3*t^3
sage: f.exponents()
[1, 2, 3]
```

is_dense()

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: t.is_dense()
True
sage: R.<t> = PowerSeriesRing(ZZ, sparse=True)
sage: t.is_dense()
False
```

is_gen()

Returns True if this the generator (the variable) of the power series ring.

EXAMPLES:

```
sage: R.<t> = QQ[[]]
sage: t.is_gen()
True
sage: (1 + 2*t).is_gen()
False
```

Note that this only returns true on the actual generator, not on something that happens to be equal to it.

```
sage: (1*t).is_gen()
False
sage: 1*t == t
True
```

is_sparse()

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: t.is_sparse()
False
sage: R.<t> = PowerSeriesRing(ZZ, sparse=True)
sage: t.is_sparse()
True
```

is_square()

Returns True if this function has a square root in this ring, e.g. there is an element y in `self.parent()` such that $y^2 =$ "$self$".

ALGORITHM: If the base ring is a field, this is true whenever the power series has even valuation and the leading coefficient is a perfect square.

For an integral domain, it attempts the square root in the fraction field and tests whether or not the result lies in the original ring.

EXAMPLES:

```
sage: K.<t> = PowerSeriesRing(QQ, 't', 5)
sage: (1+t).is_square()
True
sage: (2+t).is_square()
False
sage: (2+t.change_ring(RR)).is_square()
True
sage: t.is_square()
False
sage: K.<t> = PowerSeriesRing(ZZ, 't', 5)
sage: (1+t).is_square()
False
sage: f = (1+t)^100
sage: f.is_square()
True
```

is_unit()

Returns whether this power series is invertible, which is the case precisely when the constant term is invertible.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: (-1 + t - t^5).is_unit()
True
sage: (3 + t - t^5).is_unit()
False
```

AUTHORS:

- David Harvey (2006-09-03)

laurent_series()

Return the Laurent series associated to this power series, i.e., this series considered as a Laurent series.

EXAMPLES:

```
sage: k.<w> = QQ[[]]
sage: f = 1+17*w+15*w^3+O(w^5)
sage: parent(f)
Power Series Ring in w over Rational Field
sage: g = f.laurent_series(); g
1 + 17*w + 15*w^3 + O(w^5)
```

list()

ogf()

Returns the ordinary generating function associated to self.

This function is known as serlaplace in GP/PARI.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: f = t + t^2/factorial(2) + 2*t^3/factorial(3)
sage: f.ogf()
t + t^2 + 2*t^3
```

padded_list()

Return a list of coefficients of self up to (but not including) q^n.

Includes 0's in the list on the right so that the list has length n.

INPUT:

- n - an integer that is at least 0

EXAMPLES:

```
sage: R.<q> = PowerSeriesRing(QQ)
sage: f = 1 - 17*q + 13*q^2 + 10*q^4 + O(q^7)
sage: f.list()
[1, -17, 13, 0, 10]
sage: f.padded_list(7)
[1, -17, 13, 0, 10, 0, 0]
sage: f.padded_list(10)
[1, -17, 13, 0, 10, 0, 0, 0, 0, 0]
sage: f.padded_list(3)
[1, -17, 13]
```

polynomial()

prec()

The precision of $... + O(x^r)$ is by definition r.

EXAMPLES:

```
sage: R.<t> = ZZ[[]]
sage: (t^2 + O(t^3)).prec()
3
sage: (1 - t^2 + O(t^100)).prec()
100
```

shift()

Returns this power series multiplied by the power t^n. If n is negative, terms below t^n will be discarded. Does not change this power series.

Note: Despite the fact that higher order terms are printed to the right in a power series, right shifting decreases the powers of t, while left shifting increases them. This is to be consistent with polynomials, integers, etc.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ['y'], 't', 5)
sage: f = ~(1+t); f
1 - t + t^2 - t^3 + t^4 + O(t^5)
sage: f.shift(3)
t^3 - t^4 + t^5 - t^6 + t^7 + O(t^8)
sage: f >> 2
1 - t + t^2 + O(t^3)
sage: f << 10
t^10 - t^11 + t^12 - t^13 + t^14 + O(t^15)
sage: t << 29
t^30
```

AUTHORS:

- Robert Bradshaw (2007-04-18)

solve_linear_de()

Obtains a power series solution to an inhomogeneous linear differential equation of the form:

$$f'(t) = a(t)f(t) + b(t).$$

INPUT:

- `self` - the power series $a(t)$
- `b` - the power series $b(t)$ (default is zero)
- `f0` - the constant term of f ("initial condition") (default is 1)
- `prec` - desired precision of result (this will be reduced if either a or b have less precision available)

OUTPUT: the power series f, to indicated precision

ALGORITHM: A divide-and-conquer strategy; see the source code. Running time is approximately $M(n)\log n$, where $M(n)$ is the time required for a polynomial multiplication of length n over the coefficient ring. (If you're working over something like RationalField(), running time analysis can be a little complicated because the coefficients tend to explode.)

Note:

- If the coefficient ring is a field of characteristic zero, then the solution will exist and is unique.
- For other coefficient rings, things are more complicated. A solution may not exist, and if it does it may not be unique. Generally, by the time the nth term has been computed, the algorithm will have attempted divisions by $n!$ in the coefficient ring. So if your coefficient ring has enough 'precision', and if your coefficient ring can perform divisions even when the answer is not unique, and if you know in advance that a solution exists, then this function will find a solution (otherwise it will probably crash).

AUTHORS:

- David Harvey (2006-09-11): factored functionality out from exp() function, cleaned up precision tests a bit

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ, default_prec=10)

sage: a = 2 - 3*t + 4*t^2 + O(t^10)
sage: b = 3 - 4*t^2 + O(t^7)
sage: f = a.solve_linear_de(prec=5, b=b, f0=3/5)
sage: f
 3/5 + 21/5*t + 33/10*t^2 - 38/15*t^3 + 11/24*t^4 + O(t^5)
sage: f.derivative() - a*f - b
 O(t^4)
```

```
sage: a = 2 - 3*t + 4*t^2
sage: b = b = 3 - 4*t^2
sage: f = a.solve_linear_de(b=b, f0=3/5)
...
ValueError: cannot solve differential equation to infinite precision

sage: a.solve_linear_de(prec=5, b=b, f0=3/5)
 3/5 + 21/5*t + 33/10*t^2 - 38/15*t^3 + 11/24*t^4 + O(t^5)
```

sqrt()

The square root function.

INPUT:

- prec - integer (default: None): if not None and the series has infinite precision, truncates series at precision prec.
- extend - bool (default: False); if True, return a square root in an extension ring, if necessary. Otherwise, raise a ValueError if the square is not in the base power series ring. For example, if extend is True the square root of a power series with odd degree leading coefficient is defined as an element of a formal extension ring.
- name - if extend is True, you must also specify the print name of the formal square root.
- all - bool (default: False); if True, return all square roots of self, instead of just one.

ALGORITHM: Newton's method

$$x_{i+1} = \frac{1}{2}(x_i + self/x_i)$$

EXAMPLES:

```
sage: K.<t> = PowerSeriesRing(QQ, 't', 5)
sage: sqrt(t^2)
t
sage: sqrt(1+t)
1 + 1/2*t - 1/8*t^2 + 1/16*t^3 - 5/128*t^4 + O(t^5)
sage: sqrt(4+t)
2 + 1/4*t - 1/64*t^2 + 1/512*t^3 - 5/16384*t^4 + O(t^5)
sage: u = sqrt(2+t, prec=2, extend=True, name = 'alpha'); u
alpha
sage: u^2
2 + t
sage: u.parent()
Univariate Quotient Polynomial Ring in alpha over Power Series Ring in t over Rational Field
sage: K.<t> = PowerSeriesRing(QQ, 't', 50)
sage: sqrt(1+2*t+t^2)
1 + t
sage: sqrt(t^2 +2*t^4 + t^6)
t + t^3
sage: sqrt(1 + t + t^2 + 7*t^3)^2
1 + t + t^2 + 7*t^3 + O(t^50)
sage: sqrt(K(0))
0
sage: sqrt(t^2)
t

sage: K.<t> = PowerSeriesRing(CDF, 5)
sage: v = sqrt(-1 + t + t^3, all=True); v
[1.0*I - 0.5*I*t - 0.125*I*t^2 - 0.5625*I*t^3 - 0.2890625*I*t^4 + O(t^5),
 -1.0*I + 0.5*I*t + 0.125*I*t^2 + 0.5625*I*t^3 + 0.2890625*I*t^4 + O(t^5)]
sage: [a^2 for a in v]
[-1.0 + 1.0*t + 1.0*t^3 + O(t^5), -1.0 + 1.0*t + 1.0*t^3 + O(t^5)]
```

A formal square root:

```
sage: K.<t> = PowerSeriesRing(QQ, 5)
sage: f = 2*t + t^3 + O(t^4)
sage: s = f.sqrt(extend=True, name='sqrtf'); s
sqrtf
sage: s^2
2*t + t^3 + O(t^4)
sage: parent(s)
Univariate Quotient Polynomial Ring in sqrtf over Power Series Ring in t over Rational Field
```

AUTHORS:

- Robert Bradshaw
- William Stein

square_root()

Return the square root of self in this ring. If this cannot be done then an error will be raised.

This function succeeds if and only if `self.is_square()`

EXAMPLES:

```
sage: K.<t> = PowerSeriesRing(QQ, 't', 5)
sage: (1+t).square_root()
1 + 1/2*t - 1/8*t^2 + 1/16*t^3 - 5/128*t^4 + O(t^5)
sage: (2+t).square_root()
...
ValueError: Square root does not live in this ring.
sage: (2+t.change_ring(RR)).square_root()
1.41421356237309 + 0.353553390593274*t - 0.0441941738241592*t^2 + 0.0110485434560398*t^3 - 0
sage: t.square_root()
...
ValueError: Square root not defined for power series of odd valuation.
sage: K.<t> = PowerSeriesRing(ZZ, 't', 5)
sage: f = (1+t)^20
sage: f.square_root()
1 + 10*t + 45*t^2 + 120*t^3 + 210*t^4 + O(t^5)
sage: f = 1+t
sage: f.square_root()
...
ValueError: Square root does not live in this ring.
```

AUTHORS:

- Robert Bradshaw

truncate()

The polynomial obtained from power series by truncation.

EXAMPLES:

```
sage: R.<I> = GF(2)[[]]
sage: f = 1/(1+I+O(I^8)); f
1 + I + I^2 + I^3 + I^4 + I^5 + I^6 + I^7 + O(I^8)
sage: f.truncate(5)
I^4 + I^3 + I^2 + I + 1
```

valuation()

Return the valuation of this power series.

This is equal to the valuation of the underlying polynomial.

EXAMPLES: Sparse examples:

```
sage: R.<t> = PowerSeriesRing(QQ, sparse=True)
sage: f = t^100000 + O(t^10000000)
sage: f.valuation()
100000
sage: R(0).valuation()
+Infinity
```

Dense examples:

```
sage: R.<t> = PowerSeriesRing(ZZ)
sage: f = 17*t^100 +O(t^110)
sage: f.valuation()
100
sage: t.valuation()
1
```

valuation_zero_part()

Factor self as as $q^n \cdot (a_0 + a_1 q + \cdots)$ with a_0 nonzero. Then this function returns $a_0 + a_1 q + \cdots$.

Note: This valuation zero part need not be a unit if, e.g., a_0 is not invertible in the base ring.

EXAMPLES:

```
sage: R.<t> = PowerSeriesRing(QQ)
sage: ((1/3)*t^5*(17-2/3*t^3)).valuation_zero_part()
17/3 - 2/9*t^3
```

In this example the valuation 0 part is not a unit:

```
sage: R.<t> = PowerSeriesRing(ZZ, sparse=True)
sage: u = (-2*t^5*(17-t^3)).valuation_zero_part(); u
-34 + 2*t^3
sage: u.is_unit()
False
sage: u.valuation()
0
```

variable()

EXAMPLES:

```
sage: R.<x> = PowerSeriesRing(Rationals())
sage: f = x^2 + 3*x^4 + O(x^7)
sage: f.variable()
'x'
```

AUTHORS:

- David Harvey (2006-08-08)

is_PowerSeries()

make_element_from_parent_v0()

make_powerseries_poly_v0()

29.3 Laurent Series Rings

EXAMPLES:

```
sage: R = LaurentSeriesRing(QQ, "x")
sage: R.base_ring()
Rational Field
```

```
sage: S = LaurentSeriesRing(GF(17)['x'], 'y')
sage: S
Laurent Series Ring in y over Univariate Polynomial Ring in x over
Finite Field of size 17
sage: S.base_ring()
Univariate Polynomial Ring in x over Finite Field of size 17
```

LaurentSeriesRing(*base_ring, name=None, names=None, sparse=False*)

EXAMPLES:

```
sage: R = LaurentSeriesRing(QQ, 'x'); R
Laurent Series Ring in x over Rational Field
sage: x = R.0
sage: g = 1 - x + x^2 - x^4 +O(x^8); g
1 - x + x^2 - x^4 + O(x^8)
sage: g = 10*x^(-3) + 2006 - 19*x + x^2 - x^4 +O(x^8); g
10*x^-3 + 2006 - 19*x + x^2 - x^4 + O(x^8)
```

You can also use more mathematical notation when the base is a field:

```
sage: Frac(QQ[['x']])
Laurent Series Ring in x over Rational Field
sage: Frac(GF(5)['y'])
Fraction Field of Univariate Polynomial Ring in y over Finite Field of size 5
```

Here the fraction field is not just the Laurent series ring, so you can't use the `Frac` notation to make the Laurent series ring.

```
sage: Frac(ZZ[['t']])
Fraction Field of Power Series Ring in t over Integer Ring
```

Laurent series rings are determined by their variable and the base ring, and are globally unique.

```
sage: K = Qp(5, prec = 5)
sage: L = Qp(5, prec = 200)
sage: R.<x> = LaurentSeriesRing(K)
sage: S.<y> = LaurentSeriesRing(L)
sage: R is S
False
sage: T.<y> = LaurentSeriesRing(Qp(5,prec=200))
sage: S is T
True
sage: W.<y> = LaurentSeriesRing(Qp(5,prec=199))
sage: W is T
False
```

class **LaurentSeriesRing_domain**(*base_ring, name=None, sparse=False*)

class **LaurentSeriesRing_field**(*base_ring, name=None, sparse=False*)

class **LaurentSeriesRing_generic**(*base_ring, name=None, sparse=False*)

Univariate Laurent Series Ring

EXAMPLES:

```
sage: K, q = LaurentSeriesRing(CC, 'q').objgen(); K
Laurent Series Ring in q over Complex Field with 53 bits of precision
sage: loads(K.dumps()) == K
True
```

base_extend(*R*)
Returns the laurent series ring over R in the same variable as self, assuming there is a canonical coerce map from the base ring of self to R.

change_ring(*R*)

characteristic()

default_prec()

gen(*n=0*)

is_dense()

is_exact()

is_field()

is_sparse()

ngens()

power_series_ring()
If this is the Laurent series ring $R((t))$, return the power series ring $R[[t]]$.
EXAMPLES:

```
sage: R = LaurentSeriesRing(QQ, "x")
sage: R.power_series_ring()
Power Series Ring in x over Rational Field
```

set_default_prec(*n*)

is_LaurentSeriesRing(*x*)

29.4 Laurent Series

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(GF(7), 't'); R
Laurent Series Ring in t over Finite Field of size 7
sage: f = 1/(1-t+O(t^10)); f
1 + t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + t^8 + t^9 + O(t^10)
```

Laurent series are immutable:

```
sage: f[2]
1
sage: f[2] = 5
...
IndexError: Laurent series are immutable
```

We compute with a Laurent series over the complex mpfr numbers.

```
sage: K.<q> = Frac(CC[['q']])
sage: K
Laurent Series Ring in q over Complex Field with 53 bits of precision
sage: q
1.00000000000000*q
```

Saving and loading.

```
sage: loads(q.dumps()) == q
True
sage: loads(K.dumps()) == K
True
```

IMPLEMENTATION: Laurent series in Sage are represented internally as a power of the variable times the unit part (which need not be a unit - it's a polynomial with nonzero constant term). The zero Laurent series has unit part 0.

AUTHORS:

- William Stein: original version
- David Joyner (2006-01-22): added examples
- Robert Bradshaw (2007-04): optimizations, shifting
- Robert Bradshaw: Cython version

class LaurentSeries()

A Laurent Series.

add_bigoh()

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = t^2 + t^3 + O(t^10); f
t^2 + t^3 + O(t^10)
sage: f.add_bigoh(5)
t^2 + t^3 + O(t^5)
```

change_ring()

coefficients()

Return the nonzero coefficients of self.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = -5/t^(2) + t + t^2 - 10/3*t^3
sage: f.coefficients()
[-5, 1, 1, -10/3]
```

common_prec()

Returns minimum precision of f and self.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)

sage: f = t^(-1) + t + t^2 + O(t^3)
sage: g = t + t^3 + t^4 + O(t^4)
sage: f.common_prec(g)
3
sage: g.common_prec(f)
3

sage: f = t + t^2 + O(t^3)
sage: g = t^(-3) + t^2
sage: f.common_prec(g)
3
sage: g.common_prec(f)
3
```

```
sage: f = t + t^2
sage: f = t^2
sage: f.common_prec(g)
+Infinity

sage: f = t^(-3) + O(t^(-2))
sage: g = t^(-5) + O(t^(-1))
sage: f.common_prec(g)
-2
```

copy()

degree()

Return the degree of a polynomial equivalent to this power series modulo big oh of the precision.

EXAMPLES:

```
sage: x = Frac(QQ[['x']]).0
sage: g = x^2 - x^4 + O(x^8)
sage: g.degree()
4
sage: g = -10/x^5 + x^2 - x^4 + O(x^8)
sage: g.degree()
4
```

derivative()

The formal derivative of this Laurent series, with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

See Also:

_derivative()

EXAMPLES:

```
sage: R.<x> = LaurentSeriesRing(QQ)
sage: g = 1/x^10 - x + x^2 - x^4 + O(x^8)
sage: g.derivative()
-10*x^-11 - 1 + 2*x - 4*x^3 + O(x^7)
sage: g.derivative(x)
-10*x^-11 - 1 + 2*x - 4*x^3 + O(x^7)

sage: R.<t> = PolynomialRing(ZZ)
sage: S.<x> = LaurentSeriesRing(R)
sage: f = 2*t/x + (3*t^2 + 6*t)*x + O(x^2)
sage: f.derivative()
-2*t*x^-2 + (3*t^2 + 6*t) + O(x)
sage: f.derivative(x)
-2*t*x^-2 + (3*t^2 + 6*t) + O(x)
sage: f.derivative(t)
2*x^-1 + (6*t + 6)*x + O(x^2)
```

exponents()

Return the exponents appearing in self with nonzero coefficients.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = -5/t^(2) + t + t^2 - 10/3*t^3
sage: f.exponents()
[-2, 1, 2, 3]
```

integral()

The formal integral of this Laurent series with 0 constant term.

EXAMPLES: The integral may or may not be defined if the base ring is not a field.

```
sage: t = LaurentSeriesRing(ZZ, 't').0
sage: f = 2*t^-3 + 3*t^2 + O(t^4)
sage: f.integral()
-t^-2 + t^3 + O(t^5)

sage: f = t^3
sage: f.integral()
...
ArithmeticError: Coefficients of integral cannot be coerced into the base ring
```

The integral of 1/t is $\log(t)$, which is not given by a Laurent series:

```
sage: t = Frac(QQ['t']).0
sage: f = -1/t^3 - 31/t + O(t^3)
sage: f.integral()
...
ArithmeticError: The integral of is not a Laurent series, since t^-1 has nonzero coefficient
```

Another example with just one negative coefficient:

```
sage: A.<t> = QQ[[]]
sage: f = -2*t^(-4) + O(t^8)
sage: f.integral()
2/3*t^-3 + O(t^9)
sage: f.integral().derivative() == f
True
```

is_unit()

Returns True if this is Laurent series is a unit in this ring.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: (2+t).is_unit()
True
sage: f = 2+t^2+O(t^10); f.is_unit()
True
sage: 1/f
1/2 - 1/4*t^2 + 1/8*t^4 - 1/16*t^6 + 1/32*t^8 + O(t^10)
sage: R(0).is_unit()
False
sage: R.<s> = LaurentSeriesRing(ZZ)
sage: f = 2 + s^2 + O(s^10)
sage: f.is_unit()
False
sage: 1/f
...
ArithmeticError: division not defined
```

ALGORITHM: A Laurent series is a unit if and only if its "unit part" is a unit.

is_zero()

EXAMPLES:

```
sage: x = Frac(QQ['x']).0
sage: f = 1/x + x + x^2 + 3*x^4 + O(x^7)
sage: f.is_zero()
0
```

```
sage: z = 0*f
sage: z.is_zero()
1
```

list()

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ)
sage: f = -5/t^(2) + t + t^2 - 10/3*t^3
sage: f.list()
[-5, 0, 0, 1, 1, -10/3]
```

power_series()

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(ZZ)
sage: f = 1/(1-t+O(t^10)); f.parent()
Laurent Series Ring in t over Integer Ring
sage: g = f.power_series(); g
1 + t + t^2 + t^3 + t^4 + t^5 + t^6 + t^7 + t^8 + t^9 + O(t^10)
sage: parent(g)
Power Series Ring in t over Integer Ring
sage: f = 3/t^2 +  t^2 + t^3 + O(t^10)
sage: f.power_series()
...
ArithmeticError: self is a not a power series
```

prec()

This function returns the n so that the Laurent series is of the form (stuff) + $O(t^n)$. It doesn't matter how many negative powers appear in the expansion. In particular, prec could be negative.

EXAMPLES:

```
sage: x = Frac(QQ[['x']]).0
sage: f = x^2 + 3*x^4 + O(x^7)
sage: f.prec()
7
sage: g = 1/x^10 - x + x^2 - x^4 + O(x^8)
sage: g.prec()
8
```

shift()

Returns this laurent series multiplied by the power t^n. Does not change this series.

Note: Despite the fact that higher order terms are printed to the right in a power series, right shifting decreases the powers of t, while left shifting increases them. This is to be consistant with polynomials, integers, etc.

EXAMPLES:

```
sage: R.<t> = LaurentSeriesRing(QQ['y'])
sage: f = (t+t^-1)^4; f
t^-4 + 4*t^-2 + 6 + 4*t^2 + t^4
sage: f.shift(10)
t^6 + 4*t^8 + 6*t^10 + 4*t^12 + t^14
sage: f >> 10
t^-14 + 4*t^-12 + 6*t^-10 + 4*t^-8 + t^-6
sage: t << 4
t^5
sage: t + O(t^3) >> 4
t^-3 + O(t^-1)
```

AUTHORS:

•Robert Bradshaw (2007-04-18)

truncate()
Returns the laurent series of degree ‘ < n‘ which is equivalent to self modulo x^n.

truncate_neg()
Returns the laurent series equivalent to self except without any degree n terms.

This is equivalent to ‘`self - self.truncate(n)`‘.

valuation()
EXAMPLES:

```
sage: x = Frac(QQ[['x']]).0
sage: f = 1/x + x^2 + 3*x^4 + O(x^7)
sage: g = 1 - x + x^2 - x^4 + O(x^8)
sage: f.valuation()
-1
sage: g.valuation()
0
```

valuation_zero_part()
EXAMPLES:

```
sage: x = Frac(QQ[['x']]).0
sage: f = x + x^2 + 3*x^4 + O(x^7)
sage: f/x
1 + x + 3*x^3 + O(x^6)
sage: f.valuation_zero_part()
1 + x + 3*x^3 + O(x^6)
sage: g = 1/x^7 - x + x^2 - x^4 + O(x^8)
sage: g.valuation_zero_part()
1 - x^8 + x^9 - x^11 + O(x^15)
```

variable()
EXAMPLES:

```
sage: x = Frac(QQ[['x']]).0
sage: f = 1/x + x^2 + 3*x^4 + O(x^7)
sage: f.variable()
'x'
```

is_LaurentSeries()

make_element_from_parent()

CHAPTER

THIRTY

ALGEBRAS

30.1 Free algebras

AUTHORS:

- David Kohel (2005-09)
- William Stein (2006-11-01): add all doctests; implemented many things.

EXAMPLES:

```
sage: F = FreeAlgebra(ZZ,3,'x,y,z')
sage: F.base_ring()
Integer Ring
sage: G = FreeAlgebra(F, 2, 'm,n'); G
Free Algebra on 2 generators (m, n) over Free Algebra on 3 generators (x, y, z) over Integer Ring
sage: G.base_ring()
Free Algebra on 3 generators (x, y, z) over Integer Ring
```

TESTS:

```
sage: F = FreeAlgebra(GF(5),3,'x')
sage: F == loads(dumps(F))
True


sage: F.<x,y,z> = FreeAlgebra(GF(5),3)
sage: F == loads(dumps(F))
True


sage: F = FreeAlgebra(GF(5),3, ['xx', 'zba', 'Y'])
sage: F == loads(dumps(F))
True


sage: F = FreeAlgebra(GF(5),3, 'abc')
sage: F == loads(dumps(F))
True


sage: F = FreeAlgebra(FreeAlgebra(ZZ,1,'a'), 2, 'x')
sage: F == loads(dumps(F))
True
```

FreeAlgebra(*R, n, names*)

Return the free algebra over the ring R on n generators with given names.

INPUT:

- R - ring
- n - integer
- names - string or list/tuple of n strings

OUTPUT: a free algebra

EXAMPLES:

```
sage: FreeAlgebra(GF(5),3,'x')
Free Algebra on 3 generators (x0, x1, x2) over Finite Field of size 5
sage: F.<x,y,z> = FreeAlgebra(GF(5),3)
sage: (x+y+z)^2
x^2 + x*y + x*z + y*x + y^2 + y*z + z*x + z*y + z^2
sage: FreeAlgebra(GF(5),3, 'xx, zba, Y')
Free Algebra on 3 generators (xx, zba, Y) over Finite Field of size 5
sage: FreeAlgebra(GF(5),3, 'abc')
Free Algebra on 3 generators (a, b, c) over Finite Field of size 5
sage: FreeAlgebra(GF(5),1, 'z')
Free Algebra on 1 generators (z,) over Finite Field of size 5
sage: FreeAlgebra(GF(5),1, ['alpha'])
Free Algebra on 1 generators (alpha,) over Finite Field of size 5
sage: FreeAlgebra(FreeAlgebra(ZZ,1,'a'), 2, 'x')
Free Algebra on 2 generators (x0, x1) over Free Algebra on 1 generators (a,) over Integer Ring
```

Free algebras are globally unique:

```
sage: F = FreeAlgebra(ZZ,3,'x,y,z')
sage: G = FreeAlgebra(ZZ,3,'x,y,z')
sage: F is G
True
```

Free algebras commute with their base ring.

```
sage: K.<a,b> = FreeAlgebra(QQ,2)
sage: K.is_commutative()
False
sage: L.<c> = FreeAlgebra(K,1)
sage: L.is_commutative()
False
sage: s = a*b^2 * c^3; s
a*b^2*c^3
sage: parent(s)
Free Algebra on 1 generators (c,) over Free Algebra on 2 generators (a, b) over Rational Field
sage: c^3 * a * b^2
a*b^2*c^3
```

class **FreeAlgebra_generic**(*R, n, names*)

The free algebra on n generators over a base ring.

EXAMPLES:

```
sage: F.<x,y,z> = FreeAlgebra(QQ, 3); F
Free Algebra on 3 generators (x, y, z) over Rational Field
sage: mul(F.gens())
x*y*z
```

```
sage: mul([ F.gen(i%3) for i in range(12) ])
x*y*z*x*y*z*x*y*z*x*y*z
sage: mul([ F.gen(i%3) for i in range(12) ]) + mul([ F.gen(i%2) for i in range(12) ])
x*y*x*y*x*y*x*y*x*y*x*y + x*y*z*x*y*z*x*y*z*x*y*z
sage: (2 + x*z + x^2)^2 + (x - y)^2
4 + 5*x^2 - x*y + 4*x*z - y*x + y^2 + x^4 + x^3*z + x*z*x^2 + x*z*x*z
```

coerce_map_from_impl(*R*)

gen(*i*)

The i-th generator of the algebra.

EXAMPLES:

```
sage: F = FreeAlgebra(ZZ,3,'x,y,z')
sage: F.gen(0)
x
```

is_commutative()

Return True if this free algebra is commutative.

EXAMPLES:

```
sage: R.<x> = FreeAlgebra(QQ,1)
sage: R.is_commutative()
True
sage: R.<x,y> = FreeAlgebra(QQ,2)
sage: R.is_commutative()
False
```

is_field()

Return True if this Free Algebra is a field, which is only if the base ring is a field and there are no generators

EXAMPLES:

```
sage: A=FreeAlgebra(QQ,0,'')
sage: A.is_field()
True
sage: A=FreeAlgebra(QQ,1,'x')
sage: A.is_field()
False
```

monoid()

The free monoid of generators of the algebra.

EXAMPLES:

```
sage: F = FreeAlgebra(ZZ,3,'x,y,z')
sage: F.monoid()
Free monoid on 3 generators (x, y, z)
```

ngens()

The number of generators of the algebra.

EXAMPLES:

```
sage: F = FreeAlgebra(ZZ,3,'x,y,z')
sage: F.ngens()
3
```

quo(*mons, mats, names*)

Returns a quotient algebra defined via the action of a free algebra A on a (finitely generated) free module. The input for the quotient algebra is a list of monomials (in the underlying monoid for A) which form a free basis for the module of A, and a list of matrices, which give the action of the free generators of A on this monomial basis.

quotient (*mons, mats, names*)
Returns a quotient algebra defined via the action of a free algebra A on a (finitely generated) free module. The input for the quotient algebra is a list of monomials (in the underlying monoid for A) which form a free basis for the module of A, and a list of matrices, which give the action of the free generators of A on this monomial basis.

is_FreeAlgebra (*x*)
Return True if x is a free algebra; otherwise, return False.

EXAMPLES:

```
sage: from sage.algebras.free_algebra import is_FreeAlgebra
sage: is_FreeAlgebra(5)
False
sage: is_FreeAlgebra(ZZ)
False
sage: is_FreeAlgebra(FreeAlgebra(ZZ,100,'x'))
True
```

30.2 Free algebra elements

AUTHORS:

- David Kohel (2005-09)

TESTS:

```
sage: R.<x,y> = FreeAlgebra(QQ,2)
sage: x == loads(dumps(x))
True
sage: x*y
x*y
sage: (x*y)^0
1
sage: (x*y)^3
x*y*x*y*x*y
```

class FreeAlgebraElement (*A, x*)
A free algebra element.

30.3 Free algebra quotients

TESTS:

```
sage: n = 2
sage: A = FreeAlgebra(QQ,n,'x')
sage: F = A.monoid()
sage: i, j = F.gens()
sage: mons = [ F(1), i, j, i*j ]
sage: r = len(mons)
sage: M = MatrixSpace(QQ,r)
sage: mats = [M([0,1,0,0, -1,0,0,0, 0,0,0,-1, 0,0,1,0]), M([0,0,1,0, 0,0,0,1, -1,0,0,0, 0,-1,0,0]) ]
sage: H2.<i,j> = A.quotient(mons,mats)
```

```
sage: H2 == loads(dumps(H2))
True
sage: i == loads(dumps(i))
True
```

class **FreeAlgebraQuotient**(*A, mons, mats, names*)

dimension()
The rank of the algebra (as a free module).

free_algebra()
The free algebra generating the algebra.

gen(*i*)
The i-th generator of the algebra.

matrix_action()

module()
The free module of the algebra.

monoid()
The free monoid of generators of the algebra.

monomial_basis()
The free monoid of generators of the algebra as elements of a free monoid.

ngens()
The number of generators of the algebra.

rank()
The rank of the algebra (as a free module).

30.4 Free algebra quotient elements

AUTHORS:

- David Kohel (2005-09)

class **FreeAlgebraQuotientElement**(*A, x*)

vector()

is_FreeAlgebraQuotientElement(*x*)

30.5 The Steenrod algebra

AUTHORS:

- John H. Palmieri (2008-07-30): version 0.9

This module defines the mod p Steenrod algebra $\mathcal{A}_p$, some of its properties, and ways to define elements of it.

From a topological point of view, $\mathcal{A}_p$ is the algebra of stable cohomology operations on mod p cohomology; thus for any topological space X, its mod p cohomology algebra $H^*(X, \mathbf{F}_p)$ is a module over $\mathcal{A}_p$.

From an algebraic point of view, $\mathcal{A}_p$ is an $\mathbf{F}_p$-algebra; when $p = 2$, it is generated by elements Sq^i for $i \geq 0$ (the *Steenrod squares*), and when p is odd, it is generated by elements $\mathcal{P}^i$ for $i \geq 0$ (the *Steenrod reduced 'p'th powers*) along with an element β (the *mod 'p' Bockstein*). The Steenrod algebra is graded: Sq^i is in degree i for each i, β is in degree 1, and $\mathcal{P}^i$ is in degree $2(p-1)i$.

The unit element is Sq^0 when $p = 2$ and $\mathcal{P}^0$ when p is odd. The generating elements also satisfy the *Adem relations*. At the prime 2, these have the form

$$\mathrm{Sq}^a \mathrm{Sq}^b = \sum_{c=0}^{[a/2]} \binom{b-c-1}{a-2c} \mathrm{Sq}^{a+b-c} \mathrm{Sq}^c.$$

At odd primes, they are a bit more complicated. See Steenrod and Epstein [SE] for full details. These relations lead to the existence of the *Serre-Cartan* basis for $\mathcal{A}_p$.

The mod p Steenrod algebra has the structure of a Hopf algebra, and Milnor [Mil] has a beautiful description of the dual, leading to a construction of the *Milnor basis* for $\mathcal{A}_p$. In this module, elements in the Steenrod algebra are represented, by default, using the Milnor basis.

See the documentation for `SteenrodAlgebra` for many more details and examples.

REFERENCES:

- [Mil] J. W. Milnor, "The Steenrod algebra and its dual," Ann. of Math. (2) 67 (1958), 150-171.
- [SE] N. E. Steenrod and D. B. A. Epstein, Cohomology operations, Ann. of Math. Stud. 50 (Princeton University Press, 1962).

class SteenrodAlgebraFactory()

The mod p Steenrod algebra

INPUT:

- `p` - positive prime integer (optional, default = 2)
- `basis` - string (optional, default = 'milnor')

OUTPUT:

- mod p Steenrod algebra with given basis

This returns the mod p Steenrod algebra, elements of which are printed using basis.

EXAMPLES:

Some properties of the Steenrod algebra are available:

```
sage: A = SteenrodAlgebra(2)
sage: A.ngens()  # number of generators
+Infinity
sage: A.gen(5)   # 5th generator
Sq(32)
sage: A.order()
+Infinity
sage: A.is_finite()
False
sage: A.is_commutative()
False
sage: A.is_noetherian()
False
sage: A.is_integral_domain()
False
```

```
sage: A.is_field()
False
sage: A.is_division_algebra()
False
sage: A.category()
Category of algebras over Finite Field of size 2
```

There are methods for constructing elements of the Steenrod algebra:

```
sage: A2 = SteenrodAlgebra(2); A2
mod 2 Steenrod algebra
sage: A2.Sq(1,2,6)
Sq(1,2,6)
sage: A2.Q(3,4)  # product of Milnor primitives Q_3 and Q_4
Sq(0,0,0,1,1)
sage: A2.pst(2,3)  # Margolis pst element
Sq(0,0,4)
sage: A5 = SteenrodAlgebra(5); A5
mod 5 Steenrod algebra
sage: A5.P(1,2,6)
P(1,2,6)
sage: A5.Q(3,4)
Q_3 Q_4
sage: A5.Q(3,4) * A5.P(1,2,6)
Q_3 Q_4 P(1,2,6)
sage: A5.pst(2,3)
P(0,0,25)
```

You can test whether elements are contained in the Steenrod algebra:

```
sage: w = Sq(2) * Sq(4)
sage: w in SteenrodAlgebra(2)
True
sage: w in SteenrodAlgebra(17)
False
```

Different bases for the Steenrod algebra:

There are two standard vector space bases for the mod p Steenrod algebra: the Milnor basis and the Serre-Cartan basis. When $p = 2$, there are also several other, less well-known, bases. See the documentation for the function 'steenrod_algebra_basis' for full descriptions of each of the implemented bases.

This module implements the following bases at all primes:

- 'milnor': Milnor basis.
- 'serre-cartan' or 'adem' or 'admissible': Serre-Cartan basis.

It implements the following bases when $p = 2$:

- 'wood_y': Wood's Y basis.
- 'wood_z': Wood's Z basis.
- 'wall', 'wall_long': Wall's basis.
- 'arnon_a', 'arnon_a_long': Arnon's A basis.
- 'arnon_c': Arnon's C basis.
- 'pst', 'pst_rlex', 'pst_llex', 'pst_deg', 'pst_revz': various P_t^s-bases.
- 'comm', 'comm_rlex', 'comm_llex', 'comm_deg', 'comm_revz', or these with '_long' appended: various commutator bases.

When defining a Steenrod algebra, you can specify a basis. Then elements of that Steenrod algebra are printed in that basis

```
sage: adem = SteenrodAlgebra(2, 'adem')
sage: x = adem.Sq(2,1)  # Sq(-) always means a Milnor basis element
sage: x
Sq^{4} Sq^{1} + Sq^{5}
sage: y = Sq(0,1)     # unadorned Sq defines elements w.r.t. Milnor basis
sage: y
Sq(0,1)
sage: adem(y)
Sq^{2} Sq^{1} + Sq^{3}
sage: adem5 = SteenrodAlgebra(5, 'serre-cartan')
sage: adem5.P(0,2)
P^{10} P^{2} + 4 P^{11} P^{1} + P^{12}
```

You can get a list of basis elements in a given dimension:

```
sage: A3 = SteenrodAlgebra(3, 'milnor')
sage: A3.basis(13)
(Q_1 P(2), Q_0 P(3))
```

As noted above, several of the bases ('arnon_a', 'wall', 'comm') have alternate, longer, representations. These provide ways of expressing elements of the Steenrod algebra in terms of the Sq^{2^n}.

```
sage: A_long = SteenrodAlgebra(2, 'arnon_a_long')
sage: A_long(Sq(6))
Sq^{1} Sq^{2} Sq^{1} Sq^{2} + Sq^{2} Sq^{4}
sage: SteenrodAlgebra(2, 'wall_long')(Sq(6))
Sq^{2} Sq^{1} Sq^{2} Sq^{1} + Sq^{2} Sq^{4}
sage: SteenrodAlgebra(2, 'comm_deg_long')(Sq(6))
s_{1} s_{2} s_{12} + s_{2} s_{4}
```

Testing unique parents:

```
sage: S0 = SteenrodAlgebra(2)
sage: S1 = SteenrodAlgebra(2)
sage: S0 is S1
True
```

create_key (*p=2, basis='milnor'*)

This is an internal function that is used to ensure unique parents. Not for public consumption.

EXAMPLES:

```
sage: SteenrodAlgebra.create_key()
(2, 'milnor')
```

create_object (*version, key, **extra_args*)

This is an internal function that is used to ensure unique parents. Not for public consumption.

EXAMPLES:

```
sage: SteenrodAlgebra.create_object(1,(11,'milnor'))
mod 11 Steenrod algebra
```

class **SteenrodAlgebra_generic** (*p=2, basis='milnor'*)

The mod p Steenrod algebra.

Users should not call this, but use the function 'SteenrodAlgebra' instead. See that function for extensive documentation.

EXAMPLES:

```
sage: sage.algebras.steenrod_algebra.SteenrodAlgebra_generic()
mod 2 Steenrod algebra
sage: sage.algebras.steenrod_algebra.SteenrodAlgebra_generic(5)
mod 5 Steenrod algebra
sage: sage.algebras.steenrod_algebra.SteenrodAlgebra_generic(5, 'adem')
mod 5 Steenrod algebra
```

P (**nums*)

The element $P(a, b, c, ...)$

INPUT:

- `a, b, c, ...` - non-negative integers

OUTPUT: element of the Steenrod algebra given by the single basis element P(a, b, c, ...)

Note that at the prime 2, this is the same element as Sq$(a, b, c, ...)$.

EXAMPLES:

```
sage: A = SteenrodAlgebra(2)
sage: A.P(5)
Sq(5)
sage: B = SteenrodAlgebra(3)
sage: B.P(5,1,1)
P(5,1,1)
```

Q (**nums*)

The element $Q_{n0}Q_{n1}...$, given by specifying the subscripts.

INPUT:

- `n0, n1, ...` - non-negative integers

OUTPUT: The element $Q_{n0}Q_{n1}...$

Note that at the prime 2, Q_n is the element Sq$(0, 0, ..., 1)$, where the 1 is in the $(n + 1)^{st}$ position.

Compare this to the method 'Q_exp', which defines a similar element, but by specifying the tuple of exponents.

EXAMPLES:

```
sage: A2 = SteenrodAlgebra(2)
sage: A5 = SteenrodAlgebra(5)
sage: A2.Q(2,3)
Sq(0,0,1,1)
sage: A5.Q(1,4)
Q_1 Q_4
sage: A5.Q(1,4) == A5.Q_exp(0,1,0,0,1)
True
```

Q_exp (**nums*)

The element $Q_0^{e_0}Q_1^{e_1}...$, given by specifying the exponents.

INPUT:

- `e0, e1, ...` - 0s and 1s

OUTPUT: The element $Q_0^{e_0}Q_1^{e_1}...$

Note that at the prime 2, Q_n is the element Sq$(0, 0, ..., 1)$, where the 1 is in the $(n + 1)^{st}$ position.

Compare this to the method 'Q', which defines a similar element, but by specifying the tuple of subscripts of terms with exponent 1.

EXAMPLES:

```
sage: A2 = SteenrodAlgebra(2)
sage: A5 = SteenrodAlgebra(5)
sage: A2.Q_exp(0,0,1,1,0)
Sq(0,0,1,1)
sage: A5.Q_exp(0,0,1,1,0)
Q_2 Q_3
sage: A5.Q(2,3)
Q_2 Q_3
sage: A5.Q_exp(0,0,1,1,0) == A5.Q(2,3)
True
```

basis(*n*)

Basis for self in dimension n

INPUT:

- n - non-negative integer

OUTPUT:

- basis - tuple of Steenrod algebra elements

EXAMPLES:

```
sage: A3 = SteenrodAlgebra(3)
sage: A3.basis(13)
(Q_1 P(2), Q_0 P(3))
sage: SteenrodAlgebra(2, 'adem').basis(12)
(Sq^{12},
Sq^{11} Sq^{1},
Sq^{9} Sq^{2} Sq^{1},
Sq^{8} Sq^{3} Sq^{1},
Sq^{10} Sq^{2},
Sq^{9} Sq^{3},
Sq^{8} Sq^{4})
```

category()

The Steenrod algebra is an algebra over F_p.

EXAMPLES:

```
sage: A = SteenrodAlgebra(3)
sage: A.category()
Category of algebras over Finite Field of size 3
```

gen(*i=0*)

The ith generator of the Steenrod algebra.

INPUT:

- i - non-negative integer

OUTPUT: the ith generator of the Steenrod algebra

The i^{th} generator is $Sq(2^i)$ at the prime 2; when p is odd, the 0th generator is $\beta = Q(0)$, and for $i > 0$, the i^{th} generator is $P(p^{i-1})$.

EXAMPLES:

```
sage: A = SteenrodAlgebra(2)
sage: A.gen(4)
Sq(16)
sage: A.gen(200)
Sq(1606938044258990275541962092341162602522202993782792835301376)
sage: B = SteenrodAlgebra(5)
sage: B.gen(0)
```

```
Q_0
sage: B.gen(2)
P(5)
```

gens()
List of generators for the Steenrod algebra. Not implemented (mainly because the list of generators is infinite).
EXAMPLES:

```
sage: A3 = SteenrodAlgebra(3, 'adem')
sage: A3.gens()
...
NotImplementedError: 'gens' is not implemented for the Steenrod algebra.
```

is_commutative()
The Steenrod algebra is not commutative.
EXAMPLES:

```
sage: A = SteenrodAlgebra(3)
sage: A.is_commutative()
False
```

is_division_algebra()
The Steenrod algebra is not a division algebra.
EXAMPLES:

```
sage: A = SteenrodAlgebra(3)
sage: A.is_division_algebra()
False
```

is_field()
The Steenrod algebra is not a field.
EXAMPLES:

```
sage: A = SteenrodAlgebra(3)
sage: A.is_field()
False
```

is_finite()
The Steenrod algebra is not finite.
EXAMPLES:

```
sage: A = SteenrodAlgebra(3)
sage: A.is_finite()
False
```

is_integral_domain()
The Steenrod algebra is not an integral domain.
EXAMPLES:

```
sage: A = SteenrodAlgebra(3)
sage: A.is_integral_domain()
False
```

is_noetherian()
The Steenrod algebra is not noetherian.
EXAMPLES:

```
sage: A = SteenrodAlgebra(3)
sage: A.is_noetherian()
False
```

ngens()

Number of generators of the Steenrod algebra.

This returns infinity, since the Steenrod algebra is infinitely generated.

EXAMPLES:

```
sage: A = SteenrodAlgebra(3)
sage: A.ngens()
+Infinity
```

order()

The Steenrod algebra has infinite order.

EXAMPLES:

```
sage: A = SteenrodAlgebra(3)
sage: A.order()
+Infinity
```

pst(*s, t*)

The Margolis element P_t^s.

INPUT:

- s - non-negative integer
- t - positive integer
- p - positive prime number

OUTPUT: element of the Steenrod algebra

This returns the Margolis element P_t^s of the mod p Steenrod algebra: the element equal to $P(0, 0, ..., 0, p^s)$, where the p^s is in position t.

EXAMPLES:

```
sage: A2 = SteenrodAlgebra(2)
sage: A2.pst(3,5)
Sq(0,0,0,0,8)
sage: A2.pst(1,2) == Sq(4)*Sq(2) + Sq(2)*Sq(4)
True
sage: SteenrodAlgebra(5).pst(3,5)
P(0,0,0,0,125)
```

class SteenrodAlgebra_mod_two(*p=2, basis='milnor'*)

The mod 2 Steenrod algebra.

Users should not call this, but use the function 'SteenrodAlgebra' instead. See that function for extensive documentation. (This differs from SteenrodAlgebra_generic only in that it has a method 'Sq' for defining elements.)

Sq(**nums*)

Milnor element $\mathrm{Sq}(a, b, c, ...)$.

INPUT:

- a, b, c, ... - non-negative integers

OUTPUT: element of the Steenrod algebra

This returns the Milnor basis element $\mathrm{Sq}(a, b, c, ...)$.

EXAMPLES:

```
sage: A = SteenrodAlgebra(2)
sage: A.Sq(5)
Sq(5)
sage: A.Sq(5,0,2)
Sq(5,0,2)
```

Entries must be non-negative integers; otherwise, an error results.

get_basis_name(*basis, p*)

Return canonical basis named by string basis at the prime p.

INPUT:

- basis - string
- p - positive prime number

OUTPUT:

- basis_name - string

EXAMPLES:

```
sage: sage.algebras.steenrod_algebra.get_basis_name('adem', 2)
'serre-cartan'
sage: sage.algebras.steenrod_algebra.get_basis_name('milnor', 2)
'milnor'
sage: sage.algebras.steenrod_algebra.get_basis_name('MiLNoR', 5)
'milnor'
sage: sage.algebras.steenrod_algebra.get_basis_name('pst-llex', 2)
'pst_llex'
```

30.6 Steenrod algebra elements

AUTHORS: - John H. Palmieri (2008-07-30: version 0.9)

This package provides for basic algebra with elements in the mod p Steenrod algebra. In this package, elements in the Steenrod algebra are represented, by default, using the Milnor basis.

EXAMPLES:

Basic arithmetic, $p = 2$. To construct an element of the mod 2 Steenrod algebra, use the function Sq:

```
sage: a = Sq(1,2)
sage: b = Sq(4,1)
sage: z = a + b
sage: z
Sq(1,2) + Sq(4,1)
sage: Sq(4) * Sq(1,2)
Sq(1,1,1) + Sq(2,3) + Sq(5,2)
sage: z**2          # non-negative exponents work as they should
Sq(1,2,1) + Sq(4,1,1)
sage: z**0
Sq(0)
```

Basic arithmetic, $p > 2$. To construct an element of the mod p Steenrod algebra when p is odd, you should first define a Steenrod algebra, using the SteenrodAlgebra command:

```
sage: SteenrodAlgebra()    # 2 is the default prime
mod 2 Steenrod algebra
sage: A3 = SteenrodAlgebra(3)
sage: A3
mod 3 Steenrod algebra
```

Having done this, the newly created algebra `A3` has methods `Q` and `P` which construct elements of `A3`:

```
sage: c = A3.Q(1,3,6); c
Q_1 Q_3 Q_6
sage: d = A3.P(2,0,1); d
P(2,0,1)
sage: c * d
Q_1 Q_3 Q_6 P(2,0,1)
sage: e = A3.P(3)
sage: d * e
P(5,0,1)
sage: e * d
P(1,1,1) + P(5,0,1)
sage: c * c
0
sage: e ** 3
2 P(1,2)
```

Note that one can construct an element like `c` above in one step, without first constructing the algebra:

```
sage: c = SteenrodAlgebra(3).Q(1,3,6)
sage: c
Q_1 Q_3 Q_6
```

And of course, you can do similar constructions with the mod 2 Steenrod algebra:

```
sage: A = SteenrodAlgebra(2); A
mod 2 Steenrod algebra
sage: A.Sq(2,3,5)
Sq(2,3,5)
sage: A.P(2,3,5)    # when p=2, P = Sq
Sq(2,3,5)
sage: A.Q(1,4)      # when p=2, this gives a product of Milnor primitives
Sq(0,1,0,0,1)
```

Regardless of the prime, each element has an `excess`, and if the element is homogeneous, a `degree`. The excess of $\mathrm{Sq}(i_1, i_2, i_3, ...)$ is $i_1 + i_2 + i_3 + ...$; when p is odd, the excess of $Q_0^{e_0} Q_1^{e_1} ... \mathcal{P}(r_1, r_2, ...)$ is $\sum e_i + 2 \sum r_i$. The excess of a linear combination of Milnor basis elements is the minimum of the excesses of those basis elements.

The degree of $\mathrm{Sq}(i_1, i_2, i_3, ...)$ is $sum(2^n - 1) i_n$, and when p is odd, the degree of $Q_0^{\epsilon_0} Q_1^{\epsilon_1} ... \mathcal{P}(r_1, r_2, ...)$ is $\sum \epsilon_i (2p^i - 1) + \sum r_j (2p^j - 2)$. The degree of a linear combination of such terms is only defined if the terms all have the same degree.

Here are some simple examples:

```
sage: z = Sq(1,2) + Sq(4,1)
sage: z.degree()
7
sage: (Sq(0,0,1) + Sq(5,3)).degree()
Element is not homogeneous.
```

```
sage: Sq(7,2,1).excess()
10
sage: z.excess()
3
sage: B = SteenrodAlgebra(3)
sage: x = B.Q(1,4)
sage: y = B.P(1,2,3)
sage: x.degree()
166
sage: x.excess()
2
sage: y.excess()
12
```

Elements have a `weight` in the May filtration, which (when $p = 2$) is related to the `height` function defined by Wall:

```
sage: Sq(2,1,5).may_weight()
9
sage: Sq(2,1,5).wall_height()
[2, 3, 2, 1, 1]
sage: b = Sq(4)*Sq(8) + Sq(8)*Sq(4)
sage: b.may_weight()
2
sage: b.wall_height()
[0, 0, 1, 1]
```

Odd primary May weights:

```
sage: A5 = SteenrodAlgebra(5)
sage: a = A5.Q(1,2,4)
sage: b = A5.P(1,2,1)
sage: a.may_weight()
10
sage: b.may_weight()
8
sage: (a * b).may_weight()
18
sage: A5.P(0,0,1).may_weight()
3
```

Since the Steenrod algebra is a Hopf algebra, every element has an antipode.

```
sage: d = Sq(0,0,1); d
Sq(0,0,1)
sage: d.antipode()
Sq(0,0,1)
sage: Sq(4).antipode()
Sq(1,1) + Sq(4)
sage: (Sq(4) * Sq(2)).antipode()
Sq(6)
sage: SteenrodAlgebra(7).P(3,1).antipode()
P(3,1)
```

Applying the antipode twice returns the original element:

```
sage: y = Sq(8)*Sq(4)
sage: y == (y.antipode()).antipode()
True
```

You can treat elements of the Steenrod algebra like lists of Milnor basis elements:

```
sage: y = Sq(4) * Sq(1,2); y
Sq(1,1,1) + Sq(2,3) + Sq(5,2)
sage: for m in y: m
Sq(1,1,1)
Sq(2,3)
Sq(5,2)
sage: [(m.degree(),m.excess()) for m in y]
[(11, 3), (11, 5), (11, 7)]
```

Once you've define a Steenrod algebra, the method `pst` is another way to define elements of it: `pst(s,t)` defines the Margolis element $P(s,t)$, the basis element $\mathcal{P}(0,...,0,p^s)$ with p^s in position t:

```
sage: A2 = SteenrodAlgebra(2)
sage: Q2 = A2.pst(0,3)
sage: Q2
Sq(0,0,1)
sage: Q2*Q2
0
sage: A2.pst(1,2) == Sq(2)*Sq(4) + Sq(4)*Sq(2)
True
sage: A5 = SteenrodAlgebra(5)
sage: A5.pst(2,2)
P(0,25)
```

There are a number of different bases available in which to represent elements of the Steenrod algebra. When $p > 2$, the choices are the Milnor basis ('milnor') or the Serre-Cartan basis ('serre-cartan' or 'adem' or 'admissible'). When $p = 2$, the choices are those, along with Wood's Y basis ('wood_y'), Wood's Z basis ('wood_z'), Wall's basis ('wall' or 'wall_long'), Arnon's A basis ('arnon_a' or 'arnon_a_long'), Arnon's C basis ('arnon_c'), various $P(s,t)$ bases ('pst_ORDER' for various values of ORDER), and various commutator bases ('comm_ORDER' or 'comm_ORDER_long' for various values of ORDER).

See documentation for the function `steenrod_algebra_basis` for full descriptions of these bases.

To access representations of elements with respect to these different bases, you can either use the `basis` method for an element, or define a Steenrod algebra with respect to a particular basis and then use that:

```
sage: c = Sq(2) * Sq(1); c
Sq(0,1) + Sq(3)
sage: c.basis('serre-cartan')
Sq^{2} Sq^{1}
sage: c.basis('milnor')
Sq(0,1) + Sq(3)
sage: adem = SteenrodAlgebra(2, 'serre-cartan')
sage: x = Sq(7,3,1)    # top class in the subalagebra A(2)
sage: adem(x)
Sq^{17} Sq^{5} Sq^{1}
sage: SteenrodAlgebra(2, 'pst')(x)
P^{0}_{1} P^{0}_{2} P^{1}_{1} P^{0}_{3} P^{1}_{2} P^{2}_{1}
```

Multiplication works within bases:

```
sage: adem = SteenrodAlgebra(2, 'adem')
sage: x = adem.Sq(5)
sage: y = adem.Sq(1)
sage: x * y
Sq^{5} Sq^{1}
```

When multiplying elements defined with respect to different bases, the result is printed in the basis of the left-hand factor:

```
sage: milnor = SteenrodAlgebra(2, 'milnor')
sage: xm = milnor.Sq(5)
sage: ym = milnor.Sq(1)
sage: xm * ym
Sq(3,1)
sage: xm * y
Sq(3,1)
sage: x * ym
Sq^{5} Sq^{1}
```

Several of these bases ('arnon_a', 'wall', 'comm') have alternate, longer, representations. These provide ways of expressing elements of the Steenrod algebra in terms of the Sq^{2^n}.

```
sage: Sq(6).basis('arnon_a_long')
Sq^{1} Sq^{2} Sq^{1} Sq^{2} + Sq^{2} Sq^{4}
sage: Sq(6).basis('wall_long')
Sq^{2} Sq^{1} Sq^{2} Sq^{1} + Sq^{2} Sq^{4}
sage: SteenrodAlgebra(2,'comm_deg_long')(Sq(6))
s_{1} s_{2} s_{12} + s_{2} s_{4}
```

INTERNAL DOCUMENTATION:

Here are details on the class `SteenrodAlgebraElement` (for people who want to delve into or extend the code):

Attributes for a `SteenrodAlgebraElement` self:

- `self._base_field`: $GF(p)$, where p is the associated prime
- `self._prime`: p
- `self._basis`: basis in which to print this element
- `self._raw`: dictionary. keys are basis names, taken from `_steenrod_basis_unique_names`, and the associated values are dictionaries themselves; if the dictionary is nonempty, it gives the representation for that element in the given basis. If it is empty, that means that the representation in that basis hasn't been computed yet. The representation of an element with respect to a basis (other than the Milnor basis, which is how elements are stored internally) isn't computed until requested, either by calling the method `_basis_dictionary('basis_name')`, or more typically, by calling the method `basis('basis_name')` or by defining a Steenrod algebra at that basis and applying its call method to the element.

The dictionaries are defined as follows. In the Milnor basis at the prime 2, for example, since monomials are of the form $\mathrm{Sq}(a, b, c, ...)$, then monomials are stored as tuples of integers `(a,b,c,...)`. Thus if

$$y = \mathrm{Sq}(5, 3) + \mathrm{Sq}(0, 0, 2),$$

then `y._raw['milnor']` is `{(0, 0, 2): 1, (5, 3): 1}`. (The 1's following the colons are the coefficients of the monomials associated to the tuples.) Each basis has its own representation as a dictionary; Arnon's C

basis represents basis elements as tuples of integers, just like the Milnor basis and the Serre-Cartan basis, while the other bases represent basis elements as tuples of pairs of integers. From the descriptions of the bases given in the file 'steenrod_algebra_bases.py', it should be clear how to associate a tuple of pairs of integers to a basis element. See also the function `string_rep`.

When the element is initially defined by calling `Sq` or `SteenrodAlgebraElement`, typically only the 'milnor' dictionary is non-empty, while if the element is defined by the function `steenrod_algebra_basis`, its dictionary for the given basis is also initialized correctly. For example:

```
sage: B = steenrod_algebra_basis(6,'adem'); B
(Sq^{6}, Sq^{5} Sq^{1}, Sq^{4} Sq^{2})
sage: x = B[1]; x
Sq^{5} Sq^{1}
sage: x._raw
{'milnor': {(3, 1): 1}, 'serre-cartan': {(5, 1): 1}}
```

Note that the keys 'milnor' and 'serre-cartan' (a synonym for 'adem') have nonempty associated values.

When any element is converted to another basis (by changing the basis and then printing the element), its dictionary for that basis gets stored, as well:

```
sage: x.basis('arnona')
X^{0}_{0} X^{1}_{0} X^{1}_{1}
sage: x._raw
{'arnona': {((0, 0), (1, 0), (1, 1)): 1},
'milnor': {(3, 1): 1},
'serre-cartan': {(5, 1): 1}}
```

Methods for a `SteenrodAlgebraElement` self:

Most of these are self-explanatory.

- `_mul_`: multiply two elements. This is done using Milnor multiplication, the code for which is in a separate file, 'steenrod_milnor_multiplication'. In a long computation, it seems that a lot of time is spent here, so one way to speed things up would be to optimize the Milnor multiplication routine.
- `_basis_dictionary`: compute the dictionary of the element with respect to the given basis. This is basically done by doing a basis conversion from the Milnor basis to the given basis. There are two parts to this function; first, some elements (e.g., $\mathrm{Sq}(2^n)$) may be easy to convert directly. This is done one basis at a time, and so takes up most of the lines of code. If the element is not recognizable as being easy to convert, then the function `milnor_convert` from the file 'steenrod_algebra_bases.py' is called. This does linear algebra: it computes the Milnor basis and the new basis in the appropriate dimension, computes the change-of-basis matrix, etc.
- `basis`: display the element in the given basis.
- `milnor`: display the element in the Milnor basis.
- `serre_cartan`: display the element in the Serre-Cartan basis.
- `adem`: display the element in the Serre-Cartan basis.
- `_repr_` and `_latex_` call the function `string_rep`, which has cases depending on the basis.

REFERENCES:

- [Mil] J. W. Milnor, "The Steenrod algebra and its dual," Ann. of Math. (2) 67 (1958), 150-171.

- [Mon] K. G. Monks, "Change of basis, monomial relations, and P_t^s bases for the Steenrod algebra," J. Pure Appl. Algebra 125 (1998), no. 1-3, 235-260.
- [Woo] R. M. W. Wood, "Problems in the Steenrod algebra," Bull. London Math. Soc. 30 (1998), no. 5, 449-517.

Sq(**nums*)

Milnor element Sq(a,b,c,...).

INPUT:

- `a, b, c, ...` - non-negative integers

OUTPUT: element of the Steenrod algebra

This returns the Milnor basis element ' ext{Sq}(a, b, c, ...)'.

EXAMPLES:

```
sage: Sq(5)
Sq(5)
sage: Sq(5) + Sq(2,1) + Sq(5)  # addition is mod 2:
Sq(2,1)
sage: (Sq(4,3) + Sq(7,2)).degree()
13
```

Entries must be non-negative integers; otherwise, an error results.

This function is a good way to define elements of the Steenrod algebra.

class SteenrodAlgebraElement(*poly, p=2, basis='milnor'*)

Element of the mod p Steenrod algebra.

At the prime 2, use the function 'Sq' to define these, as in 'w=Sq(4,3,3)' or 'z=Sq(1,2)+Sq(4,1)' or 'q=Sq(8)*Sq(4) + Sq(12)'.

At odd primes, use the methods 'P' and 'Q' to define these, as in 'w=SteenrodAlgebra(3).Q(1,5) * SteenrodAlgebra(3).P(4,3)'.

EXAMPLES:

```
sage: w = Sq(4,3,3)
sage: w
Sq(4,3,3)
```

The function 'Sq', together with addition, provides an easy way to define elements when $p = 2$:

```
sage: b = Sq(3) + Sq(0,1)
sage: b
Sq(0,1) + Sq(3)
```

When p is odd, first define a Steenrod algebra to specify the prime, and then use the methods 'P' and 'Q', together with multiplication and addition:

```
sage: A7 = SteenrodAlgebra(7)
sage: u = A7.Q(0,4); u
Q_0 Q_4
sage: v = A7.P(1,2,3); v
P(1,2,3)
sage: u * v
Q_0 Q_4 P(1,2,3)
sage: 10 * u * v
3 Q_0 Q_4 P(1,2,3)
sage: u + v
P(1,2,3) + Q_0 Q_4
```

additive_order()

The additive order of any nonzero element of the mod p Steenrod algebra is p.

OUTPUT:

- `order` - positive prime number

EXAMPLES:

```
sage: z = Sq(4) + Sq(6) + Sq(0)
sage: z.additive_order()
2
sage: (Sq(3) + Sq(3)).additive_order()
1
```

adem()

Serre-Cartan representation of self.

OUTPUT: Serre-Cartan representation of self.

EXAMPLES:

```
sage: x = Sq(0,1); x
Sq(0,1)
sage: x.serre_cartan()
Sq^{2} Sq^{1} + Sq^{3}
sage: x.adem()  # 'adem' is a synonym for 'serre_cartan'
Sq^{2} Sq^{1} + Sq^{3}
```

antipode()

Antipode of element.

OUTPUT:

- `antipode` - element of the Steenrod algebra

Algorithm: according to a result of Milnor's, the antipode of $\mathrm{Sq}(n)$ is the sum of all of the Milnor basis elements in dimension n. So: convert the element to the Serre-Cartan basis and use this formula for the antipode of $\mathrm{Sq}(n)$, together with the fact that the antipode is an antihomomorphism: if we call the antipode c, then $c(ab) = c(b)c(a)$.

At odd primes, a similar method is used: the antipode of $P(n)$ is the sum of the Milnor P basis elements in dimension $n * 2(p-1)$, and the antipode of $\beta = Q_0$ is $-Q_0$. So convert to the Serre-Cartan basis, as in the $p = 2$ case.

EXAMPLES:

```
sage: d = Sq(0,0,1); d
Sq(0,0,1)
sage: d.antipode()
Sq(0,0,1)
sage: Sq(4).antipode()
Sq(1,1) + Sq(4)
sage: (Sq(4) * Sq(2)).antipode()
Sq(6)
sage: A3 = SteenrodAlgebra(3)
sage: A3.P(2).antipode()
P(2)
sage: A3.P(2,1).antipode()
2 P(2,1)
```

Applying the antipode twice returns the original element:

```
sage: y = Sq(8)*Sq(4)
sage: y == (y.antipode()).antipode()
True
sage: z = A3.P(1,1)
```

```
sage: z == (z.antipode()).antipode()
True
sage: w = A3.Q(5)
sage: w == (w.antipode()).antipode()
True
sage: a = SteenrodAlgebra(7).P(55)
sage: a == a.antipode().antipode()
True
```

TESTS:

```
sage: all(a.antipode().antipode() == a for a in steenrod_algebra_basis(201, basis='adem', p=
True
sage: all(a.antipode().antipode() == a for a in steenrod_algebra_basis(100, basis='milnor',
True
sage: all(a.antipode().antipode() == a for a in steenrod_algebra_basis(30, basis='milnor', p
True
sage: all(a.antipode().antipode() == a for a in steenrod_algebra_basis(33, basis='adem', p=2
True
```

basis (*basis*)

Representation of element with respect to basis.

INPUT:

- `basis` - string, basis in which to work.

OUTPUT: Representation of self in given basis

The choices for basis are:

- 'milnor' for the Milnor basis.
- 'serre-cartan', 'serre_cartan', 'sc', 'adem', 'admissible' for the Serre-Cartan basis.
- 'wood_y' for Wood's Y basis.
- 'wood_z' for Wood's Z basis.
- 'wall' for Wall's basis.
- 'wall_long' for Wall's basis, alternate representation
- 'arnon_a' for Arnon's A basis.
- 'arnon_a_long' for Arnon's A basis, alternate representation.
- 'arnon_c' for Arnon's C basis.
- 'pst', 'pst_rlex', 'pst_llex', 'pst_deg', 'pst_revz' for various P_t^s-bases.
- 'comm', 'comm_rlex', 'comm_llex', 'comm_deg', 'comm_revz' for various commutator bases.
- 'comm_long', 'comm_rlex_long', etc., for commutator bases, alternate representations.

See documentation for the function 'steenrod_algebra_basis' for descriptions of the different bases.

EXAMPLES:

```
sage: c = Sq(2) * Sq(1)
sage: c.basis('milnor')
Sq(0,1) + Sq(3)
sage: c.basis('serre-cartan')
Sq^{2} Sq^{1}
sage: d = Sq(0,0,1)
sage: d.basis('arnonc')
Sq^{2} Sq^{5} + Sq^{4} Sq^{2} Sq^{1} + Sq^{4} Sq^{3} + Sq^{7}
```

degree ()

Degree of element.

OUTPUT:

- `degree` - None, or non-negative integer

The degree of $\mathrm{Sq}(i_1, i_2, i_3, ...)$ is $i_1 + 3i_2 + 7i_3 + ... + (2^n - 1)i_n + ...$ When p is odd, the degree of $Q_0^{e_0} Q_1^{e_1} ... P(r_1, r_2, ...)$ is $\sum e_i(2p^i - 1) + \sum r_j(2p^j - 2)$.

The degree of a sum is undefined (and this returns 'None'), unless each summand has the same degree: that is, unless the element is homogeneous.

EXAMPLES:

```
sage: a = Sq(1,2,1)
sage: a.degree()
14
sage: for a in Sq(3) + Sq(5,1): a.degree()
3
8
sage: (Sq(3) + Sq(5,1)).degree()
Element is not homogeneous.
sage: B = SteenrodAlgebra(3)
sage: x = B.Q(1,4)
sage: y = B.P(1,2,3)
sage: x.degree()
166
sage: y.degree()
192
```

excess()

Excess of element.

OUTPUT:

- `excess` - non-negative integer

The excess of $\mathrm{Sq}(a, b, c, ...)$ is $a + b + c + ...$ When p is odd, the excess of $Q_0^{e_0} Q_1^{e_1} ... P(r_1, r_2, ...)$ is $\sum e_i + 2\sum r_i$.

The excess of a linear combination of Milnor basis elements is the minimum of the excesses of those basis elements.

See [Kra] for the proofs of these assertions.

EXAMPLES:

```
sage: a = Sq(1,2,3)
sage: a.excess()
6
sage: (Sq(0,0,1) + Sq(4,1) + Sq(7)).excess()
1
sage: [m.excess() for m in (Sq(0,0,1) + Sq(4,1) + Sq(7))]
[1, 5, 7]
sage: [m for m in (Sq(0,0,1) + Sq(4,1) + Sq(7))]
[Sq(0,0,1), Sq(4,1), Sq(7)]
sage: B = SteenrodAlgebra(7)
sage: a = B.Q(1,2,5)
sage: b = B.P(2,2,3)
sage: a.excess()
3
sage: b.excess()
14
sage: (a + b).excess()
3
sage: (a * b).excess()
17
```

REFERENCES:

- [Kra] D. Kraines, "On excess in the Milnor basis," Bull. London Math. Soc. 3 (1971), 363-365.

is_decomposable()

return True if element is decomposable, False otherwise.

OUTPUT:

- `decomposable` - boolean

That is, if element is in the square of the augmentation ideal, return True; otherwise, return False.

EXAMPLES:

```
sage: a = Sq(6)
sage: a.is_decomposable()
True
sage: for i in range(9):
...       if not Sq(i).is_decomposable():
...           print Sq(i)
Sq(0)
Sq(1)
Sq(2)
Sq(4)
Sq(8)
```

is_nilpotent()

True if element is not a unit, False otherwise.

EXAMPLES:

```
sage: z = Sq(4,2) + Sq(7,1) + Sq(3,0,1)
sage: z.is_nilpotent()
True
sage: u = 1 + Sq(3,1)
sage: u == Sq(0) + Sq(3,1)
True
sage: u.is_nilpotent()
False
```

is_unit()

True if element has a nonzero scalar multiple of P(0) as a summand, False otherwise.

EXAMPLES:

```
sage: z = Sq(4,2) + Sq(7,1) + Sq(3,0,1)
sage: z.is_unit()
False
sage: u = 1 + Sq(3,1)
sage: u == Sq(0) + Sq(3,1)
True
sage: u.is_unit()
True
sage: A5 = SteenrodAlgebra(5)
sage: v = A5.P(0)
sage: (v + v + v).is_unit()
True
```

may_weight()

May's 'weight' of element.

OUTPUT:

- `weight` - non-negative integer

If we let $F_*(A)$ be the May filtration of the Steenrod algebra, the weight of an element x is the integer k so that x is in $F_k(A)$ and not in $F_{k+1}(A)$. According to Theorem 2.6 in May's thesis [May], the weight of a Milnor basis element is computed as follows: first, to compute the weight of $P(r_1, r_2, ...)$, write each r_i in base p as $r_i = \sum_j p^j r_{ij}$. Then each nonzero binary digit r_{ij} contributes i to the weight: the weight

is $\sum_{i,j} ir_{ij}$. When p is odd, the weight of Q_i is $i+1$, so the weight of a product $Q_{i_1}Q_{i_2}...$ is equal $(i_1+1)+(i_2+1)+....$ Then the weight of $Q_{i_1}Q_{i_2}...P(r_1,r_2,...)$ is the sum of $(i_1+1)+(i_2+1)+...$ and $\sum_{i,j} ir_{ij}$.

The weight of a sum of basis elements is the minimum of the weights of the summands.

When $p=2$, we compute the weight on Milnor basis elements by adding up the terms in their 'height' - see the method 'wall_height' for documentation. (When p is odd, the height of an element is not defined.)

EXAMPLES:

```
sage: Sq(0).may_weight()
0
sage: a = Sq(4)
sage: a.may_weight()
1
sage: b = Sq(4)*Sq(8) + Sq(8)*Sq(4)
sage: b.may_weight()
2
sage: Sq(2,1,5).wall_height()
[2, 3, 2, 1, 1]
sage: Sq(2,1,5).may_weight()
9
sage: A5 = SteenrodAlgebra(5)
sage: a = A5.Q(1,2,4)
sage: b = A5.P(1,2,1)
sage: a.may_weight()
10
sage: b.may_weight()
8
sage: (a * b).may_weight()
18
sage: A5.P(0,0,1).may_weight()
3
```

REFERENCES:

- [May]: J. P. May, "The cohomology of restricted Lie algebras and of Hopf algebras; application to the Steenrod algebra." Thesis, Princeton Univ., 1964.

milnor()

Milnor representation of self.

OUTPUT: Milnor representation of self.

EXAMPLES:

```
sage: A = SteenrodAlgebra(2, 'adem')
sage: x = A (Sq(5) * Sq(2) * Sq(1)); x
Sq^{5} Sq^{2} Sq^{1}
sage: x.milnor()
Sq(1,0,1) + Sq(5,1)
```

serre_cartan()

Serre-Cartan representation of self.

OUTPUT: Serre-Cartan representation of self.

EXAMPLES:

```
sage: x = Sq(0,1); x
Sq(0,1)
sage: x.serre_cartan()
Sq^{2} Sq^{1} + Sq^{3}
sage: x.adem()  # 'adem' is a synonym for 'serre_cartan'
Sq^{2} Sq^{1} + Sq^{3}
```

wall_height()

Wall's 'height' of element.

OUTPUT:

- height - list of non-negative integers

The height of an element of the mod 2 Steenrod algebra is a list of non-negative integers, defined as follows: if the element is a monomial in the generators $\mathrm{Sq}(2^i)$, then the i^{th} entry in the list is the number of times $\mathrm{Sq}(2^i)$ appears. For an arbitrary element, write it as a sum of such monomials; then its height is the maximum, ordered right-lexicographically, of the heights of those monomials.

When p is odd, the height of an element is not defined.

According to Theorem 3 in [Wall], the height of the Milnor basis element $\mathrm{Sq}(r_1, r_2, ...)$ is obtained as follows: write each r_i in binary as $r_i = \sum_j 2^j r_{ij}$. Then each nonzero binary digit r_{ij} contributes 1 to the k^{th} entry in the height, for $j \leq k \leq i + j - 1$.

EXAMPLES:

```
sage: Sq(0).wall_height()
[]
sage: a = Sq(4)
sage: a.wall_height()
[0, 0, 1]
sage: b = Sq(4)*Sq(8) + Sq(8)*Sq(4)
sage: b.wall_height()
[0, 0, 1, 1]
sage: Sq(0,0,3).wall_height()
[1, 2, 2, 1]
```

REFERENCES:

- [Wall]: C. T. C. Wall, "Generators and relations for the Steenrod algebra," Ann. of Math. (2) **72** (1960), 429-444.

adem(*x*)

Serre-Cartan representation of x.

INPUT:

- x - element of the Steenrod algebra

OUTPUT: Serre-Cartan representation of x

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import serre_cartan
sage: x = Sq(3,2); x
Sq(3,2)
sage: serre_cartan(x)
Sq^{7} Sq^{2}
```

admissible(*x*)

Serre-Cartan representation of x.

INPUT:

- x - element of the Steenrod algebra

OUTPUT: Serre-Cartan representation of x

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import serre_cartan
sage: x = Sq(3,2); x
Sq(3,2)
sage: serre_cartan(x)
Sq^{7} Sq^{2}
```

arnonA_long_mono_to_string(*mono, latex=False, p=2*)

Alternate string representation of element of Arnon's A basis.

This is used by the _repr_ and _latex_ methods.

INPUT:

- `mono` - tuple of pairs of non-negative integers (m,k) with $m >= k$

OUTPUT:

- `string` - concatenation of strings of the form '$Sq(2^m)$'

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import arnonA_long_mono_to_string
sage: arnonA_long_mono_to_string(((1,2),(3,0)))
'Sq^{8} Sq^{4} Sq^{2} Sq^{1}'
sage: arnonA_long_mono_to_string(((1,2),(3,0)),latex=True)
'\text{Sq}^{8} \text{Sq}^{4} \text{Sq}^{2} \text{Sq}^{1}'
```

The empty tuple represents the unit element Sq(0):

```
sage: arnonA_long_mono_to_string(())
'Sq(0)'
```

arnonA_mono_to_string(*mono, latex=False, p=2*)

String representation of element of Arnon's A basis.

This is used by the _repr_ and _latex_ methods.

INPUT:

- `mono` - tuple of pairs of non-negative integers (m,k) with $m >= k$

OUTPUT:

- `string` - concatenation of 'X_k^m' for each pair (m,k)

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import arnonA_mono_to_string
sage: arnonA_mono_to_string(((1,2),(3,0)))
'X^{1}_{2} X^{3}_{0}'
sage: arnonA_mono_to_string(((1,2),(3,0)),latex=True)
'X^{1}_{2} X^{3}_{0}'
```

The empty tuple represents the unit element Sq(0):

```
sage: arnonA_mono_to_string(())
'Sq(0)'
```

base_p_expansion(*n, p*)

Return list of digits in the base p expansion of n.

INPUT:

- n - non-negative integer
- p - positive prime number

OUTPUT: list of digits in the base p expansion of n

EXAMPLES:

```
sage: sage.algebras.steenrod_algebra_element.base_p_expansion(10,2)
[0, 1, 0, 1]
sage: sage.algebras.steenrod_algebra_element.base_p_expansion(10,3)
[1, 0, 1]
sage: sage.algebras.steenrod_algebra_element.base_p_expansion(10,5)
[0, 2]
sage: sage.algebras.steenrod_algebra_element.base_p_expansion(10,7)
[3, 1]
sage: sage.algebras.steenrod_algebra_element.base_p_expansion(0,7)
[]
sage: sage.algebras.steenrod_algebra_element.base_p_expansion(100000,13)
[4, 9, 6, 6, 3]
```

check_and_trim(*nums*)

Check that list or tuple consists of non-negative integers, and strip trailing zeroes.

INPUT:

- nums - a list or tuple

OUTPUT:

- new - a list or tuple

If nums contains anything other than a non-negative integer, raise an exception, identifying the right-most problematic entry. Otherwise, return a new list or tuple, obtained from nums by omitting any zeroes from the end.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import check_and_trim
sage: check_and_trim([3,4,1])
[3, 4, 1]
sage: a=[3,2,1,0,0]
sage: check_and_trim(a)
[3, 2, 1]
sage: a    # check_and_trim doesn't affect its input
[3, 2, 1, 0, 0]
sage: check_and_trim([0]*127)
[]
sage: check_and_trim((1,2,3,4,0,0,0))  # works on tuples, too
(1, 2, 3, 4)
```

comm_long_mono_to_string(*mono, latex=False, p=2*)

Alternate string representation of element of a commutator basis.

Okay in low dimensions, but gets unwieldy as the dimension increases.

INPUT:

- mono - tuple of pairs of integers (s,t) with $s >= 0$, $t > 0$

OUTPUT:

- string - concatenation of $s_{2^s \ldots 2^{(s+t-1)}}$ for each pair (s,t)

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import comm_long_mono_to_string
sage: comm_long_mono_to_string(((1,2),(0,3)))
's_{24} s_{124}'
sage: comm_long_mono_to_string(((1,2),(0,3)),latex=True)
's_{24} s_{124}'
```

The empty tuple represents the unit element Sq(0):

```
sage: comm_long_mono_to_string(())
'Sq(0)'
```

comm_mono_to_string(*mono, latex=False, p=2*)

String representation of element of a commutator basis.

This is used by the _repr_ and _latex_ methods.

INPUT:

- mono - tuple of pairs of integers (s,t) with $s >= 0$, $t > 0$

OUTPUT:

- string - concatenation of '$c_{s,t}$' for each pair (s,t)

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import comm_mono_to_string
sage: comm_mono_to_string(((1,2),(0,3)))
'c_{1,2} c_{0,3}'
sage: comm_mono_to_string(((1,2),(0,3)),latex=True)
'c_{1,2} c_{0,3}'
```

The empty tuple represents the unit element Sq(0):

```
sage: comm_mono_to_string(())
'Sq(0)'
```

convert_perm(*m*)

Convert tuple m of non-negative integers to a permutation in one-line form.

INPUT:

- m - tuple of non-negative integers with no repetitions

OUTPUT:

- list - conversion of m to a permutation of the set 1,2,...,len(m)

If m=(3,7,4), then one can view m as representing the permutation of the set 3,4,7 sending 3 to 3, 4 to 7, and 7 to 4. This function converts m to the list [1,3,2], which represents essentially the same permutation, but of the set 1,2,3. This list can then be passed to Permutation, and its signature can be computed.

EXAMPLES:

```
sage: sage.algebras.steenrod_algebra_element.convert_perm((3,7,4))
[1, 3, 2]
sage: sage.algebras.steenrod_algebra_element.convert_perm((5,0,6,3))
[2, 4, 1, 3]
```

degree(x)

Degree of x.

INPUT:

- x - element of the Steenrod algebra

OUTPUT:

- degree - non-negative integer or None

The degree of $\mathrm{Sq}(i_1, i_2, i_3, ...)$ is $i_1 + 3i_2 + 7i_3 + ... + (2^n - 1)i_n + ...$ When p is odd, the degree of $Q_0^{e_0} Q_1^{e_1} ... P(r_1, r_2, ...)$ is $\sum e_i(2p^i - 1) + \sum r_j(2p^j - 2)$.

The degree of a sum is undefined (and this function returns None), unless each summand has the same degree: that is, unless the element is homogeneous.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import degree
sage: a = Sq(1,2,1)
sage: degree(a)
14
sage: degree(Sq(3) + Sq(5,1))
Element is not homogeneous.
sage: B = SteenrodAlgebra(3)
sage: x = B.Q(1,4)
sage: y = B.P(1,2,3)
sage: degree(x)
166
sage: degree(y)
192
```

excess(x)

Excess of x.

INPUT:

- x - element of the Steenrod algebra

OUTPUT:

- excess - non-negative integer

The excess of $\mathrm{Sq}(a, b, c, ...)$ is $a + b + c + ...$. When p is odd, the excess of $Q_0^{e_0} Q_1^{e_1} \dots P(r_1, r_2, \dots)$ is $\sum e_i + 2\sum r_i$.

The excess of a linear combination of Milnor basis elements is the minimum of the excesses of those basis elements.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import excess
sage: a = Sq(1,2,3)
sage: excess(a)
6
sage: excess(Sq(0,0,1) + Sq(4,1) + Sq(7))
1
sage: [excess(m) for m in (Sq(0,0,1) + Sq(4,1) + Sq(7))]
[1, 5, 7]
sage: B = SteenrodAlgebra(7)
sage: a = B.Q(1,2,5)
sage: b = B.P(2,2,3)
```

```
sage: excess(a)
3
sage: excess(b)
14
sage: excess(a + b)
3
sage: excess(a * b)
17
```

integer_base_2_log(*n*)

Largest integer k so that $2^k <= n$

INPUT:

- n - positive integer

OUTPUT:

- k - integer

This returns the integer k so that $2^k <= n$ and $2^{k+1} > n$.

EXAMPLES:

```
sage: sage.algebras.steenrod_algebra_element.integer_base_2_log(7)
2
sage: sage.algebras.steenrod_algebra_element.integer_base_2_log(8)
3
sage: sage.algebras.steenrod_algebra_element.integer_base_2_log(9)
3
```

milnor(*x*)

Milnor representation of x.

INPUT:

- x - element of the Steenrod algebra

OUTPUT: Milnor representation of x

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import milnor
sage: x = Sq(5) * Sq(2) * Sq(1); x.adem()
Sq^{5} Sq^{2} Sq^{1}
sage: milnor(x)
Sq(1,0,1) + Sq(5,1)
```

milnor_mono_to_string(*mono, latex=False, p=2*)

String representation of element of the Milnor basis.

This is used by the _repr_ and _latex_ methods.

INPUT:

- mono - if $p = 2$, tuple of non-negative integers (a,b,c,...); if $p > 2$, pair of tuples of non-negative integers ((e0, e1, e2, ...), (r1, r2, ...))
- latex - boolean (optional, default False), if true, output LaTeX string
- p - positive prime number (optional, default 2)

OUTPUT:

- rep - string

This returns a string like 'Sq(a,b,c,...)' when p=2, or a string like 'Q_e0 Q_e1 Q_e2 ... P(r1, r2, ...)' when p is odd.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import milnor_mono_to_string
sage: milnor_mono_to_string((1,2,3,4))
'Sq(1,2,3,4)'
sage: milnor_mono_to_string((1,2,3,4),latex=True)
'\text{Sq}(1,2,3,4)'
sage: milnor_mono_to_string(((1,0), (2,3,1)), p=3)
'Q_1 Q_0 P(2,3,1)'
sage: milnor_mono_to_string(((1,0), (2,3,1)), latex=True, p=3)
'Q_{1} Q_{0} \mathcal{P}(2,3,1)'
```

The empty tuple represents the unit element Sq(0) (or P(0) at an odd prime):

```
sage: milnor_mono_to_string(())
'Sq(0)'
sage: milnor_mono_to_string((), p=5)
'P(0)'
```

pst (*s, t, p=2*)

The Margolis element P_t^s.

INPUT:

- s - non-negative integer
- t - positive integer
- p - positive prime number (optional, default 2)

OUTPUT: element of the Steenrod algebra

This returns the Margolis element P_t^s of the mod p Steenrod algebra: the element equal to $P(0, 0, ..., 0, p^s)$, where the p^s is in position t.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import pst
sage: pst(3,5)
Sq(0,0,0,0,8)
sage: pst(1,2) + Sq(4)*Sq(2) + Sq(2)*Sq(4)
0
sage: pst(3,5,5)
P(0,0,0,0,125)
sage: pst(3,5,p=5)
P(0,0,0,0,125)
```

pst_mono_to_string (*mono, latex=False, p=2*)

String representation of element of a P_t^s-basis.

This is used by the _repr_ and _latex_ methods.

INPUT:

- mono - tuple of pairs of integers (s,t) with $s >= 0$, $t > 0$

OUTPUT:

- string - concatenation of 'P_t^s' for each pair (s,t)

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import pst_mono_to_string
sage: pst_mono_to_string(((1,2),(0,3)))
'P^{1}_{2} P^{0}_{3}'
sage: pst_mono_to_string(((1,2),(0,3)),latex=True)
'P^{1}_{2} P^{0}_{3}'
```

The empty tuple represents the unit element Sq(0):

```
sage: pst_mono_to_string(())
'Sq(0)'
```

serre_cartan(*x*)

Serre-Cartan representation of x.

INPUT:

- x - element of the Steenrod algebra

OUTPUT: Serre-Cartan representation of x

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import serre_cartan
sage: x = Sq(3,2); x
Sq(3,2)
sage: serre_cartan(x)
Sq^{7} Sq^{2}
```

serre_cartan_mono_to_string(*mono, latex=False, p=2*)

String representation of element of the Serre-Cartan basis.

This is used by the _repr_ and _latex_ methods.

INPUT:

- mono - tuple of positive integers (a,b,c,...) when $p = 2$, or tuple (e0, n1, e1, n2, ...) when $p > 2$, where each ei is 0 or 1, and each ni is positive
- latex - boolean (optional, default False), if true, output LaTeX string
- p - positive prime number (optional, default 2)

OUTPUT:

- rep - string

This returns a string like '$Sq^a Sq^b Sq^c$... when $p = 2$, or a string like $\beta^{e0} P^{n1} \beta^{e1} P^{n2}$... when p is odd.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import serre_cartan_mono_to_string
sage: serre_cartan_mono_to_string((1,2,3,4))
'Sq^{1} Sq^{2} Sq^{3} Sq^{4}'
sage: serre_cartan_mono_to_string((1,2,3,4),latex=True)
'\\text{Sq}^{1} \\text{Sq}^{2} \\text{Sq}^{3} \\text{Sq}^{4}'
sage: serre_cartan_mono_to_string((0,5,1,1,0), p=3)
'P^{5} beta P^{1}'
sage: serre_cartan_mono_to_string((0,5,1,1,0), p=3, latex=True)
'\\mathcal{P}^{5} \\beta \\mathcal{P}^{1}'
```

The empty tuple represents the unit element Sq^0 (or P^0 at an odd prime):

```
sage: serre_cartan_mono_to_string(())
'Sq^{0}'
sage: serre_cartan_mono_to_string((), p=7)
'P^{0}'
```

string_rep(*element, latex=False, sort=True*)

String representation of element.

INPUT:

- `element` - element of the Steenrod algebra
- `latex` - boolean (optional, default False), if True, output LaTeX string
- `sort` - boolean (optional, default True), if True, sort output

OUTPUT:

- `string` - string representation of element in current basis

If latex is True, output a string suitable for LaTeX; otherwise, output a plain string. If sort is True, sort element left lexicographically; otherwise, no sorting is done, and so the order in which the summands are printed may be unpredictable.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import string_rep
sage: a = Sq(0,0,2)
sage: A = SteenrodAlgebra(2, 'admissible')
sage: string_rep(A(a))
'Sq^{8} Sq^{4} Sq^{2} + Sq^{9} Sq^{4} Sq^{1} + Sq^{10} Sq^{3} Sq^{1} +
Sq^{10} Sq^{4} + Sq^{11} Sq^{2} Sq^{1} + Sq^{12} Sq^{2} + Sq^{13} Sq^{1}
+ Sq^{14}'
sage: b = Sq(0,2)
sage: string_rep(A(b),latex=True)
'\text{Sq}^{4} \text{Sq}^{2} + \text{Sq}^{5} \text{Sq}^{1} +
\text{Sq}^{6}'
sage: A_wood_z = SteenrodAlgebra(2, 'woodz')
sage: string_rep(A_wood_z(a))
'Sq^{6} Sq^{7} Sq^{1} + Sq^{14} + Sq^{4} Sq^{7} Sq^{3} + Sq^{4} Sq^{7}
Sq^{2} Sq^{1} + Sq^{12} Sq^{2} + Sq^{8} Sq^{6} + Sq^{8} Sq^{4} Sq^{2}'
sage: string_rep(SteenrodAlgebra(2, 'arnonc')(a), sort=False)
'Sq^{4} Sq^{4} Sq^{6} + Sq^{6} Sq^{8} + Sq^{4} Sq^{2} Sq^{8} + Sq^{4}
Sq^{6} Sq^{4} + Sq^{8} Sq^{4} Sq^{2} + Sq^{8} Sq^{6}'
sage: string_rep(SteenrodAlgebra(2, 'arnonc')(a))
'Sq^{4} Sq^{2} Sq^{8} + Sq^{4} Sq^{4} Sq^{6} + Sq^{4} Sq^{6} Sq^{4} +
Sq^{6} Sq^{8} + Sq^{8} Sq^{4} Sq^{2} + Sq^{8} Sq^{6}'
sage: string_rep(SteenrodAlgebra(2, 'pst_llex')(a))
'P^{1}_{3}'
sage: Ac = SteenrodAlgebra(2, 'comm_revz')
sage: string_rep(Ac(a),latex=True,sort=False)
'c_{0,2} c_{0,3} c_{2,1} + c_{1,3} + c_{0,1} c_{1,1} c_{0,3} c_{2,1}'
sage: string_rep(Ac(a),latex=True)
'c_{0,1} c_{1,1} c_{0,3} c_{2,1} + c_{0,2} c_{0,3} c_{2,1} + c_{1,3}'
sage: string_rep(a)
'Sq(0,0,2)'
sage: string_rep(a,latex=True)
'\text{Sq}(0,0,2)'
```

Some odd primary examples:

```
sage: A5 = SteenrodAlgebra(5)
sage: a = A5.P(5,1); b = A5.Q(0,1,3)
sage: string_rep(b)
'Q_0 Q_1 Q_3'
sage: string_rep(a, latex=True)
'\mathcal{P}(5,1)'
sage: A5sc = SteenrodAlgebra(5, 'serre-cartan')
sage: string_rep(A5sc(a))
'P^{10} P^{1} + 4 P^{11}'
```

wall_long_mono_to_string(*mono, latex=False, p=2*)

Alternate string representation of element of Wall's basis.

This is used by the _repr_ and _latex_ methods.

INPUT:

- mono - tuple of pairs of non-negative integers (m,k) with $m >= k$

OUTPUT:

- string - concatenation of terms of the form '$Sq(2^m)$'

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import wall_long_mono_to_string
sage: wall_long_mono_to_string(((1,2),(3,0)))
'Sq^{1} Sq^{2} Sq^{4} Sq^{8}'
sage: wall_long_mono_to_string(((1,2),(3,0)),latex=True)
'\text{Sq}^{1} \text{Sq}^{2} \text{Sq}^{4} \text{Sq}^{8}'
```

The empty tuple represents the unit element Sq(0):

```
sage: wall_long_mono_to_string(())
'Sq(0)'
```

wall_mono_to_string(*mono, latex=False, p=2*)

String representation of element of Wall's basis.

This is used by the _repr_ and _latex_ methods.

INPUT:

- mono - tuple of pairs of non-negative integers (m,k) with $m >= k$

OUTPUT:

- string - concatenation of 'Q^m_k' for each pair (m,k)

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import wall_mono_to_string
sage: wall_mono_to_string(((1,2),(3,0)))
'Q^{1}_{2} Q^{3}_{0}'
sage: wall_mono_to_string(((1,2),(3,0)),latex=True)
'Q^{1}_{2} Q^{3}_{0}'
```

The empty tuple represents the unit element Sq(0):

```
sage: wall_mono_to_string(())
'Sq(0)'
```

wood_mono_to_string(*mono, latex=False, p=2*)

String representation of element of Wood's Y and Z bases.

This is used by the _repr_ and _latex_ methods.

INPUT:

- `mono` - tuple of pairs of non-negative integers (s,t)

OUTPUT:

- `string` - concatenation of '$Sq^{2^s(2^{t+1}-1)}$' for each pair (s,t)

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_element import wood_mono_to_string
sage: wood_mono_to_string(((1,2),(3,0)))
'Sq^{14} Sq^{8}'
sage: wood_mono_to_string(((1,2),(3,0)),latex=True)
'\text{Sq}^{14} \text{Sq}^{8}'
```

The empty tuple represents the unit element Sq(0):

```
sage: wood_mono_to_string(())
'Sq(0)'
```

30.7 Steenrod algebra bases

AUTHORS:

- John H. Palmieri (2008-07-30): version 0.9

This package defines functions for computing various bases of the Steenrod algebra, and for converting between the Milnor basis and any other basis.

This packages implements a number of different bases, at least at the prime 2. The Milnor and Serre-Cartan bases are the most familiar and most standard ones, and all of the others are defined in terms of one of these. The bases are described in the documentation for the function `steenrod_algebra_basis`; also see the papers by Monks [M] and Wood [W] for more information about them. For commutator bases, see the preprint by Palmieri and Zhang [PZ].

- 'milnor': Milnor basis.
- 'serre-cartan' or 'adem' or 'admissible': Serre-Cartan basis.

The other bases are as follows; these are only defined when $p = 2$:

- 'wood_y': Wood's Y basis.
- 'wood_z': Wood's Z basis.
- 'wall', 'wall_long': Wall's basis.
- 'arnon_a', 'arnon_a_long': Arnon's A basis.
- 'arnon_c': Arnon's C basis.
- 'pst', 'pst_rlex', 'pst_llex', 'pst_deg', 'pst_revz': various P_t^s-bases.

- 'comm', 'comm_rlex', 'comm_llex', 'comm_deg', 'comm_revz', or these with '_long' appended: various commutator bases.

EXAMPLES:

```
sage: steenrod_algebra_basis(7,'milnor')
(Sq(0,0,1), Sq(1,2), Sq(4,1), Sq(7))
sage: steenrod_algebra_basis(5)   # milnor basis is the default
(Sq(2,1), Sq(5))
```

The third (optional) argument to `steenrod_algebra_basis` is the prime p:

```
sage: steenrod_algebra_basis(9, 'milnor', p=3)
(Q_1 P(1), Q_0 P(2))
sage: steenrod_algebra_basis(9, 'milnor', 3)
(Q_1 P(1), Q_0 P(2))
sage: steenrod_algebra_basis(17, 'milnor', 3)
(Q_2, Q_1 P(3), Q_0 P(0,1), Q_0 P(4))
```

Other bases:

```
sage: steenrod_algebra_basis(7,'admissible')
(Sq^{7}, Sq^{6} Sq^{1}, Sq^{4} Sq^{2} Sq^{1}, Sq^{5} Sq^{2})
sage: [x.basis('milnor') for x in steenrod_algebra_basis(7,'admissible')]
[Sq(7),
Sq(4,1) + Sq(7),
Sq(0,0,1) + Sq(1,2) + Sq(4,1) + Sq(7),
Sq(1,2) + Sq(7)]
sage: Aw = SteenrodAlgebra(2, basis = 'wall_long')
sage: [Aw(x) for x in steenrod_algebra_basis(7,'admissible')]
[Sq^{1} Sq^{2} Sq^{4},
Sq^{2} Sq^{4} Sq^{1},
Sq^{4} Sq^{2} Sq^{1},
Sq^{4} Sq^{1} Sq^{2}]
sage: steenrod_algebra_basis(13,'admissible',p=3)
(beta P^{3}, P^{3} beta)
sage: steenrod_algebra_basis(5,'wall')
(Q^{2}_{2} Q^{0}_{0}, Q^{1}_{1} Q^{1}_{0})
sage: steenrod_algebra_basis(5,'wall_long')
(Sq^{4} Sq^{1}, Sq^{2} Sq^{1} Sq^{2})
sage: steenrod_algebra_basis(5,'pst-rlex')
(P^{0}_{1} P^{2}_{1}, P^{1}_{1} P^{0}_{2})
```

This file also contains a function `milnor_convert` which converts elements from the (default) Milnor basis representation to a representation in another basis. The output is a dictionary which gives the new representation; its form depends on the chosen basis. For example, in the basis of admissible sequences (a.k.a. the Serre-Cartan basis), each basis element is of the form $\mathrm{Sq}^a\mathrm{Sq}^b...$, and so is represented by a tuple $(a, b, ...)$ of integers. Thus the dictionary has such tuples as keys, with the coefficient of the basis element as the associated value:

```
sage: from sage.algebras.steenrod_algebra_bases import milnor_convert
sage: milnor_convert(Sq(2)*Sq(4) + Sq(2)*Sq(5), 'admissible')
{(5, 1): 1, (6, 1): 1, (6,): 1}
sage: milnor_convert(Sq(2)*Sq(4) + Sq(2)*Sq(5), 'pst')
{((1, 1), (2, 1)): 1, ((0, 1), (1, 1), (2, 1)): 1, ((0, 2), (2, 1)): 1}
```

Users shouldn't need to call `milnor_convert`; they should use the `basis` method to view a single element in another basis, or define a Steenrod algebra with a different default basis and work in that algebra:

```
sage: x = Sq(2)*Sq(4) + Sq(2)*Sq(5)
sage: x
Sq(3,1) + Sq(4,1) + Sq(6) + Sq(7)
sage: x.basis('milnor')  # 'milnor' is the default basis
Sq(3,1) + Sq(4,1) + Sq(6) + Sq(7)
sage: x.basis('adem')
Sq^{5} Sq^{1} + Sq^{6} + Sq^{6} Sq^{1}
sage: x.basis('pst')
P^{0}_{1} P^{1}_{1} P^{2}_{1} + P^{0}_{2} P^{2}_{1} + P^{1}_{1} P^{2}_{1}
sage: A = SteenrodAlgebra(2, basis='pst')
sage: A(Sq(2) * Sq(4) + Sq(2) * Sq(5))
P^{0}_{1} P^{1}_{1} P^{2}_{1} + P^{0}_{2} P^{2}_{1} + P^{1}_{1} P^{2}_{1}
```

INTERNAL DOCUMENTATION:

If you want to implement a new basis for the Steenrod algebra:

In the file 'steenrod_algebra.py':

- add functionality to `get_basis_name`: this should accept as input various synonyms for the basis, and its output should be an element of `_steenrod_basis_unique_names` (see the next file).

In the file 'steenrod_algebra_element.py':

- add name of basis to `_steenrod_basis_unique_names`
- add functionality to `string_rep`, which probably involves adding a `BASIS_mono_to_string` function
- add functionality to the `_basis_dictionary` method

In this file 'steenrod_algebra_bases.py':

- add appropriate lines to `steenrod_algebra_basis`
- add a function to compute the basis in a given dimension (to be called by `steenrod_algebra_basis`)

REFERENCES:

- [M] K. G. Monks, "Change of basis, monomial relations, and P_t^s bases for the Steenrod algebra," J. Pure Appl. Algebra 125 (1998), no. 1-3, 235-260.
- [PZ] J. H. Palmieri and J. J. Zhang, "Commutators in the Steenrod algebra," preprint (2008)
- [W] R. M. W. Wood, "Problems in the Steenrod algebra," Bull. London Math. Soc. 30 (1998), no. 5, 449-517.

Q(*m, k, basis*)

Compute Q_k^m (Wall basis) and X_k^m (Arnon's A basis).

INPUT:

- `m, k` - non-negative integers with $m >= k$
- `basis` - 'wall' or 'arnona'

OUTPUT: element of Steenrod algebra

If basis is 'wall', this returns $Q_k^m = Sq(2^k)Sq(2^{k+1})...Sq(2^m)$. If basis is 'arnona', it returns the reverse of this: $X_k^m = Sq(2^m)...Sq(2^{k+1})Sq(2^k)$

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import Q
sage: Q(2,2,'wall')
Sq(4)
sage: Q(2,2,'arnona')
Sq(4)
sage: Q(3,2,'wall')
Sq(6,2) + Sq(12)
sage: Q(3,2,'arnona')
Sq(0,4) + Sq(3,3) + Sq(6,2) + Sq(12)
```

arnonC_basis(*n, bound=1*)

Arnon's C basis in dimension n.

INPUT:

- n - non-negative integer
- bound - positive integer (optional)

OUTPUT: tuple of basis elements in dimension n

The elements of Arnon's C basis are monomials of the form $\mathrm{Sq}^{t_1}...\mathrm{Sq}^{t_m}$ where for each i, we have $t_i \leq 2t_{i+1}$ and $2^i | t_{m-i}$.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import arnonC_basis
sage: arnonC_basis(7)
(Sq^{7}, Sq^{2} Sq^{5}, Sq^{4} Sq^{3}, Sq^{4} Sq^{2} Sq^{1})
```

If optional argument bound is present, include only those monomials whose first term is at least as large as bound:

```
sage: arnonC_basis(7,3)
(Sq^{7}, Sq^{4} Sq^{3}, Sq^{4} Sq^{2} Sq^{1})
```

atomic_basis(*n, basis, long=False*)

Basis for dimension n made of elements in 'atomic' degrees: degrees of the form $2^i(2^j - 1)$.

INPUT:

- n - non-negative integer
- basis - string, the name of the basis

OUTPUT: tuple of basis elements in dimension n

The atomic bases include Wood's Y and Z bases, Wall's basis, Arnon's A basis, the P_t^s-bases, and the commutator bases. (All of these bases are constructed similarly, hence their constructions have been consolidated into a single function. Also, see the documentation for 'steenrod_algebra_basis' for descriptions of them.)

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import atomic_basis
sage: atomic_basis(6,'woody')
(Sq^{2} Sq^{3} Sq^{1}, Sq^{4} Sq^{2}, Sq^{6})
sage: atomic_basis(8,'woodz')
(Sq^{4} Sq^{3} Sq^{1}, Sq^{7} Sq^{1}, Sq^{6} Sq^{2}, Sq^{8})
sage: atomic_basis(6,'woodz') == atomic_basis(6, 'woody')
True
sage: atomic_basis(9,'woodz') == atomic_basis(9, 'woody')
False
```

Wall's basis:

```
sage: atomic_basis(6,'wall')
(Q^{1}_{1} Q^{1}_{0} Q^{0}_{0}, Q^{2}_{2} Q^{1}_{1}, Q^{2}_{1})
```

Elements of the Wall basis have an alternate, 'long' representation as monomials in the Sq^{2^n} s:

```
sage: atomic_basis(6, 'wall', long=True)
(Sq^{2} Sq^{1} Sq^{2} Sq^{1}, Sq^{4} Sq^{2}, Sq^{2} Sq^{4})
```

Arnon's A basis:

```
sage: atomic_basis(7,'arnona')
(X^{0}_{0} X^{1}_{1} X^{2}_{2},
X^{0}_{0} X^{2}_{1},
X^{1}_{0} X^{2}_{2},
X^{2}_{0})
```

These also have a 'long' representation:

```
sage: atomic_basis(7,'arnona',long=True)
(Sq^{1} Sq^{2} Sq^{4},
Sq^{1} Sq^{4} Sq^{2},
Sq^{2} Sq^{1} Sq^{4},
Sq^{4} Sq^{2} Sq^{1})
```

P_t^s-bases:

```
sage: atomic_basis(7,'pst_rlex')
(P^{0}_{1} P^{1}_{1} P^{2}_{1},
P^{0}_{1} P^{1}_{2},
P^{2}_{1} P^{0}_{2},
P^{0}_{3})
sage: atomic_basis(7,'pst_llex')
(P^{0}_{1} P^{1}_{1} P^{2}_{1},
P^{0}_{1} P^{1}_{2},
P^{0}_{2} P^{2}_{1},
P^{0}_{3})
sage: atomic_basis(7,'pst_deg')
(P^{0}_{1} P^{1}_{1} P^{2}_{1},
P^{0}_{1} P^{1}_{2},
P^{0}_{2} P^{2}_{1},
P^{0}_{3})
sage: atomic_basis(7,'pst_revz')
(P^{0}_{1} P^{1}_{1} P^{2}_{1},
P^{0}_{1} P^{1}_{2},
P^{0}_{2} P^{2}_{1},
P^{0}_{3})
```

Commutator bases:

```
sage: atomic_basis(7,'comm_rlex')
(c_{0,1} c_{1,1} c_{2,1}, c_{0,1} c_{1,2}, c_{2,1} c_{0,2}, c_{0,3})
sage: atomic_basis(7,'comm_llex')
(c_{0,1} c_{1,1} c_{2,1}, c_{0,1} c_{1,2}, c_{0,2} c_{2,1}, c_{0,3})
sage: atomic_basis(7,'comm_deg')
(c_{0,1} c_{1,1} c_{2,1}, c_{0,1} c_{1,2}, c_{0,2} c_{2,1}, c_{0,3})
sage: atomic_basis(7,'comm_revz')
(c_{0,1} c_{1,1} c_{2,1}, c_{0,1} c_{1,2}, c_{0,2} c_{2,1}, c_{0,3})
```

Long representations of commutator bases:

```
sage: atomic_basis(7,'comm_revz', long=True)
(s_{1} s_{2} s_{4}, s_{1} s_{24}, s_{12} s_{4}, s_{124})
```

commutator(*s, t*)

Returns the t^{th} iterated commutator of consecutive Sq^{2^i} 's.

INPUT: s, t: integers

OUTPUT: element of the Steenrod algebra

If t=1, return $Sq(2^s)$. Otherwise, return the commutator $[commutator(s, t-1), \mathrm{Sq}(2^{s+t-1})]$.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import commutator
sage: commutator(1,2)
Sq(0,2)
sage: commutator(0,4)
Sq(0,0,0,1)
sage: commutator(2,2)
Sq(0,4) + Sq(3,3)
```

Note: `commutator(0,n)` is equal to $\mathrm{Sq}(0, ..., 0, 1)$, with the 1 in the n^{th} spot. commutator(i,n) always has $\mathrm{Sq}(0, ..., 0, 2^i)$, with 2^i in the n^{th} spot, as a summand, but there may be other terms, as the example of commutator(2,2) illustrates.

That is, commutator(s,t) is equal to P^s_t, possibly plus other Milnor basis elements.

convert_from_milnor_matrix(*n, basis, p=2*)

Change-of-basis matrix, Milnor to 'basis', in dimension n.

INPUT:

- `n` - non-negative integer, the dimension
- `basis` - string, the basis to which to convert
- `p` - positive prime number (optional, default 2)

OUTPUT:

- `matrix` - change-of-basis matrix, a square matrix over GF(p)

(This is not really intended for casual users, so no error checking is made on the integer n, the basis name, or the prime.)

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import convert_from_milnor_matrix, convert_to_mi
sage: convert_from_milnor_matrix(12,'wall')
[1 0 0 1 0 0 0]
[0 0 1 1 0 0 0]
[0 0 0 1 0 1 1]
[0 0 0 1 0 0 0]
[1 0 1 0 1 0 0]
[1 1 1 0 0 0 0]
[1 0 1 0 1 0 1]
sage: convert_from_milnor_matrix(38,'serre_cartan')
72 x 72 dense matrix over Finite Field of size 2
sage: x = convert_to_milnor_matrix(20,'wood_y')
sage: y = convert_from_milnor_matrix(20,'wood_y')
sage: x*y
```

```
[1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1]
```

The function takes an optional argument, the prime p over which to work:

```
sage: convert_from_milnor_matrix(17,'adem',3)
[2 1 1 2]
[0 2 0 1]
[1 2 0 0]
[0 1 0 0]
```

convert_to_milnor_matrix(*n, basis, p=2*)

Change-of-basis matrix, 'basis' to Milnor, in dimension n, at the prime p.

INPUT:

- `n` - non-negative integer, the dimension
- `basis` - string, the basis from which to convert
- `p` - positive prime number (optional, default 2)

OUTPUT:

- `matrix` - change-of-basis matrix, a square matrix over GF(p)

(This is not really intended for casual users, so no error checking is made on the integer n, the basis name, or the prime.)

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import convert_to_milnor_matrix
sage: convert_to_milnor_matrix(5, 'adem')
[0 1]
[1 1]
sage: convert_to_milnor_matrix(45, 'milnor')
111 x 111 dense matrix over Finite Field of size 2
sage: convert_to_milnor_matrix(12,'wall')
[1 0 0 1 0 0 0]
[1 1 0 0 0 1 0]
[0 1 0 1 0 0 0]
[0 0 0 1 0 0 0]
[1 1 0 0 1 0 0]
[0 0 1 1 1 0 1]
[0 0 0 0 1 0 1]
```

The function takes an optional argument, the prime p over which to work:

```
sage: convert_to_milnor_matrix(17,'adem',3)
[0 0 1 1]
[0 0 0 1]
[1 1 1 1]
[0 1 0 1]
sage: convert_to_milnor_matrix(48,'adem',5)
[0 1]
[1 1]
sage: convert_to_milnor_matrix(36,'adem',3)
[0 0 1]
[0 1 0]
[1 2 0]
```

list_to_hist (*list*)

Given list of positive integers [a,b,c,...], return corresponding 'histogram'.

That is, in the output [n1, n2, ...], n1 is the number of 1's in the original list, n2 is the number of 2's, etc.

INPUT:

- `list` - list of positive integers

OUTPUT:

- `answer` - list of non-negative integers

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import list_to_hist
sage: list_to_hist([1,2,3,4,2,1,2])
[2, 3, 1, 1]
sage: list_to_hist([2,2,2,2])
[0, 4]
```

make_elt_homogeneous (*poly*)

Break element of the Steenrod algebra into a list of homogeneous pieces.

INPUT:

- `poly` - an element of the Steenrod algebra

OUTPUT: list of homogeneous elements of the Steenrod algebra whose sum is poly

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import make_elt_homogeneous
sage: make_elt_homogeneous(Sq(2)*Sq(4) + Sq(2)*Sq(5))
[Sq(3,1) + Sq(6), Sq(4,1) + Sq(7)]
```

milnor_P_basis (*n, p=2*)

Milnor P basis in dimension n.

INPUT:

- `n` - non-negative integer
- `p` - positive prime number (optional, default 2)

OUTPUT:

- tuple of mod p Milnor P basis elements in dimension n

At the prime 2, the Milnor P basis consists of symbols of the form $\mathrm{Sq}(m_1, m_2, ..., m_t)$, where each m_i is a non-negative integer and if $t > 1$, then $m_t \neq 0$. At odd primes, it consists of symbols of the form $P(m_1, m_2, ..., m_t)$, where each m_i is a non-negative integer, and if $t > 1$, then $m_t \neq 0$.

Thus at the prime 2, this is just the Milnor basis. At odd primes, it is a subset of the Milnor basis of those terms with no Q factors.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import milnor_P_basis
sage: milnor_P_basis(7)
(Sq(0,0,1), Sq(1,2), Sq(4,1), Sq(7))
sage: milnor_P_basis(7, 2)
(Sq(0,0,1), Sq(1,2), Sq(4,1), Sq(7))
sage: milnor_P_basis(9, 3)
()
sage: milnor_P_basis(17, 3)
()
sage: milnor_P_basis(48, p=5)
(P(0,1), P(6))
sage: len(milnor_P_basis(100,3))
11
sage: len(milnor_P_basis(200,7))
0
sage: len(milnor_P_basis(240,7))
3
```

milnor_basis(*n, p=2*)

Milnor basis in dimension n.

INPUT:

- n - non-negative integer
- p - positive prime number (optional, default 2)

OUTPUT: tuple of mod p Milnor basis elements in dimension n

At the prime 2, the Milnor basis consists of symbols of the form $\mathrm{Sq}(m_1, m_2, ..., m_t)$, where each m_i is a non-negative integer and if $t > 1$, then $m_t \neq 0$. At odd primes, it consists of symbols of the form $Q_{e_1}Q_{e_2}...P(m_1, m_2, ..., m_t)$, where $0 \leq e_1 < e_2 < ...$, each m_i is a non-negative integer, and if $t > 1$, then $m_t \neq 0$.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import milnor_basis
sage: milnor_basis(7)
(Sq(0,0,1), Sq(1,2), Sq(4,1), Sq(7))
sage: milnor_basis(7, 2)
(Sq(0,0,1), Sq(1,2), Sq(4,1), Sq(7))
sage: milnor_basis(9, 3)
(Q_1 P(1), Q_0 P(2))
sage: milnor_basis(17, 3)
(Q_2, Q_1 P(3), Q_0 P(0,1), Q_0 P(4))
sage: milnor_basis(48, p=5)
(P(0,1), P(6))
sage: len(milnor_basis(100,3))
13
sage: len(milnor_basis(200,7))
0
sage: len(milnor_basis(240,7))
3
```

milnor_convert (*poly, basis*)

Convert an element of the Steenrod algebra in the Milnor basis to its representation in the chosen basis.

INPUT:

- `poly` - element of the Steenrod algebra
- `basis` - basis to which to convert

OUTPUT:

- `dict` - dictionary

This returns a dictionary of terms of the form (mono: coeff), where mono is a monomial in 'basis'. The form of mono depends on the chosen basis.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import milnor_convert
sage: milnor_convert(Sq(2)*Sq(4) + Sq(2)*Sq(5), 'adem')
{(5, 1): 1, (6, 1): 1, (6,): 1}
sage: A3 = SteenrodAlgebra(3)
sage: a = A3.Q(1) * A3.P(2,2); a
Q_1 P(2,2)
sage: milnor_convert(a, 'adem')
{(0, 9, 1, 2, 0): 1, (1, 9, 0, 2, 0): 2}
sage: milnor_convert(2 * a, 'adem')
{(0, 9, 1, 2, 0): 2, (1, 9, 0, 2, 0): 1}
sage: (Sq(2)*Sq(4) + Sq(2)*Sq(5)).basis('adem')
Sq^{5} Sq^{1} + Sq^{6} + Sq^{6} Sq^{1}
sage: a.basis('adem')
P^{9} beta P^{2} + 2 beta P^{9} P^{2}
sage: milnor_convert(Sq(2)*Sq(4) + Sq(2)*Sq(5), 'pst')
{((1, 1), (2, 1)): 1, ((0, 1), (1, 1), (2, 1)): 1, ((0, 2), (2, 1)): 1}
sage: (Sq(2)*Sq(4) + Sq(2)*Sq(5)).basis('pst')
P^{0}_{1} P^{1}_{1} P^{2}_{1} + P^{0}_{2} P^{2}_{1} + P^{1}_{1}
P^{2}_{1}
```

restricted_partitions (*n, list, no_repeats=False*)

List of 'restricted' partitions of n: partitions with parts taken from list.

INPUT:

- `n` - non-negative integer
- `list` - list of positive integers
- `no_repeats` - boolean (optional, default = False), if True, only return partitions with no repeated parts

This seems to be faster than RestrictedPartitions, although I don't know why. Maybe because all I want is the list of partitions (with each partition represented as a list), not the extra stuff provided by RestrictedPartitions.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import restricted_partitions
sage: restricted_partitions(10, [7,5,1])
[[7, 1, 1, 1], [5, 5], [5, 1, 1, 1, 1, 1], [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]]
sage: restricted_partitions(10, [6,5,4,3,2,1], no_repeats=True)
[[6, 4], [6, 3, 1], [5, 4, 1], [5, 3, 2], [4, 3, 2, 1]]
sage: restricted_partitions(10, [6,4,2])
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
sage: restricted_partitions(10, [6,4,2], no_repeats=True)
[[6, 4]]
```

'list' may have repeated elements. If 'no_repeats' is False, this has no effect. If 'no_repeats' is True, and if the repeated elements appear consecutively in 'list', then each element may be used only as many times as it appears in 'list':

```
sage: restricted_partitions(10, [6,4,2,2], no_repeats=True)
[[6, 4], [6, 2, 2]]
sage: restricted_partitions(10, [6,4,2,2,2], no_repeats=True)
[[6, 4], [6, 2, 2], [4, 2, 2, 2]]
```

(If the repeated elements don't appear consecutively, the results are likely meaningless, containing several partitions more than once, for example.)

In the following examples, 'no_repeats' is False:

```
sage: restricted_partitions(10, [6,4,2])
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
sage: restricted_partitions(10, [6,4,2,2,2])
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
sage: restricted_partitions(10, [6,4,4,4,2,2,2,2,2,2])
[[6, 4], [6, 2, 2], [4, 4, 2], [4, 2, 2, 2], [2, 2, 2, 2, 2]]
```

serre_cartan_basis (*n, p=2, bound=1*)

Serre-Cartan basis in dimension n.

INPUT:

- `n` - non-negative integer
- `bound` - positive integer (optional)
- `prime` - positive prime number (optional, default 2)

OUTPUT: tuple of mod p Serre-Cartan basis elements in dimension n

The Serre-Cartan basis consists of 'admissible monomials in the Steenrod squares'. Thus at the prime 2, it consists of monomials $\mathrm{Sq}^{m_1}\mathrm{Sq}^{m_2}...\mathrm{Sq}^{m_t}$ with $m_i \geq 2m_{i+1}$ for each i. At odd primes, it consists of monomials $\beta^{e_0}P^{s_1}\beta^{e_1}P^{s_2}...P^{s_k}\beta^{e_k}$ with each e_i either 0 or 1, $s_i \geq ps_{i+1} + e_i$, and $s_k \geq 1$.

EXAMPLES:

```
sage: from sage.algebras.steenrod_algebra_bases import serre_cartan_basis
sage: serre_cartan_basis(7)
(Sq^{7}, Sq^{6} Sq^{1}, Sq^{4} Sq^{2} Sq^{1}, Sq^{5} Sq^{2})
sage: serre_cartan_basis(13,3)
(beta P^{3}, P^{3} beta)
sage: serre_cartan_basis(50,5)
(beta P^{5} P^{1} beta, beta P^{6} beta)
```

If optional argument bound is present, include only those monomials whose last term is at least bound (when p=2), or those for which $s_k - epsilon_k >= bound$ (when p is odd).

```
sage: serre_cartan_basis(7, bound=2)
(Sq^{7}, Sq^{5} Sq^{2})
sage: serre_cartan_basis(13, 3, bound=3)
(beta P^{3},)
```

steenrod_algebra_basis (*n, basis='milnor', p=2*)

Basis for the Steenrod algebra in degree n.

INPUT:

- `n` - non-negative integer

- `basis` - string, which basis to use (optional, default = 'milnor')
- `p` - positive prime number (optional, default = 2)

OUTPUT: tuple of basis elements for the Steenrod algebra in dimension n

The choices for the string basis are as follows:

- 'milnor': Milnor basis. When $p = 2$, the Milnor basis consists of symbols of the form $\mathrm{Sq}(m_1, m_2, ..., m_t)$, where each m_i is a non-negative integer and if $t > 1$, then the last entry $m_t > 0$. When p is odd, the Milnor basis consists of symbols of the form $Q_{e_1}Q_{e_2}...\mathcal{P}(m_1, m_2, ..., m_t)$, where $0 \leq e_1 < e_2 < ...$, each m_i is a non-negative integer, and if $t > 1$, then the last entry $m_t > 0$.
- 'serre-cartan' or 'adem' or 'admissible': Serre-Cartan basis. The Serre-Cartan basis consists of 'admissible monomials' in the Steenrod operations. Thus at the prime 2, it consists of monomials $\mathrm{Sq}^{m_1}\mathrm{Sq}^{m_2}...\mathrm{Sq}^{m_t}$ with $m_i \geq 2m_{i+1}$ for each i. At odd primes, it consists of monomials $\beta^{\epsilon_0}\mathcal{P}^{s_1}\beta^{\epsilon_1}\mathcal{P}^{s_2}...\mathcal{P}^{s_k}\beta^{\epsilon_k}$ with each ϵ_i either 0 or 1, $s_i \geq ps_{i+1} + \epsilon_i$, and $s_k \geq 1$.

  When $p = 2$, the element Sq^a equals the Milnor element $\mathrm{Sq}(a)$; when p is odd, $\mathcal{P}^a = \mathcal{P}(a)$ and $\beta = Q_0$. Hence for any Serre-Cartan basis element, one can represent it in the Milnor basis by computing an appropriate product using Milnor multiplication.

The rest of these bases are only defined when $p = 2$.

- 'wood_y': Wood's Y basis. For pairs of non-negative integers (m, k), let $w(m, k) = \mathrm{Sq}^{2^m(2^{k+1}-1)}$. Wood's Y basis consists of monomials $w(m_0, k_0)...w(m_t, k_t)$ with $(m_i, k_i) > (m_{i+1}, k_{i+1})$, in left lex order.
- 'wood_z': Wood's Z basis. For pairs of non-negative integers (m, k), let $w(m, k) = \mathrm{Sq}^{2^m(2^{k+1}-1)}$. Wood's Z basis consists of monomials $w(m_0, k_0)...w(m_t, k_t)$ with $(m_i+k_i, m_i) > (m_{i+1}+k_{i+1}, m_{i+1})$, in left lex order.
- 'wall' or 'wall_long': Wall's basis. For any pair of integers (m, k) with $m \geq k \geq 0$, let $Q_k^m = \mathrm{Sq}^{2^k}\mathrm{Sq}^{2^{k+1}}...\mathrm{Sq}^{2^m}$. The elements of Wall's basis are monomials $Q_{k_0}^{m_0}...Q_{k_t}^{m_t}$ with $(m_i, k_i) > (m_{i+1}, k_{i+1})$, ordered left lexicographically.

  (Note that Q_k^m is the reverse of the element X_k^m used in defining Arnon's A basis.)

  The standard way of printing elements of the Wall basis is to write elements in terms of the Q_k^m. If one sets the basis to 'wall_long' instead of 'wall', then each Q_k^m is expanded as a product of factors of the form Sq^{2^i}.
- 'arnon_a' or 'arnon_a_long': Arnon's A basis. For any pair of integers (m, k) with $m \geq k \geq 0$, let $X_k^m = \mathrm{Sq}^{2^m}\mathrm{Sq}^{2^{m-1}}...\mathrm{Sq}^{2^k}$. The elements of Arnon's A basis are monomials $X_{k_0}^{m_0}...X_{k_t}^{m_t}$ with $(m_i, k_i) < (m_{i+1}, k_{i+1})$, ordered left lexicographically.

  (Note that X_k^m is the reverse of the element Q_k^m used in defining Wall's basis.)

  The standard way of printing elements of Arnon's A basis is to write elements in terms of the X_k^m. If one sets the basis to 'arnon_a_long' instead of 'arnon_a', then each X_k^m is expanded as a product of factors of the form Sq^{2^i}.
- 'arnon_c': Arnon's C basis. The elements of Arnon's C basis are monomials of the form $\mathrm{Sq}^{t_1}...\mathrm{Sq}^{t_m}$ where for each i, we have $t_i \leq 2t_{i+1}$ and $2^i | t_{m-i}$.
- 'pst', 'pst_rlex', 'pst_llex', 'pst_deg', 'pst_revz': various P_t^s-bases. For integers $s \geq 0$ and $t > 0$, the element P_t^s is the Milnor basis element $\mathrm{Sq}(0, ..., 0, 2^s, 0, ...)$, with the nonzero entry in position t. To obtain a P_t^s-basis, for each set $\{P_{t_1}^{s_1}, ..., P_{t_k}^{s_k}\}$ of (distinct) P_t^s's, one chooses an ordering and forms the resulting monomial. The set of all such monomials then forms a basis, and so one gets a basis by choosing an ordering on each monomial.

The strings 'rlex', 'llex', etc., correspond to the following orderings. These are all 'global' - they give a global ordering on the P_t^s's, not different orderings depending on the monomial. They order the P_t^s's using the pair of integers (s, t) as follows:

- 'rlex': right lexicographic ordering
- 'llex': left lexicographic ordering
- 'deg': ordered by degree, which is the same as left lexicographic ordering on the pair $(s+t, t)$
- 'revz': left lexicographic ordering on the pair $(s+t, s)$, which is the reverse of the ordering used (on elements in the same degrees as the P_t^s's) in Wood's Z basis: 'revz' stands for 'reversed Z'. This is the default: 'pst' is the same as 'pst_revz'.
- 'comm', 'comm_rlex', 'comm_llex', 'comm_deg', 'comm_revz', or any of these with '_long' appended: various commutator bases. Let $c_{i,1} = \mathrm{Sq}^{2^i}$, let $c_{i,2} = [c_{i,1}, c_{i+1,1}]$, and inductively define $c_{i,k} = [c_{i,k-1}, c_{i+k-1,1}]$. Thus $c_{i,k}$ is a k-fold iterated commutator of the elements Sq^{2^i}, ..., $\mathrm{Sq}^{2^{i+k-1}}$. Note that $\dim c_{i,k} = \dim P_k^i$.

  To obtain a commutator basis, for each set $\{c_{s_1,t_1}, ..., c_{s_k,t_k}\}$ of (distinct) $c_{s,t}$'s, one chooses an ordering and forms the resulting monomial. The set of all such monomials then forms a basis, and so one gets a basis by choosing an ordering on each monomial. The strings 'rlex', etc., have the same meaning as for the orderings on P_t^s-bases. As with the P_t^s-bases, 'comm_revz' is the default: 'comm' means 'comm_revz'.

  The commutator bases have alternative representations, obtained by appending 'long' to their names: instead of, say, $c_{2,2}$, the representation is s_{48}, indicating the commutator of Sq^4 and Sq^8, and $c_{0,3}$, which is equal to $[[\mathrm{Sq}^1, \mathrm{Sq}^2], \mathrm{Sq}^4]$, is written as s_{124}.

EXAMPLES:

```
sage: steenrod_algebra_basis(7,'milnor')
(Sq(0,0,1), Sq(1,2), Sq(4,1), Sq(7))
sage: steenrod_algebra_basis(5)   # milnor basis is the default
(Sq(2,1), Sq(5))
```

The third (optional) argument to 'steenrod_algebra_basis' is the prime p:

```
sage: steenrod_algebra_basis(9, 'milnor', p=3)
(Q_1 P(1), Q_0 P(2))
sage: steenrod_algebra_basis(9, 'milnor', 3)
(Q_1 P(1), Q_0 P(2))
sage: steenrod_algebra_basis(17, 'milnor', 3)
(Q_2, Q_1 P(3), Q_0 P(0,1), Q_0 P(4))
```

Other bases:

```
sage: steenrod_algebra_basis(7,'admissible')
(Sq^{7}, Sq^{6} Sq^{1}, Sq^{4} Sq^{2} Sq^{1}, Sq^{5} Sq^{2})
sage: [x.basis('milnor') for x in steenrod_algebra_basis(7,'admissible')]
[Sq(7),
Sq(4,1) + Sq(7),
Sq(0,0,1) + Sq(1,2) + Sq(4,1) + Sq(7),
Sq(1,2) + Sq(7)]
sage: steenrod_algebra_basis(13,'admissible',p=3)
(beta P^{3}, P^{3} beta)
sage: steenrod_algebra_basis(5,'wall')
(Q^{2}_{2} Q^{0}_{0}, Q^{1}_{1} Q^{1}_{0})
sage: steenrod_algebra_basis(5,'wall_long')
(Sq^{4} Sq^{1}, Sq^{2} Sq^{1} Sq^{2})
sage: steenrod_algebra_basis(5,'pst-rlex')
(P^{0}_{1} P^{2}_{1}, P^{1}_{1} P^{0}_{2})
```

steenrod_basis_error_check(*dim, p*)

This performs crude error checking.

INPUT:

- dim - non-negative integer
- p - positive prime number

OUTPUT: None

This checks to see if the different bases have the same length, and if the change-of-basis matrices are invertible. If something goes wrong, an error message is printed.

This function checks at the prime p as the dimension goes up from 0, in which case the basis functions use the saved basis computations in lower dimensions in the computations. It also checks as the dimension goes down from the top, in which case it doesn't have access to the saved computations. (The saved computations are deleted first: the cache _steenrod_bases is set to {} before doing the computations.)

EXAMPLES:

```
sage: sage.algebras.steenrod_algebra_bases.steenrod_basis_error_check(12,2)
p=2, in decreasing order of dimension, starting in dimension 12.
down to dimension  10
down to dimension  5
p=2, now in increasing order of dimension, up to dimension 12
up to dimension  0
up to dimension  5
up to dimension  10
done checking
sage: sage.algebras.steenrod_algebra_bases.steenrod_basis_error_check(30,3)
p=3, in decreasing order of dimension, starting in dimension 30.
down to dimension  30
down to dimension  25
down to dimension  20
down to dimension  15
down to dimension  10
down to dimension  5
p=3, now in increasing order of dimension, up to dimension 30
up to dimension  0
up to dimension  5
up to dimension  10
up to dimension  15
up to dimension  20
up to dimension  25
done checking
```

xi_degrees(*n, p=2*)

Decreasing list of degrees of the xi_i's, starting in degree n.

INPUT:

- n - integer

OUTPUT:

- list - list of integers

When $p = 2$: decreasing list of the degrees of the ξ_i's with degree at most n.

At odd primes: decreasing list of these degrees, each divided by $2(p-1)$.

EXAMPLES:

```
sage: sage.algebras.steenrod_algebra_bases.xi_degrees(17)
[15, 7, 3, 1]
sage: sage.algebras.steenrod_algebra_bases.xi_degrees(17,p=3)
[13, 4, 1]
```

```
sage: sage.algebras.steenrod_algebra_bases.xi_degrees(400,p=17)
[307, 18, 1]
```

CHAPTER

THIRTYONE

QUATERNION ALGEBRAS

31.1 Quaternion Algebras

AUTHORS:

- Jon Bobber – 2009 rewrite
- William Stein – 2009 rewrite

This code is partly based on Sage code by David Kohel from 2005.

TESTS:

We test pickles:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ,-5,-2)
sage: Q == loads(dumps(Q))
True
```

QuaternionAlgebra(*arg0, arg1=None, arg2=None, names='i, j, k'*)

There are three input formats:

- `QuaternionAlgebra(a, b)`: quaternion algebra generated by `i`, `j` subject to $i^2 = a$, $j^2 = b$, $j * i = -i * j$.
- `QuaternionAlgebra(K, a, b)`: same as above but over a field `K`. Here, `a` and `b` are nonzero elements of a field (`K`) of characteristic not 2, and we set $k = i * j$.
- `QuaternionAlgebra(D)`: a rational quaternion algebra with discriminant `D`, where $D > 1$ is a square-free integer.

EXAMPLES:

`QuaternionAlgebra(a, b)` - return quaternion algebra over the *smallest* field containing the nonzero elements `a` and `b` with generators `i`, `j`, `k` with $i^2 = a$, $j^2 = b$ and $j * i = -i * j$:

```
sage: QuaternionAlgebra(-2,-3)
Quaternion Algebra (-2, -3) with base ring Rational Field
sage: QuaternionAlgebra(GF(5)(2), GF(5)(3))
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
sage: QuaternionAlgebra(2, GF(5)(3))
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
sage: QuaternionAlgebra(QQ[sqrt(2)](-1), -5)
Quaternion Algebra (-1, -5) with base ring Number Field in sqrt2 with defining polynomial x^2 -
sage: QuaternionAlgebra(sqrt(-1), sqrt(-3))
```

```
Quaternion Algebra (I, sqrt(-3)) with base ring Symbolic Ring
sage: QuaternionAlgebra(0,0)
...
ValueError: a and b must be nonzero
sage: QuaternionAlgebra(GF(2)(1),1)
...
ValueError: a and b must be elements of a field with characteristic not 2
```

`QuaternionAlgebra(K, a, b)` - return quaternion algebra over the field `K` with generators `i`, `j`, `k` with $i^2 = a$, $j^2 = b$ and $i * j = -j * i$:

```
sage: QuaternionAlgebra(QQ, -7, -21)
Quaternion Algebra (-7, -21) with base ring Rational Field
sage: QuaternionAlgebra(QQ[sqrt(2)], -2,-3)
Quaternion Algebra (-2, -3) with base ring Number Field in sqrt2 with defining polynomial x^2 -
```

`QuaternionAlgebra(D)` - `D` is a squarefree integer; returns a rational quaternion algebra of discriminant `D`:

```
sage: QuaternionAlgebra(1)
Quaternion Algebra (-1, 1) with base ring Rational Field
sage: QuaternionAlgebra(2)
Quaternion Algebra (-1, -1) with base ring Rational Field
sage: QuaternionAlgebra(7)
Quaternion Algebra (-1, -7) with base ring Rational Field
sage: QuaternionAlgebra(2*3*5*7)
Quaternion Algebra (-22, 210) with base ring Rational Field
```

If the coefficients a and b in the definition of the quaternion algebra are not integral, then a slower generic type is used for arithmetic:

```
sage: type(QuaternionAlgebra(-1,-3).0)
<type 'sage.algebras.quatalg.quaternion_algebra_element.QuaternionAlgebraElement_rational_field'
sage: type(QuaternionAlgebra(-1,-3/2).0)
<type 'sage.algebras.quatalg.quaternion_algebra_element.QuaternionAlgebraElement_generic'>
```

Make sure caching is sane:

```
sage: A = QuaternionAlgebra(2,3); A
Quaternion Algebra (2, 3) with base ring Rational Field
sage: B = QuaternionAlgebra(GF(5)(2),GF(5)(3)); B
Quaternion Algebra (2, 3) with base ring Finite Field of size 5
sage: A is QuaternionAlgebra(2,3)
True
sage: B is QuaternionAlgebra(GF(5)(2),GF(5)(3))
True
sage: Q = QuaternionAlgebra(2); Q
Quaternion Algebra (-1, -1) with base ring Rational Field
sage: Q is QuaternionAlgebra(QQ,-1,-1)
True
sage: Q is QuaternionAlgebra(-1,-1)
True
sage: Q.<ii,jj,kk> = QuaternionAlgebra(15); Q.variable_names()
('ii', 'jj', 'kk')
sage: QuaternionAlgebra(15).variable_names()
('i', 'j', 'k')
```

class **QuaternionAlgebra_ab**(*base_ring, a, b, names='i, j, k'*)

The quaternion algebra of the form $(a, b/K)$, where $i^2 = a$, $j^2 = b$, and $j * i = -i * j$. K is a field not of characteristic 2 and a, b are nonzero elements of K.

See QuaternionAlgebra for many more examples.

EXAMPLES:

```
sage: QuaternionAlgebra(QQ, -7, -21)  # indirect doctest
Quaternion Algebra (-7, -21) with base ring Rational Field
```

discriminant()

Given a quaternion algebra A defined over the field of rational numbers, return the discriminant of A, i.e. the product of the ramified primes of A.

EXAMPLES:

```
sage: QuaternionAlgebra(210,-22).discriminant()
210
sage: QuaternionAlgebra(19).discriminant()
19
```

This raises a NotImplementedError if the base field is not the rational numbers:

```
sage: QuaternionAlgebra(QQ[sqrt(2)],3,19).discriminant()
...
NotImplementedError: base field must be rational numbers
```

gen(*i=0*)

Return the i^{th} generator of self.

INPUT:

- i - integer (optional, default 0)

EXAMPLES:

```
sage: Q.<ii,jj,kk> = QuaternionAlgebra(QQ,-1,-2); Q
Quaternion Algebra (-1, -2) with base ring Rational Field
sage: Q.gen(0)
ii
sage: Q.gen(1)
jj
sage: Q.gen(2)
kk
sage: Q.gens()
[ii, jj, kk]
```

ideal(*gens, left_order=None, right_order=None, check=True*)

Return the quaternion ideal with given gens over $\mathbf{Z}$. Neither a left or right order structure need be specified.

INPUT:

- gens – a list of elements of this quaternion order
- check – bool (default: True); if False, then gens must 4-tuple that forms a Hermite basis for an ideal
- left_order – a quaternion order or None
- right_order – a quaternion order or None

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1)
sage: R.ideal([2*a for a in R.basis()])
Fractional ideal (2, 2*i, 2*j, 2*k)
```

inner_product_matrix()

Return the inner product matrix associated to `self`, i.e. the Gram matrix of the reduced norm as a quadratic form on `self`. The standard basis 1, i, j, k is orthogonal, so this matrix is just the diagonal matrix with diagonal entries 1, a, b, ab.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(-5,-19)
sage: Q.inner_product_matrix()
[  2   0   0   0]
[  0  10   0   0]
[  0   0  38   0]
[  0   0   0 190]

sage: R.<a,b> = QQ[]; Q.<i,j,k> = QuaternionAlgebra(Frac(R),a,b)
sage: Q.inner_product_matrix()
[    2     0     0     0]
[    0  -2*a     0     0]
[    0     0  -2*b     0]
[    0     0     0 2*a*b]
```

invariants()

Return the structural invariants a, b of this quaternion algebra: `self` is generated by i, j subject to $i^2 = a$, $j^2 = b$ and $j * i = -i * j$.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(15)
sage: Q.invariants()
(-3, 5)
sage: i^2
-3
sage: j^2
5
```

maximal_order()

Return a maximal order in this quaternion algebra.

OUTPUT: an order in this quaternion algebra

EXAMPLES:

```
sage: QuaternionAlgebra(-1,-7).maximal_order()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis (1/2 + 1/2*j,
```

quaternion_order(*basis, check=True*)

Return the order of this quaternion order with given basis.

INPUT:

- `basis` - list of 4 elements of `self`
- `check` - bool (default: `True`)

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(-11,-1)
sage: Q.quaternion_order([1,i,j,k])
Order of Quaternion Algebra (-11, -1) with base ring Rational Field with basis (1, i, j, k)
```

We test out `check=False`:

```
sage: Q.quaternion_order([1,i,j,k], check=False)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field with basis [1, i, j, k]
sage: Q.quaternion_order([i,j,k], check=False)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field with basis [i, j, k]
```

ramified_primes()

Return the primes that ramify in this quaternion algebra. Currently only implemented over the rational numbers.

EXAMPLES:

```
sage: QuaternionAlgebra(QQ, -1, -1).ramified_primes()
[2]
```

class **QuaternionAlgebra_abstract**()

basis()

Return the fixed basis of `self`, which is 1, i, j, k, where i, j, k are the generators of `self`.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ,-5,-2)
sage: Q.basis()
(1, i, j, k)

sage: Q.<xyz,abc,theta> = QuaternionAlgebra(GF(9,'a'),-5,-2)
sage: Q.basis()
(1, xyz, abc, theta)
```

The basis is cached:

```
sage: Q.basis() is Q.basis()
True
```

inner_product_matrix()

Return the inner product matrix associated to `self`, i.e. the Gram matrix of the reduced norm as a quadratic form on `self`. The standard basis 1, i, j, k is orthogonal, so this matrix is just the diagonal matrix with diagonal entries 2, $2a$, $2b$, $2ab$.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(-5,-19)
sage: Q.inner_product_matrix()
[  2   0   0   0]
[  0  10   0   0]
[  0   0  38   0]
[  0   0   0 190]
```

is_commutative()

Return `False` always, since all quaternion algebras are noncommutative.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3,-7)
sage: Q.is_commutative()
False
```

is_division_algebra()

Return `True` if the quaternion algebra is a division algebra (i.e. every nonzero element in `self` is invertible), and `False` if the quaternion algebra is isomorphic to the 2x2 matrix algebra.

EXAMPLES:

```
sage: QuaternionAlgebra(QQ,-5,-2).is_division_algebra()
True
sage: QuaternionAlgebra(1).is_division_algebra()
False
sage: QuaternionAlgebra(2,9).is_division_algebra()
False
```

```
sage: QuaternionAlgebra(RR(2.),1).is_division_algebra()
...
NotImplementedError: base field must be rational numbers
```

is_exact()

Return `True` if elements of this quaternion algebra are represented exactly, i.e. there is no precision loss when doing arithmetic. A quaternion algebra is exact if and only if its base field is exact.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_exact()
True
sage: Q.<i,j,k> = QuaternionAlgebra(Qp(7), -3, -7)
sage: Q.is_exact()
False
```

is_field()

Return `False` always, since all quaternion algebras are noncommutative and all fields are commutative.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_field()
False
```

is_finite()

Return `True` if the quaternion algebra is finite as a set.

Algorithm: A quaternion algebra is finite if and only if the base field is finite.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_finite()
False
sage: Q.<i,j,k> = QuaternionAlgebra(GF(5), -3, -7)
sage: Q.is_finite()
True
```

is_integral_domain()

Return `False` always, since all quaternion algebras are noncommutative and integral domains are commutative (in Sage).

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_integral_domain()
False
```

is_matrix_ring()

Return `True` if the quaternion algebra is isomorphic to the 2x2 matrix ring, and `False` if `self` is a division algebra (i.e. every nonzero element in `self` is invertible).

EXAMPLES:

```
sage: QuaternionAlgebra(QQ,-5,-2).is_matrix_ring()
False
sage: QuaternionAlgebra(1).is_matrix_ring()
True
sage: QuaternionAlgebra(2,9).is_matrix_ring()
True
sage: QuaternionAlgebra(RR(2.),1).is_matrix_ring()
...
NotImplementedError: base field must be rational numbers
```

is_noetherian()

Return `True` always, since any quaternion algebra is a noetherian ring (because it is a finitely generated module over a field).

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.is_noetherian()
True
```

ngens()

Return the number of generators of the quaternion algebra as a K-vector space, not including 1. This value is always 3: the algebra is spanned by the standard basis 1, i, j, k.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ,-5,-2)
sage: Q.ngens()
3
sage: Q.gens()
[i, j, k]
```

order()

Return the number of elements of the quaternion algebra, or `+Infinity` if the algebra is not finite.

EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -3, -7)
sage: Q.order()
+Infinity
sage: Q.<i,j,k> = QuaternionAlgebra(GF(5), -3, -7)
sage: Q.order()
625
```

random_element(**args, **kwds*)

Return a random element of this quaternion algebra.

The `args` and `kwds` are passed to the `random_element` method of the base ring.

EXAMPLES:

```
sage: QuaternionAlgebra(QQ[sqrt(2)],-3,7).random_element()
-4 + (-1/95*sqrt2 - 1/2)*i + (-12*sqrt2 + 1/2)*j + (1/2*sqrt2 - 1)*k
sage: QuaternionAlgebra(-3,19).random_element()
-1/4 + 2/3*i - 5/2*j
sage: QuaternionAlgebra(GF(17)(2),3).random_element()
11 - i + 4*j + 13*k
```

Specify the numerator and denominator bounds:

```
sage: QuaternionAlgebra(-3,19).random_element(10^6,10^6)
-61003/263835 + 222181/103881*i - 7314/2707*j + 458453/129132*k
```

vector_space()

Return the vector space associated to `self` with inner product given by the reduced norm.

EXAMPLES:

```
sage: QuaternionAlgebra(-3,19).vector_space()
Ambient quadratic space of dimension 4 over Rational Field
Inner product matrix:
[   2    0    0    0]
[   0    6    0    0]
[   0    0  -38    0]
[   0    0    0 -114]
```

class **QuaternionFractionalIdeal**(*ring, gens, coerce=True*)

class **QuaternionFractionalIdeal_rational**(*basis, left_order=None, right_order=None, check=True*)

A fractional ideal in a rational quaternion algebra.

basis()

Return basis for this fractional ideal. The basis is in Hermite form.

OUTPUT: tuple

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().unit_ideal().basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)
```

basis_matrix()

Return basis matrix M in Hermite normal form for self as a matrix with rational entries.

If Q is the ambient quaternion algebra, then the $\mathbf{Z}$-span of the rows of M viewed as linear combinations of Q.basis() = $[1, i, j, k]$ is the fractional ideal self. Also, M * M.denominator() is an integer matrix in Hermite normal form.

OUTPUT: matrix over $\mathbf{Q}$

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().unit_ideal().basis_matrix()
[1/2   0 1/2   0]
[  0 1/2   0 1/2]
[  0   0   1   0]
[  0   0   0   1]
```

conjugate()

Return the ideal with generators the conjugates of the generators for self.

OUTPUT: a quaternionic fractional ideal

EXAMPLES:

```
sage: I = BrandtModule(3,5).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 4*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
sage: I.conjugate()
Fractional ideal (2 + 2*j + 28*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
```

free_module()

Return the free module associated to this quaternionic fractional ideal, viewed as a submodule of `Q.free_module()`, where `Q` is the ambient quaternion algebra.

OUTPUT: free $\mathbf{Z}$-module of rank 4 embeded in an ambient $\mathbf{Q}^4$.

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().unit_ideal().basis_matrix()
[1/2   0 1/2   0]
[  0 1/2   0 1/2]
[  0   0   1   0]
[  0   0   0   1]
```

gens()

Return the generators for this ideal, which are the same as the $\mathbf{Z}$-basis for this ideal.

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().unit_ideal().gens()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)
```

gram_matrix()

Return the Gram matrix of this fractional ideal.

OUTPUT: 4x4 matrix over $\mathbf{Q}$.

EXAMPLES:

```
sage: I = BrandtModule(3,5).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 4*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
sage: I.gram_matrix()
[  640  1920  2112  1920]
[ 1920 14080 13440 16320]
[ 2112 13440 13056 15360]
[ 1920 16320 15360 19200]
```

is_equivalent(*I, J, B=10*)

Return `True` if `I` and `J` are equivalent as right ideals.

INPUT:

- `I` – a fractional quaternion ideal (self)
- `J` – a fractional quaternion ideal with same order as `I`
- `B` – a bound to compute and compare theta series before doing the full equivalence test

OUTPUT: bool

EXAMPLES:

```
sage: R = BrandtModule(3,5).right_ideals(); len(R)
2
sage: R[0].is_equivalent(R[1])
False
sage: R[0].is_equivalent(R[0])
True
sage: OO = R[0].quaternion_order()
sage: S = OO.right_ideal([3*a for a in R[0].basis()])
sage: R[0].is_equivalent(S)
True
```

left_order()

Return the left order associated to this fractional ideal.

OUTPUT: an order in a quaternion algebra

EXAMPLES:

```
sage: B = BrandtModule(11)
sage: R = B.maximal_order()
sage: I = R.unit_ideal()
sage: I.left_order()
Order of Quaternion Algebra (-1, -11) with base ring Rational Field with basis (1/2 + 1/2*j,
```

multiply_by_conjugate(*J*)

Return product of self and the conjugate Jbar of J.

INPUT:

- `J` – a quaternion ideal.

OUTPUT: a quaternionic fractional ideal.

EXAMPLES:

```
sage: R = BrandtModule(3,5).right_ideals()
sage: R[0].multiply_by_conjugate(R[1])
Fractional ideal (8 + 8*j + 112*k, 8*i + 16*j + 136*k, 32*j + 128*k, 160*k)
sage: R[0]*R[1].conjugate()
Fractional ideal (8 + 8*j + 112*k, 8*i + 16*j + 136*k, 32*j + 128*k, 160*k)
```

norm()

Return the norm of this fractional ideal.

OUTPUT: rational number

EXAMPLES:

```
sage: C = BrandtModule(37).right_ideals()
sage: [I.norm() for I in C]
[32, 64, 64]
```

quadratic_form()

Return the normalized quadratic form associated to this quaternion ideal.

OUTPUT: quadratic form

EXAMPLES:

```
sage: I = BrandtModule(11).right_ideals()[1]
sage: Q = I.quadratic_form(); Q
Quadratic form in 4 variables over Rational Field with coefficients:
[ 18 22 33 22 ]
[ * 7 22 11 ]
[ * * 22 0 ]
[ * * * 22 ]
sage: Q.theta_series(10)
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + 36*q^8 + 36*q^9 + O(q^10)
sage: I.theta_series(10)
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + 24*q^6 + 24*q^7 + 36*q^8 + 36*q^9 + O(q^10)
```

quaternion_algebra()

Return the ambient quaternion algebra that contains this fractional ideal.

OUTPUT: a quaternion algebra

EXAMPLES:

```
sage: I = BrandtModule(3,5).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 4*k, 2*i + 4*j + 34*k, 8*j + 32*k, 40*k)
sage: I.quaternion_algebra()
Quaternion Algebra (-1, -3) with base ring Rational Field
```

quaternion_order()

Return the order for which this ideal is a left or right fractional ideal. If this ideal has both a left and right ideal structure, then the left order is returned. If it has neither structure, then an error is raised.

OUTPUT: QuaternionOrder

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.unit_ideal().quaternion_order() is R
True
```

right_order()

Return the right order associated to this fractional ideal.

OUTPUT: an order in a quaternion algebra

EXAMPLES:

```
sage: I = BrandtModule(389).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 2*k, i + 2*j + k, 8*j, 8*k)
sage: I.right_order()
Order of Quaternion Algebra (-2, -389) with base ring Rational Field with basis (1/2 + 1/2*j
sage: I.left_order()
...
ValueError: ideal not equipped with left order structure
```

ring()

Return ring that this is a fractional ideal for.

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.unit_ideal().ring() is R
True
```

theta_series(*B*, *var='q'*)

Return normalized theta series of self, as a power series over **Z** in the variable `var`, which is 'q' by default.

The normalized theta series is by definition

$$\theta_I(q) = \sum_{x \in I} q^{\frac{N(x)}{N(I)}}$$

INPUT:

- `B` – positive integer
- `var` – string (default: 'q')

OUTPUT: power series

EXAMPLES:

```
sage: I = BrandtModule(11).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 4*k, 2*i + 4*j + 2*k, 8*j, 8*k)
sage: I.norm()
64
sage: I.theta_series(5)
1 + 12*q^2 + 12*q^3 + 12*q^4 + O(q^5)
sage: I.theta_series(5,'T')
1 + 12*T^2 + 12*T^3 + 12*T^4 + O(T^5)
sage: I.theta_series(3)
1 + 12*q^2 + O(q^3)
```

theta_series_vector(*B*)

Return theta series coefficients of self, as a vector of B integers.

INPUT:

- `B` – positive integer

OUTPUT: vector over **Z** with `B` entries

EXAMPLES:

```
sage: I = BrandtModule(37).right_ideals()[1]; I
Fractional ideal (2 + 6*j + 2*k, i + 2*j + k, 8*j, 8*k)
sage: I.theta_series_vector(5)
(1, 0, 2, 2, 6)
sage: I.theta_series_vector(10)
(1, 0, 2, 2, 6, 4, 8, 6, 10, 10)
sage: I.theta_series_vector(5)
(1, 0, 2, 2, 6)
```

class QuaternionOrder(*A*, *basis*, *check=True*)

An order in a quaternion algebra.

EXAMPLES:

```
sage: QuaternionAlgebra(-1,-7).maximal_order()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis (1/2 + 1/2*j, 1/2*
sage: type(QuaternionAlgebra(-1,-7).maximal_order())
<class 'sage.algebras.quatalg.quaternion_algebra.QuaternionOrder'>
```

basis()

Return fix choice of basis for this quaternion order.

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)
```

discriminant()

Return the discriminant of this order, which we define as $\sqrt{det(Tr(e_i\bar{e_j}))}$, where $\{e_i\}$ is the basis of the order.

OUTPUT: rational number

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().discriminant()
11
sage: S = BrandtModule(11,5).order_of_level_N()
sage: S.discriminant()
55
sage: type(S.discriminant())
<type 'sage.rings.rational.Rational'>
```

free_module()

Return the free $\mathbf{Z}$-module that corresponds to this order inside the vector space corresponding to the ambient quaternion algebra.

OUTPUT: a free $\mathbf{Z}$-module of rank 4

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.basis()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)
sage: R.free_module()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[1/2   0 1/2   0]
[  0 1/2   0 1/2]
[  0   0   1   0]
[  0   0   0   1]
```

gen(n)

Return the n-th generator.

INPUT:

- n - an integer between 0 and 3, inclusive.

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order(); R
Order of Quaternion Algebra (-11, -1) with base ring Rational Field with basis (1/2 + 1/2*j,
sage: R.gen(0)
1/2 + 1/2*j
sage: R.gen(1)
1/2*i + 1/2*k
sage: R.gen(2)
j
sage: R.gen(3)
k
```

gens()

Return generators for self.

EXAMPLES:

```
sage: QuaternionAlgebra(-1,-7).maximal_order().gens()
(1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)
```

intersection(*other*)

Return the intersection of this order with other.

INPUT:

- `other` - a quaternion order in the same ambient quaternion algebra

OUTPUT: a quaternion order

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.intersection(R)
Order of Quaternion Algebra (-11, -1) with base ring Rational Field with basis (1/2 + 1/2*j,
```

We intersect various orders in the quaternion algebra ramified at 11:

```
sage: B = BrandtModule(11,3)
sage: R = B.maximal_order(); S = B.order_of_level_N()
sage: R.intersection(S)
Order of Quaternion Algebra (-1, -11) with base ring Rational Field with basis (1/2 + 1/2*j,
sage: R.intersection(S) == S
True
sage: B = BrandtModule(11,5)
sage: T = B.order_of_level_N()
sage: S.intersection(T)
Order of Quaternion Algebra (-1, -11) with base ring Rational Field with basis (1/2 + 1/2*j,
```

left_ideal(*gens, check=True*)

Return the ideal with given gens over $\mathbf{Z}$.

INPUT:

- `gens` – a list of elements of this quaternion order
- `check` – bool (default: `True`); if `False`, then `gens` must 4-tuple that forms a Hermite basis for an ideal

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.left_ideal([2*a for a in R.basis()])
Fractional ideal (1 + j, i + k, 2*j, 2*k)
```

ngens()

Return the number of generators (which is 4).

EXAMPLES:

```
sage: QuaternionAlgebra(-1,-7).maximal_order().ngens()
4
```

quaternion_algebra()

Return ambient quaternion algebra that contains this quaternion order.

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().quaternion_algebra()
Quaternion Algebra (-11, -1) with base ring Rational Field
```

random_element(**args, **kwds*)

Return a random element of this order.

The args and kwds are passed to the random_element method of the integer ring, and we return an element of the form

$$ae_1 + be_2 + ce_3 + de_4$$

where e_1, ..., e_4 are the basis of this order and a, b, c, d are random integers.

EXAMPLES:

```
sage: QuaternionAlgebra(-11,-1).maximal_order().random_element()
-4 + i - 4*j + k
sage: QuaternionAlgebra(-11,-1).maximal_order().random_element(-10,10)
-9/2 - 7/2*i - 7/2*j + 3/2*k
```

right_ideal(*gens, check=True*)

Return the ideal with given gens over **Z**.

INPUT:

- gens – a list of elements of this quaternion order
- check – bool (default: True); if False, then gens must 4-tuple that forms a Hermite basis for an ideal

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: R.right_ideal([2*a for a in R.basis()])
Fractional ideal (1 + j, i + k, 2*j, 2*k)
```

unit_ideal()

Return the unit ideal in this quaternion order.

EXAMPLES:

```
sage: R = QuaternionAlgebra(-11,-1).maximal_order()
sage: I = R.unit_ideal(); I
Fractional ideal (1/2 + 1/2*j, 1/2*i + 1/2*k, j, k)
```

basis_for_quaternion_lattice(*gens*)

Return a basis for the **Z**-lattice in a quaternion algebra spanned by the given gens.

INPUT:

- gens – list of elements of a single quaternion algebra

EXAMPLES:

```
sage: A.<i,j,k> = QuaternionAlgebra(-1,-7)
sage: sage.algebras.quatalg.quaternion_algebra.basis_for_quaternion_lattice([i+j, i-j, 2*k, A(1/
[1/3, i + j, 2*j, 2*k]
```

is_QuaternionAlgebra(*A*)

Return True if A is of the QuaternionAlgebra data type.

EXAMPLES:

```
sage: sage.algebras.quatalg.quaternion_algebra.is_QuaternionAlgebra(QuaternionAlgebra(QQ,-1,-1))
True
sage: sage.algebras.quatalg.quaternion_algebra.is_QuaternionAlgebra(ZZ)
False
```

unpickle_QuaternionAlgebra_v0(**key*)

The 0th version of pickling for quaternion algebras.

EXAMPLES:

```
sage: Q = QuaternionAlgebra(-5,-19)
sage: f, t = Q.__reduce__()
sage: sage.algebras.quatalg.quaternion_algebra.unpickle_QuaternionAlgebra_v0(*t)
Quaternion Algebra (-5, -19) with base ring Rational Field
sage: loads(dumps(Q)) == Q
True
```

```
sage: loads(dumps(Q)) is Q
True
```

31.2 Elements of Quaternion Algebras

Sage allows for computation with elements of quaternion algebras over a nearly arbitrary base field of characteristic not 2. Sage also has very highly optimized implementation of arithmetic in rational quaternion algebras and quaternion algebras over number fields.

class QuaternionAlgebraElement_abstract()

conjugate()

Return the conjugate of the quaternion: if $\theta = x + yi + zj + wk$, return $x - yi - zj - wk$; that is, return theta.reduced_trace() - theta.

EXAMPLES:

```
sage: A.<i,j,k> = QuaternionAlgebra(QQ,-5,-2)
sage: a = 3*i - j + 2
sage: type(a)
<type 'sage.algebras.quatalg.quaternion_algebra_element.QuaternionAlgebraElement_rational_fi
sage: a.conjugate()
2 - 3*i + j
```

The "universal" test:

```
sage: K.<x,y,z,w,a,b> = QQ[]
sage: Q.<i,j,k> = QuaternionAlgebra(a,b)
sage: theta = x+y*i+z*j+w*k
sage: theta.conjugate()
x + (-y)*i + (-z)*j + (-w)*k
```

is_constant()

Return True if this quaternion is constant, i.e., has no i, j, or k term.

OUTPUT: bool

EXAMPLES:

```
sage: A.<i,j,k>=QuaternionAlgebra(-1,-2)
sage: A(1).is_constant()
True
sage: A(1+i).is_constant()
False
sage: A(i).is_constant()
False
```

reduced_characteristic_polynomial()

Return the reduced characteristic polynomial of this quaternion algebra element, which is $X^2 - tX + n$, where t is the reduced trace and n is the reduced norm.

INPUT: • var – string (default: 'x'); indeterminate of characteristic polynomial

EXAMPLES:

```
sage: A.<i,j,k>=QuaternionAlgebra(-1,-2)
sage: i.reduced_characteristic_polynomial()
x^2 + 1
sage: j.reduced_characteristic_polynomial()
x^2 + 2
```

```
sage: (i+j).reduced_characteristic_polynomial()
x^2 + 3
sage: (2+j+k).reduced_trace()
4
sage: (2+j+k).reduced_characteristic_polynomial('T')
T^2 - 4*T + 8
```

reduced_norm()

Return the reduced norm of self: if $\theta = x+yi+zj+wk$, then θ has reduced norm $x^2 - ay^2 - bz^2 + abw^2$.

EXAMPLES:

```
sage: K.<x,y,z,w,a,b> = QQ[]
sage: Q.<i,j,k> = QuaternionAlgebra(a,b)
sage: theta = x+y*i+z*j+w*k
sage: theta.reduced_norm()
w^2*a*b - y^2*a - z^2*b + x^2
```

reduced_trace()

Return the reduced trace of self: if $\theta = x + yi + zj + wk$, then θ has reduced trace $2x$.

EXAMPLES:

```
sage: K.<x,y,z,w,a,b> = QQ[]
sage: Q.<i,j,k> = QuaternionAlgebra(a,b)
sage: theta = x+y*i+z*j+w*k
sage: theta.reduced_trace()
2*x
```

class **QuaternionAlgebraElement_generic**()

TESTS:

We test pickling:

```
sage: R.<x> = Frac(QQ['x']); Q.<i,j,k> = QuaternionAlgebra(R,-5*x,-2)
sage: theta = x + i*x^3 + j*x^2 + k*x
sage: theta == loads(dumps(theta))
True
```

class **QuaternionAlgebraElement_number_field**()

class **QuaternionAlgebraElement_rational_field**()

TESTS:

We test pickling:

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ,-5,-2)
sage: i + j + k == loads(dumps(i+j+k))
True
```

coefficient_tuple()

Return 4-tuple of rational numbers which are the coefficients of this quaternion.

EXAMPLES:

```
sage: A.<i,j,k>=QuaternionAlgebra(-1,-2)
sage: (2/3 + 3/5*i + 4/3*j - 5/7*k).coefficient_tuple()
(2/3, 3/5, 4/3, -5/7)
```

conjugate()

Return the conjugate of this quaternion.

EXAMPLES:

```
sage: A.<i,j,k> = QuaternionAlgebra(QQ,-5,-2)
sage: a = 3*i - j + 2
sage: type(a)
<type 'sage.algebras.quatalg.quaternion_algebra_element.QuaternionAlgebraElement_rational_fi
sage: a.conjugate()
2 - 3*i + j
sage: b = 1 + 1/3*i + 1/5*j - 1/7*k
sage: b.conjugate()
1 - 1/3*i - 1/5*j + 1/7*k
```

denominator()

Return the lowest common multiple of the denominators of the coefficients of i, j and k for this quaternion.

EXAMPLES:

```
sage: A = QuaternionAlgebra(QQ, -1, -1)
sage: A.<i,j,k> = QuaternionAlgebra(QQ, -1, -1)
sage: a = (1/2) + (1/5)*i + (5/12)*j + (1/13)*k
sage: a
1/2 + 1/5*i + 5/12*j + 1/13*k
sage: a.denominator()
780
sage: lcm([2, 5, 12, 13])
780
sage: (a * a).denominator()
608400
sage: (a + a).denominator()
390
```

denominator_and_integer_coefficient_tuple()

Return 5-tuple d, x, y, z, w, where this rational quaternion is equal to $(x + yi + zj + wk)/d$ and x, y, z, w do not share a common factor with d.

OUTPUT: 5-tuple of Integers

EXAMPLES:

```
sage: A.<i,j,k>=QuaternionAlgebra(-1,-2)
sage: (2 + 3*i + 4/3*j - 5*k).denominator_and_integer_coefficient_tuple()
(3, 6, 9, 4, -15)
```

integer_coefficient_tuple()

Returns integer part of this quaternion, ignoring the common denominator.

OUTPUT: 4-tuple of Integers

EXAMPLES:

```
sage: A.<i,j,k>=QuaternionAlgebra(-1,-2)
sage: (2 + 3*i + 4/3*j - 5*k).integer_coefficient_tuple()
(6, 9, 4, -15)
```

is_constant()

Return True if this quaternion is constant, i.e., has no i, j, or k term.

OUTPUT: bool

EXAMPLES:

```
sage: A.<i,j,k>=QuaternionAlgebra(-1,-2)
sage: A(1/3).is_constant()
True
sage: A(-1).is_constant()
True
```

```
sage: (1+i).is_constant()
False
sage: j.is_constant()
False
```

reduced_norm()
Return the reduced norm of self. Given a quaternion $x + yi + zj + wk$, this is $x^2 - ay^2 - bz^2 + abw^2$.
EXAMPLES:

```
sage: K.<i,j,k> = QuaternionAlgebra(QQ, -5, -2)
sage: i.reduced_norm()
5
sage: j.reduced_norm()
2
sage: a = 1/3 + 1/5*i + 1/7*j + k
sage: a.reduced_norm()
22826/2205
```

reduced_trace()
Return the reduced trace of self, which is $2x$ if self is $x + iy + zj + wk$.
EXAMPLES:

```
sage: K.<i,j,k> = QuaternionAlgebra(QQ, -5, -2)
sage: i.reduced_trace()
0
sage: j.reduced_trace()
0
sage: a = 1/3 + 1/5*i + 1/7*j + k
sage: a.reduced_trace()
2/3
```

clear_mpz_globals()

gmp_randrange()

init_mpz_globals()

unpickle_QuaternionAlgebraElement_generic_v0()
EXAMPLES:

```
sage: K.<X> = QQ[]
sage: Q.<i,j,k> = QuaternionAlgebra(Frac(K), -5,-19); z = 2/3 + i*X - X^2*j + X^3*k
sage: f, t = z.__reduce__()
sage: sage.algebras.quatalg.quaternion_algebra_element.unpickle_QuaternionAlgebraElement_generic
2/3 + X*i + (-X^2)*j + X^3*k
sage: sage.algebras.quatalg.quaternion_algebra_element.unpickle_QuaternionAlgebraElement_generic
True
```

unpickle_QuaternionAlgebraElement_number_field_v0()
EXAMPLES:

```
sage: K.<a> = QQ[2^(1/3)]; Q.<i,j,k> = QuaternionAlgebra(K, -3, a); z = i + j
sage: f, t = z.__reduce__()
sage: sage.algebras.quatalg.quaternion_algebra_element.unpickle_QuaternionAlgebraElement_number_
i + j
sage: sage.algebras.quatalg.quaternion_algebra_element.unpickle_QuaternionAlgebraElement_number_
True
```

unpickle_QuaternionAlgebraElement_rational_field_v0()
EXAMPLES:

```
sage: Q.<i,j,k> = QuaternionAlgebra(-5,-19); a = 2/3 + i*5/7 - j*2/5 +19/2
sage: f, t = a.__reduce__()
sage: sage.algebras.quatalg.quaternion_algebra_element.unpickle_QuaternionAlgebraElement_rationa
61/6 + 5/7*i - 2/5*j
```

CHAPTER

THIRTYTWO

MATRICES AND SPACES OF MATRICES

Sage provides native support for working with matrices over any commutative or noncommutative ring. The parent object for a matrix is a matrix space `MatrixSpace(R, n, m)` of all $n \times m$ matrices over a ring R.

To create a matrix, either use the `matrix(...)` function or create a matrix space using the `MatrixSpace` command and coerce an object into it.

Matrices also act on row vectors, which you create using the `vector(...)` command or by making a `VectorSpace` and coercing lists into it. The natural action of matrices on row vectors is from the right. Sage currently does not have a column vector class (on which matrices would act from the left), but this is planned.

In addition to native Sage matrices, Sage also includes the following additional ways to compute with matrices:

- Several math software systems included with Sage have their own native matrix support, which can be used from Sage. E.g., PARI, GAP, Maxima, and Singular all have a notion of matrices.
- The GSL C-library is included with Sage, and can be used via Cython.
- The `scipy` module provides support for *sparse* numerical linear algebra, among many other things.
- The `numpy` module, which you load by typing `import numpy` is included standard with Sage. It contains a very sophisticated and well developed array class, plus optimized support for *numerical linear algebra*. Sage's matrices over RDF and CDF (native floating-point real and complex numbers) use numpy.

32.1 Matrix Spaces.

You can create any space $\mathrm{Mat}_{n\times m}(R)$ of either dense or sparse matrices with given number of rows and columns over any commutative or noncommutative ring.

EXAMPLES:

```
sage: MS = MatrixSpace(QQ,6,6,sparse=True); MS
Full MatrixSpace of 6 by 6 sparse matrices over Rational Field
sage: MS.base_ring()
Rational Field
sage: MS = MatrixSpace(ZZ,3,5,sparse=False); MS
Full MatrixSpace of 3 by 5 dense matrices over Integer Ring
```

TESTS:

```
sage: matrix(RR,2,2,sparse=True)
[0.000000000000000 0.000000000000000]
[0.000000000000000 0.000000000000000]
```

```
sage: matrix(GF(11),2,2,sparse=True)
[0 0]
[0 0]
```

MatrixSpace(*base_ring, nrows, ncols=None, sparse=False*)

Create with the command

MatrixSpace(base_ring , nrows [, ncols] [, sparse])

The default value of the optional argument sparse is False. The default value of the optional argument ncols is nrows.

INPUT:

- `base_ring` - a ring
- `nrows` - int, the number of rows
- `ncols` - (default nrows) int, the number of columns
- `sparse` - (default false) whether or not matrices are given a sparse representation

OUTPUT: The unique space of all nrows x ncols matrices over base_ring.

EXAMPLES:

```
sage: MS = MatrixSpace(RationalField(),2)
sage: MS.base_ring()
Rational Field
sage: MS.dimension()
4
sage: MS.dims()
(2, 2)
sage: B = MS.basis()
sage: B
[
[1 0]
[0 0],
[0 1]
[0 0],
[0 0]
[1 0],
[0 0]
[0 1]
]
sage: B[0]
[1 0]
[0 0]
sage: B[1]
[0 1]
[0 0]
sage: B[2]
[0 0]
[1 0]
sage: B[3]
[0 0]
[0 1]
sage: A = MS.matrix([1,2,3,4])
sage: A
[1 2]
[3 4]
sage: MS2 = MatrixSpace(RationalField(),2,3)
```

```
sage: B = MS2.matrix([1,2,3,4,5,6])
sage: A*B
[ 9 12 15]
[19 26 33]


sage: M = MatrixSpace(ZZ, 10)
sage: M
Full MatrixSpace of 10 by 10 dense matrices over Integer Ring
sage: loads(M.dumps()) == M
True
```

class **MatrixSpace_generic**(*base_ring, nrows, ncols=None, sparse=False*)

The space of all nrows x ncols matrices over base_ring.

EXAMPLES:

```
sage: MatrixSpace(ZZ,10,5)
Full MatrixSpace of 10 by 5 dense matrices over Integer Ring
sage: MatrixSpace(ZZ,10,2^31)
...                                                                    # 32-bit
ValueError: number of rows and columns must be less than 2^31 (on a 32-bit computer -- use a 64-
Full MatrixSpace of 10 by 2147483648 dense matrices over Integer Ring   # 64-bit
```

base_extend(*R*)

Return base extension of this matrix space to R.

INPUT:

- R - ring

OUTPUT: a matrix space

EXAMPLES:

```
sage: Mat(ZZ,3,5).base_extend(QQ)
Full MatrixSpace of 3 by 5 dense matrices over Rational Field
sage: Mat(QQ,3,5).base_extend(GF(7))
...
TypeError: no base extension defined
```

basis()

Returns a basis for this matrix space.

> **Warning:** This will of course compute every generator of this matrix space. So for large matrices, this could take a long time, waste a massive amount of memory (for dense matrices), and is likely not very useful. Don't use this on large matrix spaces.

EXAMPLES:

```
sage: Mat(ZZ,2,2).basis()
[
[1 0]
[0 0],
[0 1]
[0 0],
[0 0]
[1 0],
[0 0]
[0 1]
]
```

change_ring(*R*)

Return matrix space over R with otherwise same parameters as self.

INPUT:

- R - ring

OUTPUT: a matrix space

EXAMPLES:

```
sage: Mat(QQ,3,5).change_ring(GF(7))
Full MatrixSpace of 3 by 5 dense matrices over Finite Field of size 7
```

column_space()

Return the module spanned by all columns of matrices in this matrix space. This is a free module of rank the number of columns. It will be sparse or dense as this matrix space is sparse or dense.

EXAMPLES:

```
sage: M = Mat(GF(9,'a'),20,5,sparse=True); M.column_space()
Sparse vector space of dimension 20 over Finite Field in a of size 3^2
```

construction()

EXAMPLES:

```
sage: A = matrix(ZZ, 2, [1..4], sparse=True)
sage: A.parent().construction()
(MatrixFunctor, Integer Ring)
sage: A.parent().construction()[0](QQ['x'])
Full MatrixSpace of 2 by 2 sparse matrices over Univariate Polynomial Ring in x over Rationa
sage: parent(A/2)
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
```

dimension()

Returns (m rows) * (n cols) of self as Integer

EXAMPLES:

```
sage: MS = MatrixSpace(ZZ,4,6)
sage: u = MS.dimension()
sage: u - 24 == 0
True
```

dims()

Returns (m row, n col) representation of self dimension

EXAMPLES:

```
sage: MS = MatrixSpace(ZZ,4,6)
sage: MS.dims()
(4, 6)
```

gen(*n*)

Return the n-th generator of this matrix space.

This doesn't compute all basis matrices, so it is reasonably intelligent.

EXAMPLES:

```
sage: M = Mat(GF(7),10000,5); M.ngens()
50000
sage: a = M.10
sage: a[:4]
[0 0 0 0 0]
[0 0 0 0 0]
[1 0 0 0 0]
[0 0 0 0 0]
```

get_action_impl(*S, op, self_on_left*)

identity_matrix()

Create an identity matrix in self. (Must be a space of square matrices).

EXAMPLES:

```
sage: MS1 = MatrixSpace(ZZ,4)
sage: MS2 = MatrixSpace(QQ,3,4)
sage: I = MS1.identity_matrix()
sage: I
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: Er = MS2.identity_matrix()
...
TypeError: self must be a space of square matrices
```

is_dense()

Returns True if matrices in self are dense and False otherwise.

EXAMPLES:

```
sage: Mat(RDF,2,3).is_sparse()
False
sage: Mat(RR,123456,22,sparse=True).is_sparse()
True
```

is_finite()

EXAMPLES:

```
sage: MatrixSpace(GF(101), 10000).is_finite()
True
sage: MatrixSpace(QQ, 2).is_finite()
False
```

is_sparse()

Returns True if matrices in self are sparse and False otherwise.

EXAMPLES:

```
sage: Mat(GF(2011),10000).is_sparse()
False
sage: Mat(GF(2011),10000,sparse=True).is_sparse()
True
```

matrix(*x=0, coerce=True, copy=True, rows=True*)

Create a matrix in self. The entries can be specified either as a single list of length nrows*ncols, or as a list of lists.

EXAMPLES:

```
sage: M = MatrixSpace(ZZ, 2)
sage: M.matrix([[1,0],[0,-1]])
[ 1  0]
[ 0 -1]
sage: M.matrix([1,0,0,-1])
[ 1  0]
[ 0 -1]
sage: M.matrix([1,2,3,4])
[1 2]
[3 4]
sage: M.matrix([1,2,3,4],rows=False)
```

```
[1 3]
[2 4]
```

matrix_space(*nrows=None, ncols=None, sparse=False*)

Return the matrix space with given number of rows, columns and sparcity over the same base ring as self, and defaults the same as self.

EXAMPLES:

```
sage: M = Mat(GF(7),100,200)
sage: M.matrix_space(5000)
Full MatrixSpace of 5000 by 200 dense matrices over Finite Field of size 7
sage: M.matrix_space(ncols=5000)
Full MatrixSpace of 100 by 5000 dense matrices over Finite Field of size 7
sage: M.matrix_space(sparse=True)
Full MatrixSpace of 100 by 200 sparse matrices over Finite Field of size 7
```

ncols()

Return the number of columns of matrices in this space.

EXAMPLES:

```
sage: M = Mat(ZZ['x'],200000,500000,sparse=True)
sage: M.ncols()
500000
```

ngens()

Return the number of generators of this matrix space, which is the number of entries in the matrices in this space.

EXAMPLES:

```
sage: M = Mat(GF(7),100,200); M.ngens()
20000
```

nrows()

Return the number of rows of matrices in this space.

EXAMPLES:

```
sage: M = Mat(ZZ,200000,500000)
sage: M.nrows()
200000
```

random_element(*density=1, *args, **kwds*)

INPUT:

- `density` - integer (default: 1) rough measure of the proportion of nonzero entries in the random matrix
- `*args, **kwds` - rest of parameters may be passed to the random_element function of the base ring. ("may be", since this function calls the randomize function on the zero matrix, which need not call the random_element function of the base ring at all in general.)

EXAMPLES:

```
sage: Mat(ZZ,2,5).random_element()
[ -8   2   0   0   1]
[ -1   2   1 -95  -1]
sage: Mat(QQ,2,5).random_element(density=0.5)
[  2   0   0   0   1]
[  0   0   0 1/2   0]
sage: Mat(QQ,3,sparse=True).random_element()
[  -1  1/3    1]
[   0   -1    0]
```

```
[  -1    1 -1/4]
sage: Mat(GF(9,'a'),3,sparse=True).random_element()
[      1       2       1]
[2*a + 1       a       2]
[      2 2*a + 2       1]
```

row_space()

Return the module spanned by all rows of matrices in this matrix space. This is a free module of rank the number of rows. It will be sparse or dense as this matrix space is sparse or dense.

EXAMPLES:

```
sage: M = Mat(ZZ,20,5,sparse=False); M.row_space()
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

zero_matrix()

Return the zero matrix.

dict_to_list(*entries, nrows, ncols*)

Given a dictionary of coordinate tuples, return the list given by reading off the nrows*ncols matrix in row order.

EXAMLES:

```
sage: from sage.matrix.matrix_space import dict_to_list
sage: d = {}
sage: d[(0,0)] = 1
sage: d[(1,1)] = 2
sage: dict_to_list(d, 2, 2)
[1, 0, 0, 2]
sage: dict_to_list(d, 2, 3)
[1, 0, 0, 0, 2, 0]
```

is_MatrixSpace(*x*)

Returns True if self is an instance of MatrixSpace returns false if self is not an instance of MatrixSpace

EXAMPLES:

```
sage: from sage.matrix.matrix_space import is_MatrixSpace
sage: MS = MatrixSpace(QQ,2)
sage: A = MS.random_element()
sage: is_MatrixSpace(MS)
True
sage: is_MatrixSpace(A)
False
sage: is_MatrixSpace(5)
False
```

list_to_dict(*entries, nrows, ncols, rows=True*)

Given a list of entries, create a dictionary whose keys are coordinate tuples and values are the entries.

EXAMPLES:

```
sage: from sage.matrix.matrix_space import list_to_dict
sage: d = list_to_dict([1,2,3,4],2,2)
sage: d[(0,1)]
2
sage: d = list_to_dict([1,2,3,4],2,2,rows=False)
sage: d[(0,1)]
3
```

test_trivial_matrices_inverse(*ring, sparse=True, checkrank=True*)

Tests inversion, determinant and is_inverstible for trivial matrices.

This function is a helper to check that the inversion of trivial matrices (of size 0x0, nx0, 0xn or 1x1) is handled consistently by the various implementation of matrices. The coherency is checked through a bunch of assertions. If an inconsistency is found, an AssertionError is raised which should make clear what is the problem.

INPUT:

- `ring` - a ring
- `sparse` - a boolean
- `checkrank` - a boolean

OUTPUT:

- nothing if everything is correct otherwise raise an AssertionError

The methods determinant, is_invertible, rank and inverse are checked for

- the 0x0 empty identity matrix
- the 0x3 and 3x0 matrices
- the 1x1 null matrix [0]
- the 1x1 identity matrix [1]

If `checkrank` is `False` then the rank is not checked. This is used the check matrix over ring where echelon form is not implemented.

TODO: must be adapted to category check framework when ready (see trac #5274).

TESTS:

```
sage: from sage.matrix.matrix_space import test_trivial_matrices_inverse as tinv
sage: tinv(ZZ, sparse=True)
sage: tinv(ZZ, sparse=False)
sage: tinv(QQ, sparse=True)
sage: tinv(QQ, sparse=False)
sage: tinv(GF(11), sparse=True)
sage: tinv(GF(11), sparse=False)
sage: tinv(GF(2), sparse=True)
sage: tinv(GF(2), sparse=False)
sage: tinv(SR, sparse=True)
sage: tinv(SR, sparse=False)
sage: tinv(RDF, sparse=True)
sage: tinv(RDF, sparse=False)
sage: tinv(CDF, sparse=True)
sage: tinv(CDF, sparse=False)
sage: tinv(CyclotomicField(7), sparse=True)
sage: tinv(CyclotomicField(7), sparse=False)
```

TODO: As soon as rank of sparse matrix over QQ['x,y'] is implemented, please remove the following test and the `checkrank=False` in the next one:

sage: MatrixSpace(QQ['x,y'], 3, 3, sparse=True)(1).rank() Traceback (most recent call last): ... NotImplementedError: echelon form over Multivariate Polynomial Ring in x, y over Rational Field not yet implemented sage: tinv(QQ['x,y'], sparse=True, checkrank=False)

TODO: As soon as rank of dense matrix over QQ['x,y'] is implemented, please remove the following test and the `checkrank=False` in the next one:

sage: MatrixSpace(QQ['x,y'], 3, 3, sparse=False)(1).rank() Traceback (most recent call last): ... RuntimeError: BUG: matrix pivots should have been set but weren't, matrix parent = 'Full MatrixSpace of 3 by 3 dense matrices over Multivariate Polynomial Ring in x, y over Rational Field'

sage: tinv(QQ['x,y'], sparse=False, checkrank=False)

32.2 Matrix Constructor.

Matrix(**args, **kwds*)

Create a matrix.

INPUT: The matrix command takes the entries of a matrix, optionally preceded by a ring and the dimensions of the matrix, and returns a matrix.

The entries of a matrix can be specified as a flat list of elements, a list of lists (i.e., a list of rows), a list of Sage vectors, or a dictionary having positions as keys and matrix entries as values (see the examples). You can create a matrix of zeros by passing an empty list or the integer zero for the entries. To construct a multiple of the identity (cI), you can specify square dimensions and pass in c. Calling matrix() with a Sage object may return something that makes sense. Calling matrix() with a numpy array will convert the array to a matrix.

The ring, number of rows, and number of columns of the matrix can be specified by setting the ring, nrows, or ncols parameters or by passing them as the first arguments to the function in the order ring, nrows, ncols. The ring defaults to ZZ if it is not specified or cannot be determined from the entries. If the numbers of rows and columns are not specified and cannot be determined, then an empty 0x0 matrix is returned.

- `ring` - the base ring for the entries of the matrix.
- `nrows` - the number of rows in the matrix.
- `ncols` - the number of columns in the matrix.
- `sparse` - create a sparse matrix. This defaults to True when the entries are given as a dictionary, otherwise defaults to False.

OUTPUT:

a matrix

EXAMPLES:

```
sage: m=matrix(2); m; m.parent()
[0 0]
[0 0]
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring

sage: m=matrix(2,3); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring

sage: m=matrix(QQ,[[1,2,3],[4,5,6]]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field

sage: v1=vector((1,2,3))
sage: v2=vector((4,5,6))
sage: m=matrix([v1,v2]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring

sage: m=matrix(QQ,2,[1,2,3,4,5,6]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
```

```
sage: m=matrix(QQ,2,3,[1,2,3,4,5,6]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field

sage: m=matrix({(0,1): 2, (1,1):2/5}); m; m.parent()
[  0   2]
[  0 2/5]
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field

sage: m=matrix(QQ,2,3,{(1,1): 2}); m; m.parent()
[0 0 0]
[0 2 0]
Full MatrixSpace of 2 by 3 sparse matrices over Rational Field

sage: import numpy
sage: n=numpy.array([[1,2],[3,4]],float)
sage: m=matrix(n); m; m.parent()
[1.0 2.0]
[3.0 4.0]
Full MatrixSpace of 2 by 2 dense matrices over Real Double Field

sage: v = vector(ZZ, [1, 10, 100])
sage: m=matrix(v); m; m.parent()
[  1  10 100]
Full MatrixSpace of 1 by 3 dense matrices over Integer Ring
sage: m=matrix(GF(7), v); m; m.parent()
[1 3 2]
Full MatrixSpace of 1 by 3 dense matrices over Finite Field of size 7

sage: g = graphs.PetersenGraph()
sage: m = matrix(g); m; m.parent()
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
Full MatrixSpace of 10 by 10 dense matrices over Integer Ring

sage: matrix(ZZ, 10, 10, range(100), sparse=True).parent()
Full MatrixSpace of 10 by 10 sparse matrices over Integer Ring

sage: R = PolynomialRing(QQ, 9, 'x')
sage: A = matrix(R, 3, 3, R.gens()); A
[x0 x1 x2]
[x3 x4 x5]
[x6 x7 x8]
sage: det(A)
-x2*x4*x6 + x1*x5*x6 + x2*x3*x7 - x0*x5*x7 - x1*x3*x8 + x0*x4*x8
```

TESTS:

```
sage: m=matrix(); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
sage: m=matrix(QQ); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Rational Field
sage: m=matrix(QQ,2); m; m.parent()
[0 0]
[0 0]
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: m=matrix(QQ,2,3); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
sage: m=matrix([]); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
sage: m=matrix(QQ,[]); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Rational Field
sage: m=matrix(2,2,1); m; m.parent()
[1 0]
[0 1]
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: m=matrix(QQ,2,2,1); m; m.parent()
[1 0]
[0 1]
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: m=matrix(2,3,0); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: m=matrix(QQ,2,3,0); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
sage: m=matrix([[1,2,3],[4,5,6]]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: m=matrix(QQ,2,[[1,2,3],[4,5,6]]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
sage: m=matrix(QQ,3,[[1,2,3],[4,5,6]]); m; m.parent()
...
ValueError: Number of rows does not match up with specified number.
sage: m=matrix(QQ,2,3,[[1,2,3],[4,5,6]]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
sage: m=matrix(QQ,2,4,[[1,2,3],[4,5,6]]); m; m.parent()
...
ValueError: Number of columns does not match up with specified number.
sage: m=matrix([(1,2,3),(4,5,6)]); m; m.parent()
[1 2 3]
[4 5 6]
```

```
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: m=matrix([1,2,3,4,5,6]); m; m.parent()
[1 2 3 4 5 6]
Full MatrixSpace of 1 by 6 dense matrices over Integer Ring
sage: m=matrix((1,2,3,4,5,6)); m; m.parent()
[1 2 3 4 5 6]
Full MatrixSpace of 1 by 6 dense matrices over Integer Ring
sage: m=matrix(QQ,[1,2,3,4,5,6]); m; m.parent()
[1 2 3 4 5 6]
Full MatrixSpace of 1 by 6 dense matrices over Rational Field
sage: m=matrix(QQ,3,2,[1,2,3,4,5,6]); m; m.parent()
[1 2]
[3 4]
[5 6]
Full MatrixSpace of 3 by 2 dense matrices over Rational Field
sage: m=matrix(QQ,2,4,[1,2,3,4,5,6]); m; m.parent()
...
ValueError: entries has the wrong length
sage: m=matrix(QQ,5,[1,2,3,4,5,6]); m; m.parent()
...
TypeError: entries has the wrong length
sage: m=matrix({(1,1): 2}); m; m.parent()
[0 0]
[0 2]
Full MatrixSpace of 2 by 2 sparse matrices over Integer Ring
sage: m=matrix(QQ,{(1,1): 2}); m; m.parent()
[0 0]
[0 2]
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
sage: m=matrix(QQ,3,{(1,1): 2}); m; m.parent()
[0 0 0]
[0 2 0]
[0 0 0]
Full MatrixSpace of 3 by 3 sparse matrices over Rational Field
sage: m=matrix(QQ,3,4,{(1,1): 2}); m; m.parent()
[0 0 0 0]
[0 2 0 0]
[0 0 0 0]
Full MatrixSpace of 3 by 4 sparse matrices over Rational Field
sage: m=matrix(QQ,2,{(1,1): 2}); m; m.parent()
[0 0]
[0 2]
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
sage: m=matrix(QQ,1,{(1,1): 2}); m; m.parent()
...
IndexError: invalid entries list
sage: m=matrix({}); m; m.parent()
[]
Full MatrixSpace of 0 by 0 sparse matrices over Integer Ring
sage: m=matrix(QQ,{}); m; m.parent()
[]
Full MatrixSpace of 0 by 0 sparse matrices over Rational Field
sage: m=matrix(QQ,2,{}); m; m.parent()
[0 0]
[0 0]
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
sage: m=matrix(QQ,2,3,{}); m; m.parent()
[0 0 0]
```

```
[0 0 0]
Full MatrixSpace of 2 by 3 sparse matrices over Rational Field
sage: m=matrix(2,{}); m; m.parent()
[0 0]
[0 0]
Full MatrixSpace of 2 by 2 sparse matrices over Integer Ring
sage: m=matrix(2,3,{}); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 sparse matrices over Integer Ring
sage: m=matrix(0); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
sage: m=matrix(0,2); m; m.parent()
[]
Full MatrixSpace of 0 by 2 dense matrices over Integer Ring
sage: m=matrix(2,0); m; m.parent()
[]
Full MatrixSpace of 2 by 0 dense matrices over Integer Ring
sage: m=matrix(0,[1]); m; m.parent()
...
ValueError: entries has the wrong length
sage: m=matrix(1,0,[]); m; m.parent()
[]
Full MatrixSpace of 1 by 0 dense matrices over Integer Ring
sage: m=matrix(0,1,[]); m; m.parent()
[]
Full MatrixSpace of 0 by 1 dense matrices over Integer Ring
sage: m=matrix(0,[]); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
sage: m=matrix(0,{}); m; m.parent()
[]
Full MatrixSpace of 0 by 0 sparse matrices over Integer Ring
sage: m=matrix(0,{(1,1):2}); m; m.parent()
...
IndexError: invalid entries list
sage: m=matrix(2,0,{(1,1):2}); m; m.parent()
...
IndexError: invalid entries list
sage: import numpy
sage: n=numpy.array([[numpy.complex(0,1),numpy.complex(0,2)],[3,4]],complex)
sage: m=matrix(n); m; m.parent()
[1.0*I 2.0*I]
[  3.0   4.0]
Full MatrixSpace of 2 by 2 dense matrices over Complex Double Field
sage: n=numpy.array([[1,2],[3,4]],'int32')
sage: m=matrix(n); m; m.parent()
[1 2]
[3 4]
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: n = numpy.array([[1,2,3],[4,5,6],[7,8,9]],'float32')
sage: m=matrix(n); m; m.parent()
[1.0 2.0 3.0]
[4.0 5.0 6.0]
[7.0 8.0 9.0]
Full MatrixSpace of 3 by 3 dense matrices over Real Double Field
sage: n=numpy.array([[1,2,3],[4,5,6],[7,8,9]],'float64')
```

```
sage: m=matrix(n); m; m.parent()
[1.0 2.0 3.0]
[4.0 5.0 6.0]
[7.0 8.0 9.0]
Full MatrixSpace of 3 by 3 dense matrices over Real Double Field
sage: n=numpy.array([[1,2,3],[4,5,6],[7,8,9]],'complex64')
sage: m=matrix(n); m; m.parent()
[1.0 2.0 3.0]
[4.0 5.0 6.0]
[7.0 8.0 9.0]
Full MatrixSpace of 3 by 3 dense matrices over Complex Double Field
sage: n=numpy.array([[1,2,3],[4,5,6],[7,8,9]],'complex128')
sage: m=matrix(n); m; m.parent()
[1.0 2.0 3.0]
[4.0 5.0 6.0]
[7.0 8.0 9.0]
Full MatrixSpace of 3 by 3 dense matrices over Complex Double Field
sage: a = matrix([[1,2],[3,4]])
sage: b = matrix(a.numpy()); b
[1 2]
[3 4]
sage: a == b
True
sage: c = matrix(a.numpy('float32')); c
[1.0 2.0]
[3.0 4.0]
sage: v = vector(ZZ, [1, 10, 100])
sage: m=matrix(ZZ['x'], v); m; m.parent()
[  1  10 100]
Full MatrixSpace of 1 by 3 dense matrices over Univariate Polynomial Ring in x over Integer Ring
sage: matrix(ZZ, 10, 10, range(100)).parent()
Full MatrixSpace of 10 by 10 dense matrices over Integer Ring
sage: m = matrix(GF(7), [[1/3,2/3,1/2], [3/4,4/5,7]]); m; m.parent()
[5 3 4]
[6 5 0]
Full MatrixSpace of 2 by 3 dense matrices over Finite Field of size 7
sage: m = matrix([[1,2,3], [RDF(2), CDF(1,2), 3]]); m; m.parent()
[        1.0         2.0         3.0]
[        2.0 1.0 + 2.0*I         3.0]
Full MatrixSpace of 2 by 3 dense matrices over Complex Double Field
sage: m=matrix(3,3,1/2); m; m.parent()
[1/2   0   0]
[  0 1/2   0]
[  0   0 1/2]
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
sage: matrix([[1],[2,3]])
...
ValueError: List of rows is not valid (rows are wrong types or lengths)
sage: matrix([[1],2])
...
ValueError: List of rows is not valid (rows are wrong types or lengths)
sage: matrix(vector(RR,[1,2,3])).parent()
Full MatrixSpace of 1 by 3 dense matrices over Real Field with 53 bits of precision
```

AUTHORS:

- ??: Initial implementation
- Jason Grout (2008-03): almost a complete rewrite, with bits and pieces from the original implementation

block_diagonal_matrix(*sub_matrices, **kwds)

Create a block matrix whose diagonal block entries are given by sub_matrices, with zero elsewhere.

See also `block_matrix`.

EXAMPLES:

```
sage: A = matrix(ZZ, 2, [1,2,3,4])
sage: block_diagonal_matrix(A, A)
[1 2|0 0]
[3 4|0 0]
[---+---]
[0 0|1 2]
[0 0|3 4]
```

The sub-matrices need not be square:

```
sage: B = matrix(QQ, 2, 3, range(6))
sage: block_diagonal_matrix(~A, B)
[  -2    1|   0    0    0]
[ 3/2 -1/2|   0    0    0]
[---------+--------------]
[   0    0|   0    1    2]
[   0    0|   3    4    5]
```

block_matrix(*sub_matrices, nrows=None, ncols=None, subdivide=True*)

Returns a larger matrix made by concatinating the sub_matrices (rows first, then columns). For example, the matrix

```
[ A B ]
[ C D ]
```

is made up of submatrices A, B, C, and D.

INPUT:

- `sub_matrices` - matrices (must be of the correct size, or constants)
- `nrows` - (optional) the number of block rows
- `ncols` - (optional) the number of block cols
- `subdivide` - boolean, whether or not to add subdivision information to the matrix

EXAMPLES:

```
sage: A = matrix(QQ, 2, 2, [3,9,6,10])
sage: block_matrix([A, -A, ~A, 100*A])
[    3     9|   -3    -9]
[    6    10|   -6   -10]
[-----------+-----------]
[-5/12   3/8|  300   900]
[  1/4  -1/8|  600  1000]
```

One can use constant entries:

```
sage: block_matrix([1, A, 0, 1])
[ 1  0| 3  9]
[ 0  1| 6 10]
[-----+-----]
[ 0  0| 1  0]
[ 0  0| 0  1]
```

One can specify the number of rows or columns (optional for square number of matrices):

```
sage: block_matrix([A, -A, ~A, 100*A], ncols=4)
[    3     9|   -3    -9|-5/12   3/8|  300   900]
[    6    10|   -6   -10|  1/4  -1/8|  600  1000]


sage: block_matrix([A, -A, ~A, 100*A], nrows=1)
[    3     9|   -3    -9|-5/12   3/8|  300   900]
[    6    10|   -6   -10|  1/4  -1/8|  600  1000]
```

It handle baserings nicely too:

```
sage: R.<x> = ZZ['x']
sage: block_matrix([1/2, A, 0, x-1])
[  1/2     0|    3     9]
[    0   1/2|    6    10]
[-----------+-----------]
[    0     0|x - 1     0]
[    0     0|    0 x - 1]
sage: block_matrix([1/2, A, 0, x-1]).parent()
Full MatrixSpace of 4 by 4 dense matrices over Univariate Polynomial Ring in x over Rational Fie
```

Subdivisions are optional:

```
sage: B = matrix(QQ, 2, 3, range(6))
sage: block_matrix([~A, B, B, ~A], subdivide=False)
[-5/12   3/8     0     1     2]
[  1/4  -1/8     3     4     5]
[    0     1     2 -5/12   3/8]
[    3     4     5   1/4  -1/8]
```

diagonal_matrix(*arg0=None, arg1=None, arg2=None, sparse=None*)

INPUT:

Supported formats

1. diagonal_matrix(diagonal_entries, [sparse=True]): diagonal matrix with flat list of entries
2. diagonal_matrix(nrows, diagonal_entries, [sparse=True]): diagonal matrix with flat list of entries and the rest zeros
3. diagonal_matrix(ring, diagonal_entries, [sparse=True]): diagonal matrix over specified ring with flat list of entries
4. diagonal_matrix(ring, nrows, diagonal_entries, [sparse=True]): diagonal matrix over specified ring with flat list of entries and the rest zeros
5. diagonal_matrix(vect, [sparse=True]): diagonal matrix with entries taken from a vector

The sparse option is optional, must be explicitly named (i.e., sparse=True), and may be either True or False.

EXAMPLES:

Input format 1:

```
sage: diagonal_matrix([1,2,3])
[1 0 0]
[0 2 0]
[0 0 3]
```

Input format 2:

```
sage: diagonal_matrix(3, [1,2])
[1 0 0]
[0 2 0]
[0 0 0]
```

Input format 3:

```
sage: diagonal_matrix(GF(3), [1,2,3])
[1 0 0]
[0 2 0]
[0 0 0]
```

Input format 4:

```
sage: diagonal_matrix(GF(3), 3, [8,2])
[2 0 0]
[0 2 0]
[0 0 0]
```

Input format 5:

```
sage: diagonal_matrix(vector(GF(3),[1,2,3]))
[1 0 0]
[0 2 0]
[0 0 0]
```

identity_matrix(*ring, n=0, sparse=False*)

Return the $n \times n$ identity matrix over the given ring.

The default ring is the integers.

EXAMPLES:

```
sage: M = identity_matrix(QQ, 2); M
[1 0]
[0 1]
sage: M.parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: M = identity_matrix(2); M
[1 0]
[0 1]
sage: M.parent()
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: M = identity_matrix(3, sparse=True); M
[1 0 0]
[0 1 0]
[0 0 1]
sage: M.parent()
Full MatrixSpace of 3 by 3 sparse matrices over Integer Ring
```

jordan_block(*eigenvalue, size, sparse=False*)

Form the Jordan block with the specified size associated with the eigenvalue.

INPUT:

- `eigenvalue` - eigenvalue for the diagonal entries of the block
- `size` - size of the Jordan block
- `sparse` - (default False) if True, return a sparse matrix

EXAMPLE:

```
sage: jordan_block(5, 3)
[5 1 0]
[0 5 1]
[0 0 5]
```

matrix(**args, **kwds*)

Create a matrix.

INPUT: The matrix command takes the entries of a matrix, optionally preceded by a ring and the dimensions of the matrix, and returns a matrix.

The entries of a matrix can be specified as a flat list of elements, a list of lists (i.e., a list of rows), a list of Sage vectors, or a dictionary having positions as keys and matrix entries as values (see the examples). You can create a matrix of zeros by passing an empty list or the integer zero for the entries. To construct a multiple of the identity (cI), you can specify square dimensions and pass in c. Calling matrix() with a Sage object may return something that makes sense. Calling matrix() with a numpy array will convert the array to a matrix.

The ring, number of rows, and number of columns of the matrix can be specified by setting the ring, nrows, or ncols parameters or by passing them as the first arguments to the function in the order ring, nrows, ncols. The ring defaults to ZZ if it is not specified or cannot be determined from the entries. If the numbers of rows and columns are not specified and cannot be determined, then an empty 0x0 matrix is returned.

- `ring` - the base ring for the entries of the matrix.
- `nrows` - the number of rows in the matrix.
- `ncols` - the number of columns in the matrix.
- `sparse` - create a sparse matrix. This defaults to True when the entries are given as a dictionary, otherwise defaults to False.

OUTPUT:

a matrix

EXAMPLES:

```
sage: m=matrix(2); m; m.parent()
[0 0]
[0 0]
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring

sage: m=matrix(2,3); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring

sage: m=matrix(QQ,[[1,2,3],[4,5,6]]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field

sage: v1=vector((1,2,3))
sage: v2=vector((4,5,6))
sage: m=matrix([v1,v2]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
```

```
sage: m=matrix(QQ,2,[1,2,3,4,5,6]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field

sage: m=matrix(QQ,2,3,[1,2,3,4,5,6]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field

sage: m=matrix({(0,1): 2, (1,1):2/5}); m; m.parent()
[  0   2]
[  0 2/5]
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field

sage: m=matrix(QQ,2,3,{(1,1): 2}); m; m.parent()
[0 0 0]
[0 2 0]
Full MatrixSpace of 2 by 3 sparse matrices over Rational Field

sage: import numpy
sage: n=numpy.array([[1,2],[3,4]],float)
sage: m=matrix(n); m; m.parent()
[1.0 2.0]
[3.0 4.0]
Full MatrixSpace of 2 by 2 dense matrices over Real Double Field

sage: v = vector(ZZ, [1, 10, 100])
sage: m=matrix(v); m; m.parent()
[  1  10 100]
Full MatrixSpace of 1 by 3 dense matrices over Integer Ring
sage: m=matrix(GF(7), v); m; m.parent()
[1 3 2]
Full MatrixSpace of 1 by 3 dense matrices over Finite Field of size 7

sage: g = graphs.PetersenGraph()
sage: m = matrix(g); m; m.parent()
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
Full MatrixSpace of 10 by 10 dense matrices over Integer Ring

sage: matrix(ZZ, 10, 10, range(100), sparse=True).parent()
Full MatrixSpace of 10 by 10 sparse matrices over Integer Ring
```

```
sage: R = PolynomialRing(QQ, 9, 'x')
sage: A = matrix(R, 3, 3, R.gens()); A
[x0 x1 x2]
[x3 x4 x5]
[x6 x7 x8]
sage: det(A)
-x2*x4*x6 + x1*x5*x6 + x2*x3*x7 - x0*x5*x7 - x1*x3*x8 + x0*x4*x8
```

TESTS:

```
sage: m=matrix(); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
sage: m=matrix(QQ); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Rational Field
sage: m=matrix(QQ,2); m; m.parent()
[0 0]
[0 0]
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: m=matrix(QQ,2,3); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
sage: m=matrix([]); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
sage: m=matrix(QQ,[]); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Rational Field
sage: m=matrix(2,2,1); m; m.parent()
[1 0]
[0 1]
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: m=matrix(QQ,2,2,1); m; m.parent()
[1 0]
[0 1]
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: m=matrix(2,3,0); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: m=matrix(QQ,2,3,0); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
sage: m=matrix([[1,2,3],[4,5,6]]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: m=matrix(QQ,2,[[1,2,3],[4,5,6]]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
sage: m=matrix(QQ,3,[[1,2,3],[4,5,6]]); m; m.parent()
...
ValueError: Number of rows does not match up with specified number.
sage: m=matrix(QQ,2,3,[[1,2,3],[4,5,6]]); m; m.parent()
```

```
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
sage: m=matrix(QQ,2,4,[[1,2,3],[4,5,6]]); m; m.parent()
...
ValueError: Number of columns does not match up with specified number.
sage: m=matrix([(1,2,3),(4,5,6)]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: m=matrix([1,2,3,4,5,6]); m; m.parent()
[1 2 3 4 5 6]
Full MatrixSpace of 1 by 6 dense matrices over Integer Ring
sage: m=matrix((1,2,3,4,5,6)); m; m.parent()
[1 2 3 4 5 6]
Full MatrixSpace of 1 by 6 dense matrices over Integer Ring
sage: m=matrix(QQ,[1,2,3,4,5,6]); m; m.parent()
[1 2 3 4 5 6]
Full MatrixSpace of 1 by 6 dense matrices over Rational Field
sage: m=matrix(QQ,3,2,[1,2,3,4,5,6]); m; m.parent()
[1 2]
[3 4]
[5 6]
Full MatrixSpace of 3 by 2 dense matrices over Rational Field
sage: m=matrix(QQ,2,4,[1,2,3,4,5,6]); m; m.parent()
...
ValueError: entries has the wrong length
sage: m=matrix(QQ,5,[1,2,3,4,5,6]); m; m.parent()
...
TypeError: entries has the wrong length
sage: m=matrix({(1,1): 2}); m; m.parent()
[0 0]
[0 2]
Full MatrixSpace of 2 by 2 sparse matrices over Integer Ring
sage: m=matrix(QQ,{(1,1): 2}); m; m.parent()
[0 0]
[0 2]
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
sage: m=matrix(QQ,3,{(1,1): 2}); m; m.parent()
[0 0 0]
[0 2 0]
[0 0 0]
Full MatrixSpace of 3 by 3 sparse matrices over Rational Field
sage: m=matrix(QQ,3,4,{(1,1): 2}); m; m.parent()
[0 0 0 0]
[0 2 0 0]
[0 0 0 0]
Full MatrixSpace of 3 by 4 sparse matrices over Rational Field
sage: m=matrix(QQ,2,{(1,1): 2}); m; m.parent()
[0 0]
[0 2]
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
sage: m=matrix(QQ,1,{(1,1): 2}); m; m.parent()
...
IndexError: invalid entries list
sage: m=matrix({}); m; m.parent()
[]
Full MatrixSpace of 0 by 0 sparse matrices over Integer Ring
```

```
sage: m=matrix(QQ,{}); m; m.parent()
[]
Full MatrixSpace of 0 by 0 sparse matrices over Rational Field
sage: m=matrix(QQ,2,{}); m; m.parent()
[0 0]
[0 0]
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
sage: m=matrix(QQ,2,3,{}); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 sparse matrices over Rational Field
sage: m=matrix(2,{}); m; m.parent()
[0 0]
[0 0]
Full MatrixSpace of 2 by 2 sparse matrices over Integer Ring
sage: m=matrix(2,3,{}); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 sparse matrices over Integer Ring
sage: m=matrix(0); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
sage: m=matrix(0,2); m; m.parent()
[]
Full MatrixSpace of 0 by 2 dense matrices over Integer Ring
sage: m=matrix(2,0); m; m.parent()
[]
Full MatrixSpace of 2 by 0 dense matrices over Integer Ring
sage: m=matrix(0,[1]); m; m.parent()
...
ValueError: entries has the wrong length
sage: m=matrix(1,0,[]); m; m.parent()
[]
Full MatrixSpace of 1 by 0 dense matrices over Integer Ring
sage: m=matrix(0,1,[]); m; m.parent()
[]
Full MatrixSpace of 0 by 1 dense matrices over Integer Ring
sage: m=matrix(0,[]); m; m.parent()
[]
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
sage: m=matrix(0,{}); m; m.parent()
[]
Full MatrixSpace of 0 by 0 sparse matrices over Integer Ring
sage: m=matrix(0,{(1,1):2}); m; m.parent()
...
IndexError: invalid entries list
sage: m=matrix(2,0,{(1,1):2}); m; m.parent()
...
IndexError: invalid entries list
sage: import numpy
sage: n=numpy.array([[numpy.complex(0,1),numpy.complex(0,2)],[3,4]],complex)
sage: m=matrix(n); m; m.parent()
[1.0*I 2.0*I]
[  3.0   4.0]
Full MatrixSpace of 2 by 2 dense matrices over Complex Double Field
sage: n=numpy.array([[1,2],[3,4]],'int32')
sage: m=matrix(n); m; m.parent()
[1 2]
```

```
[3 4]
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
sage: n = numpy.array([[1,2,3],[4,5,6],[7,8,9]],'float32')
sage: m=matrix(n); m; m.parent()
[1.0 2.0 3.0]
[4.0 5.0 6.0]
[7.0 8.0 9.0]
Full MatrixSpace of 3 by 3 dense matrices over Real Double Field
sage: n=numpy.array([[1,2,3],[4,5,6],[7,8,9]],'float64')
sage: m=matrix(n); m; m.parent()
[1.0 2.0 3.0]
[4.0 5.0 6.0]
[7.0 8.0 9.0]
Full MatrixSpace of 3 by 3 dense matrices over Real Double Field
sage: n=numpy.array([[1,2,3],[4,5,6],[7,8,9]],'complex64')
sage: m=matrix(n); m; m.parent()
[1.0 2.0 3.0]
[4.0 5.0 6.0]
[7.0 8.0 9.0]
Full MatrixSpace of 3 by 3 dense matrices over Complex Double Field
sage: n=numpy.array([[1,2,3],[4,5,6],[7,8,9]],'complex128')
sage: m=matrix(n); m; m.parent()
[1.0 2.0 3.0]
[4.0 5.0 6.0]
[7.0 8.0 9.0]
Full MatrixSpace of 3 by 3 dense matrices over Complex Double Field
sage: a = matrix([[1,2],[3,4]])
sage: b = matrix(a.numpy()); b
[1 2]
[3 4]
sage: a == b
True
sage: c = matrix(a.numpy('float32')); c
[1.0 2.0]
[3.0 4.0]
sage: v = vector(ZZ, [1, 10, 100])
sage: m=matrix(ZZ['x'], v); m; m.parent()
[  1  10 100]
Full MatrixSpace of 1 by 3 dense matrices over Univariate Polynomial Ring in x over Integer Ring
sage: matrix(ZZ, 10, 10, range(100)).parent()
Full MatrixSpace of 10 by 10 dense matrices over Integer Ring
sage: m = matrix(GF(7), [[1/3,2/3,1/2], [3/4,4/5,7]]); m; m.parent()
[5 3 4]
[6 5 0]
Full MatrixSpace of 2 by 3 dense matrices over Finite Field of size 7
sage: m = matrix([[1,2,3], [RDF(2), CDF(1,2), 3]]); m; m.parent()
[        1.0         2.0         3.0]
[        2.0 1.0 + 2.0*I         3.0]
Full MatrixSpace of 2 by 3 dense matrices over Complex Double Field
sage: m=matrix(3,3,1/2); m; m.parent()
[1/2   0   0]
[  0 1/2   0]
[  0   0 1/2]
Full MatrixSpace of 3 by 3 dense matrices over Rational Field
sage: matrix([[1],[2,3]])
...
ValueError: List of rows is not valid (rows are wrong types or lengths)
sage: matrix([[1],2])
```

```
...
ValueError: List of rows is not valid (rows are wrong types or lengths)
sage: matrix(vector(RR,[1,2,3])).parent()
Full MatrixSpace of 1 by 3 dense matrices over Real Field with 53 bits of precision
```

AUTHORS:

- ??: Initial implementation
- Jason Grout (2008-03): almost a complete rewrite, with bits and pieces from the original implementation

ncols_from_dict(*d*)

Given a dictionary that defines a sparse matrix, return the number of columns that matrix should have.

This is for internal use by the matrix function.

INPUT:

- d - dict

OUTPUT:

integer

EXAMPLES:

```
sage: sage.matrix.constructor.ncols_from_dict({})
0
```

Here the answer is 301 not 300, since there is a 0-th row.

```
sage: sage.matrix.constructor.ncols_from_dict({(4,300):10})
301
```

nrows_from_dict(*d*)

Given a dictionary that defines a sparse matrix, return the number of rows that matrix should have.

This is for internal use by the matrix function.

INPUT:

- d - dict

OUTPUT:

integer

EXAMPLES:

```
sage: sage.matrix.constructor.nrows_from_dict({})
0
```

Here the answer is 301 not 300, since there is a 0-th row.

:: sage: sage.matrix.constructor.nrows_from_dict({(300,4):10}) 301

prepare(*w*)

Given a list w of numbers, find a common ring that they all canonically map to, and return the list of images of the elements of w in that ring along with the ring.

This is for internal use by the matrix function.

INPUT:

- w - list

OUTPUT:

list, ring

EXAMPLES:

```
sage: sage.matrix.constructor.prepare([-2, Mod(1,7)])
([5, 1], Ring of integers modulo 7)
```

Notice that the elements must all canonically coerce to a common ring (since Sequence is called):

```
sage: sage.matrix.constructor.prepare([2/1, Mod(1,7)])
...
TypeError: unable to find a common ring for all elements
```

prepare_dict(*w*)

Given a dictionary w of numbers, find a common ring that they all canonically map to, and return the dictionary of images of the elements of w in that ring along with the ring.

This is for internal use by the matrix function.

INPUT:

- `w` - dict

OUTPUT:

dict, ring

EXAMPLES:

```
sage: sage.matrix.constructor.prepare_dict({(0,1):2, (4,10):Mod(1,7)})
({(0, 1): 2, (4, 10): 1}, Ring of integers modulo 7)
```

random_matrix(*R, nrows, ncols=None, sparse=False, density=1, *args, **kwds*)

Return a random matrix with entries in the ring R.

INPUT:

- `R` - a ring
- `nrows` - integer; number of rows
- `ncols` - (default: None); number of columns; if None defaults to nrows
- `sparse` - (default; False); whether or not matrix is sparse.
- `density` - integer (default: 1)
- `*args, **kwds` - passed on to randomize function

EXAMPLES:

```
sage: A = random_matrix(ZZ,50,x=2^16)     # entries are up to 2^16 i size
sage: A
50 x 50 dense matrix over Integer Ring
```

zero_matrix(*ring, nrows, ncols=None, sparse=False*)

Return the $nrows \times ncols$ zero matrix over the given ring.

The default ring is the integers.

EXAMPLES:

```
sage: M = zero_matrix(QQ, 2); M
[0 0]
[0 0]
sage: M.parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: M = zero_matrix(2, 3); M
[0 0 0]
[0 0 0]
sage: M.parent()
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: M = zero_matrix(3, 1, sparse=True); M
[0]
[0]
[0]
sage: M.parent()
Full MatrixSpace of 3 by 1 sparse matrices over Integer Ring
```

32.3 Matrices over an arbitrary ring

AUTHORS:

- William Stein
- Martin Albrecht: conversion to Pyrex
- Jaap Spies: various functions
- Gary Zablackis: fixed a sign bug in generic determinant.
- William Stein and Robert Bradshaw - complete restructuring.
- Rob Beezer - refactor kernel functions.

Elements of matrix spaces are of class `Matrix` (or a class derived from Matrix). They can be either sparse or dense, and can be defined over any base ring.

EXAMPLES:

We create the 2×3 matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

as an element of a matrix space over $\mathbf{Q}$:

```
sage: M = MatrixSpace(QQ,2,3)
sage: A = M([1,2,3, 4,5,6]); A
[1 2 3]
[4 5 6]
sage: A.parent()
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
```

Alternatively, we could create A more directly as follows (which would completely avoid having to create the matrix space):

```
sage: A = matrix(QQ, 2, [1,2,3, 4,5,6]); A
[1 2 3]
[4 5 6]
```

We next change the top-right entry of A. Note that matrix indexing is 0-based in Sage, so the top right entry is $(0, 2)$, which should be thought of as "row number 0, column number 2".

```
sage: A[0,2] = 389
sage: A
[  1   2 389]
[  4   5   6]
```

Also notice how matrices print. All columns have the same width and entries in a given column are right justified. Next we compute the reduced row echelon form of A.

```
sage: A.echelon_form()
[      1       0 -1933/3]
[      0       1  1550/3]
```

We save and load a matrix:

```
sage: A = matrix(Integers(8),3,range(9))
sage: loads(dumps(A)) == A
True
```

MUTABILITY: Matrices are either immutable or not. When initially created, matrices are typically mutable, so one can change their entries. Once a matrix A is made immutable using `A.set_immutable()` the entries of A cannot be changed, and A can never be made mutable again. However, properies of A such as its rank, characteristic polynomial, etc., are all cached so computations involving A may be more efficient. Once A is made immutable it cannot be changed back. However, one can obtain a mutable copy of A using `A.copy()`.

EXAMPLES:

```
sage: A = matrix(RR,2,[1,10,3.5,2])
sage: A.set_immutable()
sage: A.copy() is A
False
```

The echelon form method always returns immutable matrices with known rank.

EXAMPLES:

```
sage: A = matrix(Integers(8),3,range(9))
sage: A.determinant()
0
sage: A[0,0] = 5
sage: A.determinant()
1
sage: A.set_immutable()
sage: A[0,0] = 5
...
ValueError: matrix is immutable; please change a copy instead (use self.copy()).
```

32.3.1 Implementation and Design

Class Diagram (an x means that class is currently supported):

```
x Matrix
x   Matrix_sparse
x     Matrix_generic_sparse
x     Matrix_integer_sparse
x     Matrix_rational_sparse
      Matrix_cyclo_sparse
x     Matrix_modn_sparse
      Matrix_RR_sparse
      Matrix_CC_sparse
      Matrix_RDF_sparse
      Matrix_CDF_sparse

x  Matrix_dense
x     Matrix_generic_dense
x     Matrix_integer_dense
      Matrix_integer_2x2_dense
x     Matrix_rational_dense
      Matrix_cyclo_dense    -- idea: restrict scalars to QQ, compute charpoly there, then factor
x     Matrix_modn_dense
      Matrix_RR_dense
      Matrix_CC_dense
x     Matrix_real_double_dense
x     Matrix_complex_double_dense
```

The corresponding files in the sage/matrix library code directory are named

```
[matrix] [base ring] [dense or sparse].
```

See the files `matrix_template.pxd` and `matrix_template.pyx`.

```
New matrices types can only be implemented in Cython.

********** LEVEL 1  **********
NON-OPTIONAL
For each base field it is *absolutely* essential to completely
implement the following functionality for that base ring:

   * __new__       -- should use sage_malloc from ext/stdsage.pxi (only
                      needed if allocate memory)
   * __init__      -- this signature: 'def __init__(self, parent, entries, copy, coerce)'
   * __dealloc__   -- use sage_free (only needed if allocate memory)
   * set_unsafe(self, size_t i, size_t j, x) -- doesn't do bounds or any other checks; assumes x is
   * get_unsafe(self, size_t i, size_t j) -- doesn't do checks
   * __richcmp__    -- always the same (I don't know why its needed -- bug in PYREX).
********** LEVEL 2  **********

IMPORTANT (and *highly* recommended):

After getting the special class with all level 1 functionality to
work, implement all of the following (they should not change
functionality, except speed (always faster!) in any way):

   * def _pickle(self):
```

```
        return data, version
   * def _unpickle(self, data, int version)
        reconstruct matrix from given data and version; may assume _parent, _nrows, and _ncols are
        Use version numbers >= 0 so if you change the pickle strategy then
        old objects still unpickle.
   * cdef _list -- list of underlying elements (need not be a copy)
   * cdef _dict -- sparse dictionary of underlying elements
   * cdef _add_ -- add two matrices with identical parents
   * _matrix_times_matrix_c_impl -- multiply two matrices with compatible dimensions and
                                    identical base rings (both sparse or both dense)
   * cdef _cmp_c_impl -- compare two matrices with identical parents
   * cdef _lmul_c_impl -- multiply this matrix on the right by a scalar, i.e., self * scalar
   * cdef _rmul_c_impl -- multiply this matrix on the left by a scalar, i.e., scalar * self
   * __copy__
   * __neg__

The list and dict returned by _list and _dict will *not* be changed
by any internal algorithms and are not accessible to the user.

********** LEVEL 3  **********
OPTIONAL:

   * cdef _sub_
   * __invert__
   * _multiply_classical
   * __deepcopy__

Further special support:
   * Matrix windows -- to support Strassen multiplication for a given base ring.
   * Other functions, e.g., transpose, for which knowing the
     specific representation can be helpful.

.. note::

   - For caching, use self.fetch and self.cache.

   - Any method that can change the matrix should call
     ``check_mutability()`` first.  There are also many fast cdef'd bounds checking methods.

   - Kernels of matrices
     Implement only a left_kernel() or right_kernel() method, whichever requires
     the least overhead (usually meaning little or no transpose'ing).  Let the
     methods in the matrix2 class handle left, right, generic kernel distinctions.
```

32.4 Abstract base class for matrices.

For design documentation see matrix/docs.py.

class Matrix()

is_Matrix()

EXAMPLES:

```
sage: from sage.matrix.matrix import is_Matrix
sage: is_Matrix(0)
False
```

```
sage: is_Matrix(matrix([[1,2],[3,4]]))
True
```

32.5 Base class for matrices, part 0

Note: For design documentation see matrix/docs.py.

EXAMPLES:

```
sage: matrix(2,[1,2,3,4])
[1 2]
[3 4]
```

class Matrix()

A generic matrix.

The `Matrix` class is the base class for all matrix classes. To create a `Matrix`, first create a `MatrixSpace`, then coerce a list of elements into the `MatrixSpace`. See the documentation of `MatrixSpace` for more details.

EXAMPLES:

We illustrate matrices and matrix spaces. Note that no actual matrix that you make should have class Matrix; the class should always be derived from Matrix.

```
sage: M = MatrixSpace(CDF,2,3); M
Full MatrixSpace of 2 by 3 dense matrices over Complex Double Field
sage: a = M([1,2,3,  4,5,6]); a
[1.0 2.0 3.0]
[4.0 5.0 6.0]
sage: type(a)
<type 'sage.matrix.matrix_complex_double_dense.Matrix_complex_double_dense'>
sage: parent(a)
Full MatrixSpace of 2 by 3 dense matrices over Complex Double Field
```

```
sage: matrix(CDF, 2,3, [1,2,3, 4,5,6])
[1.0 2.0 3.0]
[4.0 5.0 6.0]
sage: Mat(CDF,2,3)(range(1,7))
[1.0 2.0 3.0]
[4.0 5.0 6.0]
```

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -1,-1)
sage: matrix(Q,2,1,[1,2])
[1]
[2]
```

act_on_polynomial()

Returns the polynomial f(self*x).

INPUT:

- `self` - an nxn matrix
- `f` - a polynomial in n variables x=(x1,...,xn)

OUTPUT: The polynomial f(self*x).

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: x, y = R.gens()
sage: f = x**2 - y**2
sage: M = MatrixSpace(QQ, 2)
sage: A = M([1,2,3,4])
sage: A.act_on_polynomial(f)
-8*x^2 - 20*x*y - 12*y^2
```

add_multiple_of_column()

Add s times column j to column i.

EXAMPLES: We add -1 times the third column to the second column of an integer matrix, remembering to start numbering cols at zero:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.add_multiple_of_column(1,2,-1)
sage: a
[ 0 -1  2]
[ 3 -1  5]
```

To add a rational multiple, we first need to change the base ring:

```
sage: a = a.change_ring(QQ)
sage: a.add_multiple_of_column(1,0,1/3)
sage: a
[ 0 -1  2]
[ 3  0  5]
```

If not, we get an error message:

```
sage: a.add_multiple_of_column(1,0,i)
...
TypeError: Multiplying column by Symbolic Ring element cannot be done over Rational Field, u
```

add_multiple_of_row()

Add s times row j to row i.

EXAMPLES: We add -3 times the first row to the second row of an integer matrix, remembering to start numbering rows at zero:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.add_multiple_of_row(1,0,-3)
sage: a
[ 0  1  2]
[ 3  1 -1]
```

To add a rational multiple, we first need to change the base ring:

```
sage: a = a.change_ring(QQ)
sage: a.add_multiple_of_row(1,0,1/3)
sage: a
[   0    1    2]
[   3  4/3 -1/3]
```

If not, we get an error message:

```
sage: a.add_multiple_of_row(1,0,i)
...
TypeError: Multiplying row by Symbolic Ring element cannot be done over Rational Field, use
```

base_ring()

change_ring()

Return the matrix obtained by coercing the entries of this matrix into the given ring.

Always returns a copy (unless self is immutable, in which case returns self).

EXAMPLES:

```
sage: A = Matrix(QQ, 2, 2, [1/2, 1/3, 1/3, 1/4])
sage: A.parent()
 Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: A.change_ring(GF(25,'a'))
[3 2]
[2 4]
sage: A.change_ring(GF(25,'a')).parent()
 Full MatrixSpace of 2 by 2 dense matrices over Finite Field in a of size 5^2
sage: A.change_ring(ZZ)
...
TypeError: matrix has denominators so can't change to ZZ.
```

Changing rings preserves subdivisions:

```
sage: A.subdivide([1], []); A
[1/2 1/3]
[-------]
[1/3 1/4]
sage: A.change_ring(GF(25,'a'))
[3 2]
[---]
[2 4]
```

commutator()

Return the commutator self*other - other*self.

EXAMPLES:

```
sage: A = Matrix(ZZ, 2, 2, range(4))
sage: B = Matrix(ZZ, 2, 2, [0, 1, 0, 0])
sage: A.commutator(B)
[-2 -3]
[ 0  2]
sage: A.commutator(B) == -B.commutator(A)
True
```

copy()

Make a copy of self. If self is immutable, the copy will be mutable.

> **Warning:** The individual elements aren't themselves copied (though the list is copied). This shouldn't matter, since ring elements are (almost!) always immutable in Sage.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: a = matrix(R,2,[x+1,2/3,  x^2/2, 1+x^3]); a
[  x + 1     2/3]
[1/2*x^2 x^3 + 1]
sage: b = a.copy()
sage: b[0,0] = 5
sage: b
[      5     2/3]
[1/2*x^2 x^3 + 1]
sage: a
```

```
[  x + 1     2/3]
[1/2*x^2 x^3 + 1]

sage: b = copy(a)
sage: f = b[0,0]; f[0] = 10
...
IndexError: polynomials are immutable
```

dict()

Dictionary of the elements of self with keys pairs (i,j) and values the nonzero entries of self.

It is safe to change the returned dictionary.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: a = matrix(R,2,[x,y,0, 0,0,2*x+y]); a
[      x       y       0]
[      0       0 2*x + y]
sage: d = a.dict(); d
{(0, 1): y, (1, 2): 2*x + y, (0, 0): x}
```

Notice that changing the returned list does not change a (the list is a copy):

```
sage: d[0,0] = 25
sage: a
[      x       y       0]
[      0       0 2*x + y]
```

is_dense()

Returns True if this is a dense matrix.

In Sage, being dense is a property of the underlying representation, not the number of nonzero entries.

EXAMPLES:

```
sage: matrix(QQ,2,2,range(4)).is_dense()
True
sage: matrix(QQ,2,2,range(4),sparse=True).is_dense()
False
```

is_immutable()

Return True if this matrix is immutable.

See the documentation for self.set_immutable for more details about mutability.

EXAMPLES:

```
sage: A = Matrix(QQ['t','s'], 2, 2, range(4))
sage: A.is_immutable()
False
sage: A.set_immutable()
sage: A.is_immutable()
True
```

is_invertible()

Return True if this matrix is invertible.

EXAMPLES: The following matrix is invertible over $\mathbf{Q}$ but not over $\mathbf{Z}$.

```
sage: A = MatrixSpace(ZZ, 2)(range(4))
sage: A.is_invertible()
False
sage: A.matrix_over_field().is_invertible()
True
```

The inverse function is a constructor for matrices over the fraction field, so it can work even if A is not invertible.

```
sage: ~A   # inverse of A
[-3/2  1/2]
[   1    0]
```

The next matrix is invertible over $\mathbf{Z}$.

```
sage: A = MatrixSpace(IntegerRing(),2)([1,10,0,-1])
sage: A.is_invertible()
True
sage: ~A                # compute the inverse
[ 1 10]
[ 0 -1]
```

The following nontrivial matrix is invertible over $\mathbf{Z}[x]$.

```
sage: R.<x> = PolynomialRing(IntegerRing())
sage: A = MatrixSpace(R,2)([1,x,0,-1])
sage: A.is_invertible()
True
sage: ~A
[ 1  x]
[ 0 -1]
```

is_mutable()

Return True if this matrix is mutable.

See the documentation for self.set_immutable for more details about mutability.

EXAMPLES:

```
sage: A = Matrix(QQ['t','s'], 2, 2, range(4))
sage: A.is_mutable()
True
sage: A.set_immutable()
sage: A.is_mutable()
False
```

is_sparse()

Return True if this is a sparse matrix.

In Sage, being sparse is a property of the underlying representation, not the number of nonzero entries.

EXAMPLES:

```
sage: matrix(QQ,2,2,range(4)).is_sparse()
False
sage: matrix(QQ,2,2,range(4),sparse=True).is_sparse()
True
```

is_square()

Return True precisely if this matrix is square, i.e., has the same number of rows and columns.

EXAMPLES:

```
sage: matrix(QQ,2,2,range(4)).is_square()
True
sage: matrix(QQ,2,3,range(6)).is_square()
False
```

is_symmetric()

Returns True if this is a symmetric matrix.

iterates()

Let A be this matrix and v be a free module element. If rows is True, return a matrix whose rows are the

entries of the following vectors:

$$v, vA, vA^2, \ldots, vA^{n-1}.$$

If rows is False, return a matrix whose columns are the entries of the following vectors:

$$v, Av, A^2v, \ldots, A^{n-1}v.$$

INPUT:

- v - free module element
- n - nonnegative integer

EXAMPLES:

```
sage: A = matrix(ZZ,2, [1,1,3,5]); A
[1 1]
[3 5]
sage: v = vector([1,0])
sage: A.iterates(v,0)
[]
sage: A.iterates(v,5)
[  1   0]
[  1   1]
[  4   6]
[ 22  34]
[124 192]
```

Another example:

```
sage: a = matrix(ZZ,3,range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: v = vector([1,0,0])
sage: a.iterates(v,4)
[  1   0   0]
[  0   1   2]
[ 15  18  21]
[180 234 288]
sage: a.iterates(v,4,rows=False)
[  1   0  15 180]
[  0   3  42 558]
[  0   6  69 936]
```

linear_combination_of_columns()

Return the linear combination of the columns of self given by the coefficients in the list v. Raise a ValueError if the length of v is longer than the number of columns of the matrix.

INPUT:

- v - list of length at most the number of columns of self (less is fine)

EXAMPLES:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.linear_combination_of_columns([1,1,1])
(3, 12)
sage: a.linear_combination_of_columns([0,0,0])
(0, 0)
sage: a.linear_combination_of_columns([1/2,2/3,3/4])
```

```
 (13/6, 95/12)
sage: matrix(QQ,2,0).linear_combination_of_columns([])
(0, 0)
```

The length of v can be less than the number of columns, but not more than the number of columns:

```
sage: A = matrix(QQ,2,3)
sage: A.linear_combination_of_columns([0])
(0, 0)
sage: A.linear_combination_of_columns([1,2,3,4])
...
ValueError: length of v must be at most the number of columns of self
```

linear_combination_of_rows()

Return the linear combination of the rows of self given by the coefficients in the list v. Raise a ValueError if the length of v is longer than the number of rows of the matrix.

INPUT:

- **v - list of length at most the number of rows of** self (less is fine)

EXAMPLES:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.linear_combination_of_rows([1,2])
(6, 9, 12)
sage: a.linear_combination_of_rows([0,0])
(0, 0, 0)
sage: a.linear_combination_of_rows([1/2,2/3])
(2, 19/6, 13/3)
sage: matrix(QQ,0,2).linear_combination_of_rows([])
(0, 0)
```

The length of v can be less than the number of rows, but not more than the number of rows:

```
sage: A = matrix(QQ,2,3)
sage: A.linear_combination_of_rows([0])
(0, 0, 0)
sage: A.linear_combination_of_rows([1,2,3])
...
ValueError: length of v must be at most the number of rows of self
```

list()

List of elements of self. It is safe to change the returned list.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: a = matrix(R,2,[x,y,x*y, y,x,2*x+y]); a
[      x       y     x*y]
[      y       x 2*x + y]
sage: v = a.list(); v
[x, y, x*y, y, x, 2*x + y]
```

Notice that changing the returned list does not change a (the list is a copy):

```
sage: v[0] = 25
sage: a
[      x       y     x*y]
[      y       x 2*x + y]
```

mod()

Return matrix mod p, over the reduced ring.

EXAMPLES:

```
sage: M = matrix(ZZ, 2, 2, [5, 9, 13, 15])
sage: M.mod(7)
[5 2]
[6 1]
sage: parent(M.mod(7))
Full MatrixSpace of 2 by 2 dense matrices over Ring of integers modulo 7
```

ncols()

Return the number of columns of this matrix.

EXAMPLES:

```
sage: M = MatrixSpace(QQ, 2, 3)
sage: A = M([1,2,3, 4,5,6])
sage: A
[1 2 3]
[4 5 6]
sage: A.ncols()
3
sage: A.nrows()
2
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples

nonpivots()

Return the list of i such that the i-th column of self is NOT a pivot column of the reduced row echelon form of self.

OUTPUT:

- `list` - sorted list of (Python) integers

EXAMPLES:

```
sage: a = matrix(QQ,3,3,range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: a.echelon_form()
[ 1  0 -1]
[ 0  1  2]
[ 0  0  0]
sage: a.nonpivots()
[2]
```

nonzero_positions()

Returns the sorted list of pairs (i,j) such that self[i,j] != 0.

INPUT:

- `copy` - (default: True) It is safe to change the resulting list (unless you give the option copy=False).
- `column_order` - (default: False) If true, returns the list of pairs (i,j) such that self[i,j] != 0, but sorted by columns, i.e., column j=0 entries occur first, then column j=1 entries, etc.

EXAMPLES:

```
sage: a = matrix(QQ, 2,3, [1,2,0,2,0,0]); a
[1 2 0]
[2 0 0]
```

```
sage: a.nonzero_positions()
[(0, 0), (0, 1), (1, 0)]
sage: a.nonzero_positions(copy=False)
[(0, 0), (0, 1), (1, 0)]
sage: a.nonzero_positions(column_order=True)
[(0, 0), (1, 0), (0, 1)]
sage: a = matrix(QQ, 2,3, [1,2,0,2,0,0], sparse=True); a
[1 2 0]
[2 0 0]
sage: a.nonzero_positions()
[(0, 0), (0, 1), (1, 0)]
sage: a.nonzero_positions(copy=False)
[(0, 0), (0, 1), (1, 0)]
sage: a.nonzero_positions(column_order=True)
[(0, 0), (1, 0), (0, 1)]
```

nonzero_positions_in_column()

Return a sorted list of the integers j such that self[j,i] is nonzero, i.e., such that the j-th position of the i-th column is nonzero.

INPUT:

- i - an integer

OUTPUT: list

EXAMPLES:

```
sage: a = matrix(QQ, 3,2, [1,2,0,2,0,0]); a
[1 2]
[0 2]
[0 0]
sage: a.nonzero_positions_in_column(0)
[0]
sage: a.nonzero_positions_in_column(1)
[0, 1]
```

You'll get an IndexError, if you select an invalid column:

```
sage: a.nonzero_positions_in_column(2)
...
IndexError: matrix column index out of range
```

nonzero_positions_in_row()

Return the integers j such that self[i,j] is nonzero, i.e., such that the j-th position of the i-th row is nonzero.

INPUT:

- i - an integer

OUTPUT: list

EXAMPLES:

```
sage: a = matrix(QQ, 3,2, [1,2,0,2,0,0]); a
[1 2]
[0 2]
[0 0]
sage: a.nonzero_positions_in_row(0)
[0, 1]
sage: a.nonzero_positions_in_row(1)
[1]
sage: a.nonzero_positions_in_row(2)
[]
```

nrows()

Return the number of rows of this matrix.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,6,7)
sage: A = M([1,2,3,4,5,6,7, 22,3/4,34,11,7,5,3, 99,65,1/2,2/3,3/5,4/5,5/6, 9,8/9, 9/8,7/6,6/
sage: A
[   1    2    3    4    5    6    7]
[  22  3/4   34   11    7    5    3]
[  99   65  1/2  2/3  3/5  4/5  5/6]
[   9  8/9  9/8  7/6  6/7   76    4]
[   0    9    8    7    6    5    4]
[ 123   99   91   28    6 1024    1]
sage: A.ncols()
7
sage: A.nrows()
6
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples

pivots()

Return the pivot column positions of this matrix as a list of Python integers.

This returns a list, of the position of the first nonzero entry in each row of the echelon form.

OUTPUT:

- `list` - a list of Python ints

EXAMPLES:

rank()

rescale_col()

Replace i-th col of self by s times i-th col of self.

INPUT:

- `i` - ith column
- `s` - scalar
- `start_row` - only rescale entries at this row and lower

EXAMPLES: We rescale the last column of a matrix over the rational numbers:

```
sage: a = matrix(QQ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.rescale_col(2,1/2); a
[  0   1   1]
[  3   4 5/2]
sage: R.<x> = QQ[]
```

We rescale the last column of a matrix over a polynomial ring:

```
sage: a = matrix(R,2,3,[1,x,x^2,x^3,x^4,x^5]); a
[  1   x x^2]
[x^3 x^4 x^5]
sage: a.rescale_col(2,1/2); a
[      1       x 1/2*x^2]
[    x^3     x^4 1/2*x^5]
```

We try and fail to rescale a matrix over the integers by a non-integer:

```
sage: a = matrix(ZZ,2,3,[0,1,2, 3,4,4]); a
[0 1 2]
[3 4 4]
sage: a.rescale_col(2,1/2)
...
TypeError: Rescaling column by Rational Field element cannot be done over Integer Ring, use
```

To rescale the matrix by 1/2, you must change the base ring to the rationals:

```
sage: a = a.change_ring(QQ); a
[0 1 2]
[3 4 4]
sage: a.rescale_col(2,1/2); a
[0 1 1]
[3 4 2]
```

rescale_row()

Replace i-th row of self by s times i-th row of self.

INPUT:

- `i` - ith row
- `s` - scalar
- `start_col` - only rescale entries at this column and to the right

EXAMPLES: We rescale the second row of a matrix over the rational numbers:

```
sage: a = matrix(QQ,3,range(6)); a
[0 1]
[2 3]
[4 5]
sage: a.rescale_row(1,1/2); a
[ 0   1]
[ 1 3/2]
[ 4   5]
```

We rescale the second row of a matrix over a polynomial ring:

```
sage: R.<x> = QQ[]
sage: a = matrix(R,3,[1,x,x^2,x^3,x^4,x^5]);a
[  1   x]
[x^2 x^3]
[x^4 x^5]
sage: a.rescale_row(1,1/2); a
[      1       x]
[1/2*x^2 1/2*x^3]
[    x^4     x^5]
```

We try and fail to rescale a matrix over the integers by a non-integer:

```
sage: a = matrix(ZZ,2,3,[0,1,2, 3,4,4]); a
[0 1 2]
[3 4 4]
sage: a.rescale_row(1,1/2)
...
TypeError: Rescaling row by Rational Field element cannot be done over Integer Ring, use cha
```

To rescale the matrix by 1/2, you must change the base ring to the rationals:

```
sage: a = a.change_ring(QQ); a
[0 1 2]
[3 4 4]
sage: a.rescale_col(1,1/2); a
```

```
[  0 1/2   2]
[  3   2   4]
```

set_col_to_multiple_of_col()

Set column i equal to s times column j.

EXAMPLES: We change the second column to -3 times the first column.

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.set_col_to_multiple_of_col(1,0,-3)
sage: a
[ 0  0  2]
[ 3 -9  5]
```

If we try to multiply a column by a rational number, we get an error message:

```
sage: a.set_col_to_multiple_of_col(1,0,1/2)
...
TypeError: Multiplying column by Rational Field element cannot be done over Integer Ring, us
```

set_immutable()

Call this function to matrix a matrix immutable.

Matrices are always mutable by default, i.e., you can change their entries using `A[i,j] = x`. However, mutable matrices aren't hasheable, so can't be used as keys in dictionaries, etc. Also, often when implementing a class, you might compute a matrix associated to it, e.g., the matrix of a Hecke operator. If you return this matrix to the user you're really returning a reference and the user could then change an entry; this could be confusing. Thus you shoulds set such a matrix immutable.

EXAMPLES:

```
sage: A = Matrix(QQ, 2, 2, range(4))
sage: A.is_mutable()
True
sage: A[0,0] = 10
sage: A
[10   1]
[ 2   3]
```

Mutable matrices are not hasheable, so can't be used as keys for dictionaries:

```
sage: hash(A)
...
TypeError: mutable matrices are unhashable
sage: v = {A:1}
...
TypeError: mutable matrices are unhashable
```

If we make A immutable it suddenly is hasheable.

```
sage: A.set_immutable()
sage: A.is_mutable()
False
sage: A[0,0] = 10
...
ValueError: matrix is immutable; please change a copy instead (use self.copy()).
sage: hash(A)
12
sage: v = {A:1}; v
{[10  1]
 [ 2  3]: 1}
```

set_row_to_multiple_of_row()
Set row i equal to s times row j.

EXAMPLES: We change the second row to -3 times the first row:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.set_row_to_multiple_of_row(1,0,-3)
sage: a
[ 0  1  2]
[ 0 -3 -6]
```

If we try to multiply a row by a rational number, we get an error message:

```
sage: a.set_row_to_multiple_of_row(1,0,1/2)
...
TypeError: Multiplying row by Rational Field element cannot be done over Integer Ring, use c
```

str()
EXAMPLES:

```
sage: R = PolynomialRing(QQ,6,'z')
sage: a = matrix(2,3, R.gens())
sage: a.__repr__()
'[z0 z1 z2]\n[z3 z4 z5]'
```

subdivisions

swap_columns()
Swap columns c1 and c2 of self.

EXAMPLES: We create a rational matrix:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,9,-7,4/5,4,3,6,4,3])
sage: A
[  1   9  -7]
[4/5   4   3]
[  6   4   3]
```

Since the first column is numbered zero, this swaps the second and third columns:

```
sage: A.swap_columns(1,2); A
[  1  -7   9]
[4/5   3   4]
[  6   3   4]
```

swap_rows()
Swap rows r1 and r2 of self.

EXAMPLES: We create a rational matrix:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,9,-7,4/5,4,3,6,4,3])
sage: A
[  1   9  -7]
[4/5   4   3]
[  6   4   3]
```

Since the first row is numbered zero, this swaps the first and third rows:

```
sage: A.swap_rows(0,2); A
[  6   4   3]
```

```
[4/5   4   3]
[  1   9  -7]
```

with_added_multiple_of_column()

Add s times column j to column i, returning new matrix.

EXAMPLES: We add -1 times the third column to the second column of an integer matrix, remembering to start numbering cols at zero:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: b = a.with_added_multiple_of_column(1,2,-1); b
[ 0 -1  2]
[ 3 -1  5]
```

The original matrix is unchanged:

```
sage: a
[0 1 2]
[3 4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_added_multiple_of_column(0,1,1/3); a
[ 1/3    1    2]
[13/3    4    5]
```

with_added_multiple_of_row()

Add s times row j to row i, returning new matrix.

EXAMPLES: We add -3 times the first row to the second row of an integer matrix, remembering to start numbering rows at zero:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: b = a.with_added_multiple_of_row(1,0,-3); b
[ 0  1  2]
[ 3  1 -1]
```

The original matrix is unchanged:

```
sage: a
[0 1 2]
[3 4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_added_multiple_of_row(0,1,1/3); a
[   1  7/3 11/3]
[   3    4    5]
```

with_col_set_to_multiple_of_col()

Set column i equal to s times column j, returning a new matrix.

EXAMPLES: We change the second column to -3 times the first column.

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: b = a.with_col_set_to_multiple_of_col(1,0,-3); b
[ 0  0  2]
[ 3 -9  5]
```

Note that the original matrix is unchanged:

```
sage: a
[0 1 2]
[3 4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_col_set_to_multiple_of_col(1,0,1/2); a
[  0   0   2]
[  3 3/2   5]
```

with_rescaled_col()

Replaces i-th col of self by s times i-th col of self, returning new matrix.

EXAMPLES: We rescale the last column of a matrix over the integers:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: b = a.with_rescaled_col(2,-2); b
[  0   1  -4]
[  3   4 -10]
```

The original matrix is unchanged:

```
sage: a
[0 1 2]
[3 4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_rescaled_col(1,1/3); a
[  0 1/3   2]
[  3 4/3   5]
```

with_rescaled_row()

Replaces i-th row of self by s times i-th row of self, returning new matrix.

EXAMPLES: We rescale the second row of a matrix over the integers:

```
sage: a = matrix(ZZ,3,2,range(6)); a
[0 1]
[2 3]
[4 5]
sage: b = a.with_rescaled_row(1,-2); b
[ 0  1]
[-4 -6]
[ 4  5]
```

The original matrix is unchanged:

```
sage: a
[0 1]
[2 3]
[4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_rescaled_row(2,1/3); a
[  0   1]
[  2   3]
[4/3 5/3]
```

with_row_set_to_multiple_of_row()

Set row i equal to s times row j, returning a new matrix.

EXAMPLES: We change the second row to -3 times the first row:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: b = a.with_row_set_to_multiple_of_row(1,0,-3); b
[ 0  1  2]
[ 0 -3 -6]
```

Note that the original matrix is unchanged:

```
sage: a
[0 1 2]
[3 4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_row_set_to_multiple_of_row(1,0,1/2); a
[  0   1   2]
[  0 1/2   1]
```

set_max_cols()

set_max_rows()

unpickle()

Unpickle a matrix. This is only used internally by Sage. Users should never call this function directly.

EXAMPLES: We illustrating saving and loading several different types of matrices.

OVER $\mathbf{Z}$:

```
sage: A = matrix(ZZ,2,range(4))
sage: loads(dumps(A))
[0 1]
[2 3]
```

Sparse OVER $\mathbf{Q}$:

Dense over $\mathbf{Q}[x, y]$:

Dense over finite field.

32.6 Base class for matrices, part 1

For design documentation see sage.matrix.docs.

TESTS:

```
sage: A = Matrix(GF(5),3,3,srange(9))
sage: A == loads(dumps(A))
True
```

class **Matrix**()

adjoint()

Returns the adjoint matrix of self (matrix of cofactors).

INPUT:

- M - a square matrix

OUTPUT:

- N - the adjoint matrix, such that N * M = M * N = M.parent(M.det())

ALGORITHM: Use PARI

EXAMPLES:

```
sage: M = Matrix(ZZ,2,2,[5,2,3,4]) ; M
[5 2]
[3 4]
sage: N = M.adjoint() ; N
[ 4 -2]
[-3  5]
sage: M * N
[14  0]
[ 0 14]
sage: N * M
[14  0]
[ 0 14]
sage: M = Matrix(QQ,2,2,[5/3,2/56,33/13,41/10]) ; M
[  5/3  1/28]
[33/13 41/10]
sage: N = M.adjoint() ; N
[ 41/10  -1/28]
[-33/13    5/3]
sage: M * N
[7363/1092          0]
[         0 7363/1092]
```

TODO: Only implemented for matrices over ZZ or QQ PARI can deal with more general base rings

augment()

Return the augmented matrix of the form:

```
[self | other].
```

EXAMPLES:

```
sage: M = MatrixSpace(QQ,2,2)
sage: A = M([1,2, 3,4])
sage: A
[1 2]
[3 4]
sage: N = MatrixSpace(QQ,2,1)
sage: B = N([9,8])
sage: B
[9]
[8]
sage: A.augment(B)
[1 2 9]
[3 4 8]
sage: B.augment(A)
[9 1 2]
[8 3 4]
sage: M = MatrixSpace(QQ,3,4)
sage: A = M([1,2,3,4, 0,9,8,7, 2/3,3/4,4/5,9/8])
sage: A
[  1   2   3   4]
[  0   9   8   7]
[2/3 3/4 4/5 9/8]
sage: N = MatrixSpace(QQ,3,2)
sage: B = N([1,2, 3,4, 4,5])
sage: B
```

```
[1 2]
[3 4]
[4 5]
sage: A.augment(B)
[  1   2   3   4   1   2]
[  0   9   8   7   3   4]
[2/3 3/4 4/5 9/8   4   5]
sage: B.augment(A)
[  1   2   1   2   3   4]
[  3   4   0   9   8   7]
[  4   5 2/3 3/4 4/5 9/8]
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples

block_sum()

Return the block matrix that has self and other on the diagonal:

```
[ self     0 ]
[    0 other ]
```

EXAMPLES:

```
sage: A = matrix(QQ[['t']], 2, range(1, 5))
sage: A.block_sum(100*A)
[  1   2   0   0]
[  3   4   0   0]
[  0   0 100 200]
[  0   0 300 400]
```

column()

Return the `i`'th column of this matrix as a vector.

This column is a dense vector if and only if the matrix is a dense matrix.

INPUT:

- `i` - integer
- `from_list` - bool (default: False); if true, returns the `i`'th element of `self.columns()` (see `columns()`), which may be faster, but requires building a list of all columns the first time it is called after an entry of the matrix is changed.

EXAMPLES:

```
sage: a = matrix(2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.column(1)
(1, 4)
```

If the column is negative, it wraps around, just like with list indexing, e.g., -1 gives the right-most column:

```
sage: a.column(-1)
(2, 5)
```

TESTS:

```
sage: a = matrix(2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.column(3)
...
IndexError: column index out of range
sage: a.column(-4)
```

```
...
IndexError: column index out of range
```

columns()

Return a list of the columns of self.

INPUT:

- **copy - (default: True) if True, return a copy of the list** of columns which is safe to change.

If self is sparse, returns columns as sparse vectors, and if self is dense returns them as dense vectors.

EXAMPLES:

```
sage: matrix(3, [1..9]).columns()
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
sage: matrix(RR, 2, [sqrt(2), pi, exp(1), 0]).columns()
[(1.41421356237310, 2.71828182845905), (3.14159265358979, 0.000000000000000)]
sage: matrix(RR, 0, 2, []).columns()
[(), ()]
sage: matrix(RR, 2, 0, []).columns()
[]
sage: m = matrix(RR, 3, 3, {(1,2): pi, (2, 2): -1, (0,1): sqrt(2)})
sage: parent(m.columns()[0])
Sparse vector space of dimension 3 over Real Field with 53 bits of precision
```

dense_columns()

Return list of the dense columns of self.

INPUT:

- copy - (default: True) if True, return a copy so you can modify it safely

EXAMPLES:

An example over the integers:

```
sage: a = matrix(3,3,range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: a.dense_columns()
[(0, 3, 6), (1, 4, 7), (2, 5, 8)]
```

We do an example over a polynomial ring:

```
sage: R.<x> = QQ[ ]
sage: a = matrix(R, 2, [x,x^2, 2/3*x,1+x^5]); a
[      x     x^2]
[  2/3*x x^5 + 1]
sage: a.dense_columns()
[(x, 2/3*x), (x^2, x^5 + 1)]
sage: a = matrix(R, 2, [x,x^2, 2/3*x,1+x^5], sparse=True)
sage: c = a.dense_columns(); c
[(x, 2/3*x), (x^2, x^5 + 1)]
sage: parent(c[1])
Ambient free module of rank 2 over the principal ideal domain Univariate Polynomial Ring in
```

dense_matrix()

If this matrix is sparse, return a dense matrix with the same entries. If this matrix is dense, return this matrix (not a copy).

Note: The definition of "dense" and "sparse" in Sage have nothing to do with the number of nonzero entries. Sparse and dense are properties of the underlying representation of the matrix.

EXAMPLES:

```
sage: A = MatrixSpace(QQ,2, sparse=True)([1,2,0,1])
sage: A.is_sparse()
True
sage: B = A.dense_matrix()
sage: B.is_sparse()
False
sage: A*B
[1 4]
[0 1]
sage: A.parent()
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
sage: B.parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

In Sage, the product of a sparse and a dense matrix is always dense:

```
sage: (A*B).parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: (B*A).parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

TESTS:

Make sure that subdivisions are preserved when switching between dense and sparse matrices:

```
sage: a = matrix(ZZ, 3, range(9))
sage: a.subdivide([1,2],2)
sage: a.get_subdivisions()
([1, 2], [2])
sage: b = a.sparse_matrix().dense_matrix()
sage: b.get_subdivisions()
([1, 2], [2])
```

dense_rows()

Return list of the dense rows of self.

INPUT:

- copy - (default: True) if True, return a copy so you can modify it safely (note that the individual vectors in the copy should not be modified since they are mutable!)

EXAMPLES:

```
sage: m = matrix(3, range(9)); m
[0 1 2]
[3 4 5]
[6 7 8]
sage: v = m.dense_rows(); v
[(0, 1, 2), (3, 4, 5), (6, 7, 8)]
sage: v is m.dense_rows()
False
sage: m.dense_rows(copy=False) is m.dense_rows(copy=False)
True
sage: m[0,0] = 10
sage: m.dense_rows()
[(10, 1, 2), (3, 4, 5), (6, 7, 8)]
```

lift()

Return lift of self to the covering ring of the base ring R, which is by definition the ring returned by calling cover_ring() on R, or just R itself if the cover_ring method is not defined.

EXAMPLES:

```
sage: M = Matrix(Integers(7), 2, 2, [5, 9, 13, 15]) ; M
[5 2]
[6 1]
sage: M.lift()
[5 2]
[6 1]
sage: parent(M.lift())
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
```

The field QQ doesn't have a cover_ring method:

```
sage: hasattr(QQ, 'cover_ring')
False
```

So lifting a matrix over QQ gives back the same exact matrix.

```
sage: B = matrix(QQ, 2, [1..4])
sage: B.lift()
[1 2]
[3 4]
sage: B.lift() is B
True
```

matrix_from_columns()

Return the matrix constructed from self using columns with indices in the columns list.

EXAMPLES:

```
sage: M = MatrixSpace(Integers(8),3,3)
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 0]
sage: A.matrix_from_columns([2,1])
[2 1]
[5 4]
[0 7]
```

matrix_from_rows()

Return the matrix constructed from self using rows with indices in the rows list.

EXAMPLES:

```
sage: M = MatrixSpace(Integers(8),3,3)
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 0]
sage: A.matrix_from_rows([2,1])
[6 7 0]
[3 4 5]
```

matrix_from_rows_and_columns()

Return the matrix constructed from self from the given rows and columns.

EXAMPLES:

```
sage: M = MatrixSpace(Integers(8),3,3)
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 0]
sage: A.matrix_from_rows_and_columns([1], [0,2])
```

```
[3 5]
sage: A.matrix_from_rows_and_columns([1,2], [1,2])
[4 5]
[7 0]
```

Note that row and column indices can be reordered or repeated:

```
sage: A.matrix_from_rows_and_columns([2,1], [2,1])
[0 7]
[5 4]
```

For example here we take from row 1 columns 2 then 0 twice, and do this 3 times.

```
sage: A.matrix_from_rows_and_columns([1,1,1],[2,0,0])
[5 3 3]
[5 3 3]
[5 3 3]
```

AUTHORS:

- Jaap Spies (2006-02-18)
- Didier Deshommes: some pyrex speedups implemented

matrix_over_field()

Return copy of this matrix, but with entries viewed as elements of the fraction field of the base ring (assuming it is defined).

EXAMPLES:

```
sage: A = MatrixSpace(IntegerRing(),2)([1,2,3,4])
sage: B = A.matrix_over_field()
sage: B
[1 2]
[3 4]
sage: B.parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

matrix_space()

Return the ambient matrix space of self.

INPUT:

- nrows, ncols - (optional) number of rows and columns in returned matrix space.
- sparse - whether the returned matrix space uses sparse or dense matrices.

EXAMPLES:

```
sage: m = matrix(3, [1..9])
sage: m.matrix_space()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
sage: m.matrix_space(ncols=2)
Full MatrixSpace of 3 by 2 dense matrices over Integer Ring
sage: m.matrix_space(1)
Full MatrixSpace of 1 by 3 dense matrices over Integer Ring
sage: m.matrix_space(1, 2, True)
Full MatrixSpace of 1 by 2 sparse matrices over Integer Ring
```

new_matrix()

Create a matrix in the parent of this matrix with the given number of rows, columns, etc. The default parameters are the same as for self.

INPUT:

These three variables get sent to matrix_space():

- nrows, ncols - number of rows and columns in returned matrix. If not specified, defaults to None and will give a matrix of the same size as self.

- `sparse` - whether returned matrix is sparse or not. Defaults to same value as self.

The remaining three variables (`coerce`, `entries`, and `copy`) are used by `sage.matrix.matrix_space.MatrixSpace()` to construct the new matrix.

Warning: This function called with no arguments returns the zero matrix of the same dimension and sparseness of self.

EXAMPLES:

sage: A = matrix(ZZ,2,2,[1,2,3,4]); A [1 2] [3 4] sage: A.new_matrix() [0 0] [0 0] sage: A.new_matrix(1,1) [0] sage: A.new_matrix(3,3).parent() Full MatrixSpace of 3 by 3 dense matrices over Integer Ring

```
sage: A = matrix(RR,2,3,[1.1,2.2,3.3,4.4,5.5,6.6]); A
[1.10000000000000 2.20000000000000 3.30000000000000]
[4.40000000000000 5.50000000000000 6.60000000000000]
sage: A.new_matrix()
[0.000000000000000 0.000000000000000 0.000000000000000]
[0.000000000000000 0.000000000000000 0.000000000000000]
sage: A.new_matrix().parent()
Full MatrixSpace of 2 by 3 dense matrices over Real Field with 53 bits of precision
```

numpy()

Return the Numpy matrix associated to this matrix.

INPUT:

- `dtype` - The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence.

EXAMPLES:

```
sage: a = matrix(3,range(12))
sage: a.numpy()
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [8, 9, 10, 11]], dtype=object)
sage: a.numpy('f')
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]], dtype=float32)
sage: a.numpy('d')
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]])
sage: a.numpy('B')
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]], dtype=uint8)
```

Type `numpy.typecodes` for a list of the possible typecodes:

```
sage: import numpy
sage: numpy.typecodes
{'All': '?bhilqpBHILQPfdgFDGSUVO', 'AllInteger': 'bBhHiIlLqQpP', 'AllFloat': 'fdgFDG', 'Unsi
```

row()

Return the `i`'th row of this matrix as a vector.

This row is a dense vector if and only if the matrix is a dense matrix.

INPUT:

- `i` - integer

- from_list - bool (default: False); if true, returns the i'th element of self.rows() (see rows()), which may be faster, but requires building a list of all rows the first time it is called after an entry of the matrix is changed.

EXAMPLES:

```
sage: a = matrix(2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.row(0)
(0, 1, 2)
sage: a.row(1)
(3, 4, 5)
sage: a.row(-1)  # last row
(3, 4, 5)
```

TESTS:

```
sage: a = matrix(2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.row(2)
...
IndexError: row index out of range
sage: a.row(-3)
...
IndexError: row index out of range
```

rows()

Return a list of the rows of self.

INPUT:

- copy - (default: True) if True, return a copy of the list of rows which is safe to change.

If self is sparse, returns rows as sparse vectors, and if self is dense returns them as dense vectors.

EXAMPLES:

```
sage: matrix(3, [1..9]).rows()
[(1, 2, 3), (4, 5, 6), (7, 8, 9)]
sage: matrix(RR, 2, [sqrt(2), pi, exp(1), 0]).rows()
[(1.41421356237310, 3.14159265358979), (2.71828182845905, 0.000000000000000)]
sage: matrix(RR, 0, 2, []).rows()
[]
sage: matrix(RR, 2, 0, []).rows()
[(), ()]
sage: m = matrix(RR, 3, 3, {(1,2): pi, (2, 2): -1, (0,1): sqrt(2)})
sage: parent(m.rows()[0])
Sparse vector space of dimension 3 over Real Field with 53 bits of precision
```

set_column()

Sets the entries of column col in self to be the entries of v.

EXAMPLES:

```
sage: A = matrix([[1,2],[3,4]]); A
[1 2]
[3 4]
sage: A.set_column(0, [0,0]); A
[0 2]
[0 4]
sage: A.set_column(1, [0,0]); A
[0 0]
```

```
[0 0]
sage: A.set_column(2, [0,0]); A
...
IndexError: index out of range

sage: A.set_column(0, [0,0,0])
...
ValueError: v must be of length 2
```

set_row()

Sets the entries of row `row` in self to be the entries of `v`.

EXAMPLES:

```
sage: A = matrix([[1,2],[3,4]]); A
[1 2]
[3 4]
sage: A.set_row(0, [0,0]); A
[0 0]
[3 4]
sage: A.set_row(1, [0,0]); A
[0 0]
[0 0]
sage: A.set_row(2, [0,0]); A
...
IndexError: index out of range

sage: A.set_row(0, [0,0,0])
...
ValueError: v must be of length 2
```

sparse_columns()

Return list of the sparse columns of self.

INPUT:

- **copy - (default: True) if True, return a copy so you can** modify it safely

EXAMPLES:

```
sage: a = matrix(2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: v = a.sparse_columns(); v
[(0, 3), (1, 4), (2, 5)]
sage: v[1].is_sparse()
True
```

sparse_matrix()

If this matrix is dense, return a sparse matrix with the same entries. If this matrix is sparse, return this matrix (not a copy).

Note: The definition of "dense" and "sparse" in Sage have nothing to do with the number of nonzero entries. Sparse and dense are properties of the underlying representation of the matrix.

EXAMPLES:

```
sage: A = MatrixSpace(QQ,2, sparse=False)([1,2,0,1])
sage: A.is_sparse()
False
sage: B = A.sparse_matrix()
sage: B.is_sparse()
True
```

```
sage: A
[1 2]
[0 1]
sage: B
[1 2]
[0 1]
sage: A*B
[1 4]
[0 1]
sage: A.parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: B.parent()
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
sage: (A*B).parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: (B*A).parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

sparse_rows()

Return list of the sparse rows of self.

INPUT:

- **copy - (default: True) if True, return a copy so you can** modify it safely

EXAMPLES:

```
sage: m = Mat(ZZ,3,3,sparse=True)(range(9)); m
[0 1 2]
[3 4 5]
[6 7 8]
sage: v = m.sparse_rows(); v
[(0, 1, 2), (3, 4, 5), (6, 7, 8)]
sage: m.sparse_rows(copy=False) is m.sparse_rows(copy=False)
True
sage: v[1].is_sparse()
True
sage: m[0,0] = 10
sage: m.sparse_rows()
[(10, 1, 2), (3, 4, 5), (6, 7, 8)]
```

stack()

Return the augmented matrix self on top of other:

```
[ self  ]
[ other ]
```

EXAMPLES:

```
sage: M = Matrix(QQ, 2, 3, range(6))
sage: N = Matrix(QQ, 1, 3, [10,11,12])
sage: M.stack(N)
[ 0  1  2]
[ 3  4  5]
[10 11 12]
```

submatrix()

Return the matrix constructed from self using the specified range of rows and columns.

INPUT:

- `row`, `col` - index of the starting row and column. Indices start at zero.
- `nrows`, `ncols` - (optional) number of rows and columns to take. If not provided, take all rows below and all columns to the right of the starting entry.

SEE ALSO:

The functions `matrix_from_rows()`, `matrix_from_columns()`, and `matrix_from_rows_and_columns()` allow one to select arbitrary subsets of rows and/or columns.

EXAMPLES:

Take the 3×3 submatrix starting from entry (1,1) in a 4×4 matrix:

```
sage: m = matrix(4, [1..16])
sage: m.submatrix(1, 1)
[ 6  7  8]
[10 11 12]
[14 15 16]
```

Same thing, except take only two rows:

```
sage: m.submatrix(1, 1, 2)
[ 6  7  8]
[10 11 12]
```

And now take only one column:

```
sage: m.submatrix(1, 1, 2, 1)
[ 6]
[10]
```

You can take zero rows or columns if you want:

```
sage: m.submatrix(1, 1, 0)
[]
sage: parent(m.submatrix(1, 1, 0))
Full MatrixSpace of 0 by 3 dense matrices over Integer Ring
```

32.7 Base class for matrices, part 2

TESTS:

```
sage: m = matrix(ZZ['x'], 2, 3, [1..6])
sage: loads(dumps(m)) == m
True
```

class Matrix()

characteristic_polynomial()

Synonym for self.charpoly(...).

EXAMPLES:

```
sage: a = matrix(QQ, 2,2, [1,2,3,4]); a
[1 2]
[3 4]
sage: a.characteristic_polynomial('T')
T^2 - 5*T - 2
```

charpoly()

Return the characteristic polynomial of self, as a polynomial over the base ring.

ALGORITHM: Compute the Hessenberg form of the matrix and read off the characteristic polynomial from that.

If the base ring of the matrix is a number field, use PARI's charpoly instead.

The result is cached.

INPUT:

- `var` - a variable name (default: 'x')
- `algorithm` - string:
- `'hessenberg'` - default (use Hessenberg form of matrix)

EXAMPLES:

First a matrix over $\mathbf{Z}$:

```
sage: A = MatrixSpace(ZZ,2)( [1,2,  3,4] )
sage: f = A.charpoly('x')
sage: f
x^2 - 5*x - 2
sage: f.parent()
Univariate Polynomial Ring in x over Integer Ring
sage: f(A)
[0 0]
[0 0]
```

An example over $\mathbf{Q}$:

```
sage: A = MatrixSpace(QQ,3)(range(9))
sage: A.charpoly('x')
x^3 - 12*x^2 - 18*x
sage: A.trace()
12
sage: A.determinant()
0
```

We compute the characteristic polynomial of a matrix over the polynomial ring $\mathbf{Z}[a]$:

```
sage: R.<a> = PolynomialRing(ZZ)
sage: M = MatrixSpace(R,2)([a,1,  a,a+1]); M
[    a     1]
[    a a + 1]
sage: f = M.charpoly('x'); f
x^2 + (-2*a - 1)*x + a^2
sage: f.parent()
Univariate Polynomial Ring in x over Univariate Polynomial Ring in a over Integer Ring
sage: M.trace()
2*a + 1
sage: M.determinant()
a^2
```

We compute the characteristic polynomial of a matrix over the multi-variate polynomial ring $\mathbf{Z}[x, y]$:

```
sage: R.<x,y> = PolynomialRing(ZZ,2)
sage: A = MatrixSpace(R,2)([x, y, x^2, y^2])
sage: f = A.charpoly('x'); f
x^2 + (-y^2 - x)*x - x^2*y + x*y^2
```

It's a little difficult to distinguish the variables. To fix this, we temporarily view the indeterminate as Z:

```
sage: with localvars(f.parent(), 'Z'): print f
Z^2 + (-y^2 - x)*Z - x^2*y + x*y^2
```

We could also compute f in terms of Z from the start:

```
sage: A.charpoly('Z')
Z^2 + (-y^2 - x)*Z - x^2*y + x*y^2
```

Here is an example over a number field:

```
sage: x = QQ['x'].gen()
sage: K.<a> = NumberField(x^2 - 2)
sage: m = matrix(K, [[a-1, 2], [a, a+1]])
sage: m.charpoly('Z')
Z^2 - 2*a*Z - 2*a + 1
sage: m.charpoly('a')(m) == 0
True
```

TESTS:

```
sage: P.<a,b,c> = PolynomialRing(Rationals())
sage: u = MatrixSpace(P,3)([[0,0,a],[1,0,b],[0,1,c]])
sage: Q.<x> = PolynomialRing(P)
sage: u.charpoly('x')
x^3 - c*x^2 - b*x - a
```

cholesky_decomposition()

Return the Cholesky decomposition of `self`.

INPUT:

The input matrix must be:

- real, symmetric, and positive definite; or
- imaginary, Hermitian, and positive definite.

If not, a `ValueError` exception will be raised.

OUTPUT:

A lower triangular matrix L such that LL^t equals `self`.

ALGORITHM:

Calls the method `_cholesky_decomposition_`, which by default uses a standard recursion.

Warning: This implementation uses a standard recursion that is not known to be numerically stable.

Warning: It is potentially expensive to ensure that the input is positive definite. Therefore this is not checked and it is possible that the output matrix is *not* a valid Cholesky decomposition of `self`. An example of this is given in the tests below.

EXAMPLES:

Here is an example over the real double field; internally, this uses scipy:

```
sage: r = matrix(RDF, 5, 5, [ 0,0,0,0,1, 1,1,1,1,1, 16,8,4,2,1, 81,27,9,3,1, 256,64,16,4,1 ]
sage: m = r * r.transpose(); m
[    1.0     1.0     1.0     1.0     1.0]
[    1.0     5.0    31.0   121.0   341.0]
[    1.0    31.0   341.0  1555.0  4681.0]
[    1.0   121.0  1555.0  7381.0 22621.0]
[    1.0   341.0  4681.0 22621.0 69905.0]
sage: L = m.cholesky_decomposition(); L
[          1.0           0.0           0.0           0.0           0.0]
[          1.0           2.0           0.0           0.0           0.0]
[          1.0          15.0 10.7238052948           0.0           0.0]
[          1.0          60.0 60.9858144589 7.79297342371           0.0]
[          1.0         170.0 198.623524155 39.3665667796 1.72309958068]
sage: L.parent()
Full MatrixSpace of 5 by 5 dense matrices over Real Double Field
sage: L*L.transpose()
[ 1.0     1.0     1.0     1.0     1.0]
[ 1.0     5.0    31.0   121.0   341.0]
```

```
[ 1.0    31.0   341.0  1555.0  4681.0]
[ 1.0   121.0  1555.0  7381.0 22621.0]
[ 1.0   341.0  4681.0 22621.0 69905.0]
sage: ( L*L.transpose() - m ).norm(1) < 2^-30
True
```

Here is an example over a higher precision real field:

```
sage: r = matrix(RealField(100), 5, 5, [ 0,0,0,0,1, 1,1,1,1,1, 16,8,4,2,1, 81,27,9,3,1, 256,
sage: m = r * r.transpose()
sage: L = m.cholesky_decomposition()
sage: L.parent()
Full MatrixSpace of 5 by 5 dense matrices over Real Field with 100 bits of precision
sage: ( L*L.transpose() - m ).norm(1) < 2^-50
True
```

Here is a Hermitian example:

```
sage: r = matrix(CDF, 2, 2, [ 1, -2*I, 2*I, 6 ]); r
[   1.0 -2.0*I]
[ 2.0*I    6.0]
sage: r.eigenvalues()
[0.298437881284, 6.70156211872]
sage: ( r - r.conjugate().transpose() ).norm(1) < 1e-30
True
sage: L = r.cholesky_decomposition(); L
[          1.0             0]
[        2.0*I 1.41421356237]
sage: ( r - L*L.conjugate().transpose() ).norm(1) < 1e-30
True
sage: L.parent()
Full MatrixSpace of 2 by 2 dense matrices over Complex Double Field
```

TESTS:

The following examples are not positive definite:

```
sage: m = -identity_matrix(3).change_ring(RDF)
sage: m.cholesky_decomposition()
...
ValueError: The input matrix was not symmetric and positive definite

sage: m = -identity_matrix(2).change_ring(RealField(100))
sage: m.cholesky_decomposition()
...
ValueError: The input matrix was not symmetric and positive definite
```

Here is a large example over a higher precision complex field:

```
sage: r = MatrixSpace(ComplexField(100), 6, 6).random_element()
sage: m = r * r.conjugate().transpose()
sage: m.change_ring(CDF) # for display purposes
[                    2.5891918451    1.58308081508 - 0.93917354232*I    0.4508660242 - 0.8
[   1.58308081508 + 0.93917354232*I                      3.39096359127  -0.823614467666 - 0.
[   0.4508660242 + 0.898986215453*I  -0.823614467666 + 0.70698381556*I
[ -0.125366701515 + 1.32575360944*I   0.964188058124 + 1.80624774667*I  -1.61505575668 + 0.5
[ -0.161174433016 + 1.92791089094*I   0.884237835922 + 1.12339941545*I    1.16580777654 + 2.
[-0.852634739628 - 0.592301526741*I   -1.14625014365 - 0.64233624728*I    1.22264068801 - 1.
sage: eigs = m.change_ring(CDF).eigenvalues() # again for display purposes
sage: all(imag(e) < 1e-15 for e in eigs)
True
sage: [real(e) for e in eigs]
```

```
[10.463115298, 7.42365754809, 3.36964641458, 1.25904669699, 0.00689184179485, 0.330700789655

sage: ( m - m.conjugate().transpose() ).norm(1) < 1e-50
True
sage: L = m.cholesky_decomposition(); L.change_ring(CDF)
[                      1.60909659284                                   0
[   0.98383205963 + 0.583665111527*I                       1.44304300258
[  0.280198234342 + 0.558690024857*I  -0.987753204014 + 0.222355529831*I
[-0.0779112342122 + 0.823911762252*I   0.388034921026 + 0.658457765816*I  -0.967353506777 +
[  -0.100164548065 + 1.19813247975*I  0.196442380181 - 0.0788779556296*I   0.391945946049 +
[ -0.529884124682 - 0.368095693804*I  -0.284183173327 - 0.408488713349*I      0.738503847 -
sage: ( m - L*L.conjugate().transpose() ).norm(1) < 1e-20
True
sage: L.parent()
Full MatrixSpace of 6 by 6 dense matrices over Complex Field with 100 bits of precision
```

Here is an example that returns an incorrect answer, because the input is *not* positive definite:

```
sage: r = matrix(CDF, 2, 2, [ 1, -2*I, 2*I, 0 ]); r
[   1.0 -2.0*I]
[ 2.0*I      0]
sage: r.eigenvalues()
[2.56155281281, -1.56155281281]
sage: ( r - r.conjugate().transpose() ).norm(1) < 1e-30
True
sage: L = r.cholesky_decomposition(); L
[  1.0     0]
[2.0*I 2.0*I]
sage: L*L.conjugate().transpose()
[   1.0 -2.0*I]
[ 2.0*I    8.0]
```

column_module()

Return the free module over the base ring spanned by the columns of this matrix.

EXAMPLES:

```
sage: t = matrix(QQ, 3, range(9)); t
[0 1 2]
[3 4 5]
[6 7 8]
sage: t.column_module()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
```

column_space()

Return the vector space over the base ring spanned by the columns of this matrix.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,9,-7,4/5,4,3,6,4,3])
sage: A.column_space()
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
sage: W = MatrixSpace(CC,2,2)
```

```
sage: B = W([1, 2+3*I,4+5*I,9]); B
[                      1.00000000000000 2.00000000000000 + 3.00000000000000*I]
[4.00000000000000 + 5.00000000000000*I                      9.00000000000000]
sage: B.column_space()
Vector space of degree 2 and dimension 2 over Complex Field with 53 bits of precision
Basis matrix:
[1.00000000000000                0]
[               0 1.00000000000000]
```

conjugate()

Return the conjugate of self, i.e. the matrix whose entries are the conjugates of the entries of self.

EXAMPLES:

```
sage: A = matrix(CDF, [[1+I,1],[0,2*I]])
sage: A.conjugate()
[1.0 - 1.0*I          1.0]
[           0      -2.0*I]
```

A matrix over a not-totally-real number field:

```
sage: K.<j> = NumberField(x^2+5)
sage: M = matrix(K, [[1+j,1], [0,2*j]])
sage: M.conjugate()
[-j + 1      1]
[     0   -2*j]
```

Conjugates work (trivially) for matrices over rings that embed canonically into the real numbers:

```
sage: M = random_matrix(ZZ, 2)
sage: M == M.conjugate()
True
sage: M = random_matrix(QQ, 3)
sage: M == M.conjugate()
True
sage: M = random_matrix(RR, 2)
sage: M == M.conjugate()
True
```

decomposition()

Returns the decomposition of the free module on which this matrix A acts from the right (i.e., the action is x goes to x A), along with whether this matrix acts irreducibly on each factor. The factors are guaranteed to be sorted in the same way as the corresponding factors of the characteristic polynomial.

Let A be the matrix acting from the on the vector space V of column vectors. Assume that A is square. This function computes maximal subspaces W_1, ..., W_n corresponding to Galois conjugacy classes of eigenvalues of A. More precisely, let $f(X)$ be the characteristic polynomial of A. This function computes the subspace $W_i = ker(g_(A)^n)$, where $g_i(X)$ is an irreducible factor of $f(X)$ and $g_i(X)$ exactly divides $f(X)$. If the optional parameter is_diagonalizable is True, then we let $W_i = ker(g(A))$, since then we know that $ker(g(A)) = ker(g(A)^n)$.

INPUT:

- `self` - a matrix
- `algorithm` - 'spin' (default): algorithm involves iterating the action of self on a vector. 'kernel': naively just compute $ker(f_i(A))$ for each factor f_i.
- `dual` - bool (default: False): If True, also returns the corresponding decomposition of V under the action of the transpose of A. The factors are guaranteed to correspond.
- `is_diagonalizable` - if the matrix is known to be diagonalizable, set this to True, which might speed up the algorithm in some cases.

Note: If the base ring is not a field, the kernel algorithm is used.

OUTPUT:

- Sequence - list of pairs (V,t), where V is a vector spaces and t is a bool, and t is True exactly when the charpoly of self on V is irreducible.
- (optional) list - list of pairs (W,t), where W is a vector space and t is a bool, and t is True exactly when the charpoly of the transpose of self on W is irreducible.

EXAMPLES:

```
sage: A = matrix(ZZ, 4, [3,4,5,6,7,3,8,10,14,5,6,7,2,2,10,9])
sage: B = matrix(QQ, 6, range(36))
sage: B*11
[  0  11  22  33  44  55]
[ 66  77  88  99 110 121]
[132 143 154 165 176 187]
[198 209 220 231 242 253]
[264 275 286 297 308 319]
[330 341 352 363 374 385]
sage: A.decomposition()
[
(Ambient free module of rank 4 over the principal ideal domain Integer Ring, True)
]
sage: B.decomposition()
[
(Vector space of degree 6 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1 -2 -3 -4]
[ 0  1  2  3  4  5], True),
(Vector space of degree 6 and dimension 4 over Rational Field
Basis matrix:
[ 1  0  0  0 -5  4]
[ 0  1  0  0 -4  3]
[ 0  0  1  0 -3  2]
[ 0  0  0  1 -2  1], False)
]
```

decomposition_of_subspace()

Suppose the right action of self on M leaves M invariant. Return the decomposition of M as a list of pairs (W, is_irred) where is_irred is True if the charpoly of self acting on the factor W is irreducible.

Additional inputs besides M are passed onto the decomposition command.

EXAMPLES:

```
sage: t = matrix(QQ, 3, [3, 0, -2, 0, -2, 0, 0, 0, 0]); t
[ 3  0 -2]
[ 0 -2  0]
[ 0  0  0]
sage: t.fcp('X')    # factored charpoly
(X - 3) * X * (X + 2)
sage: v = kernel(t*(t+2)); v    # an invariant subspace
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[0 1 0]
[0 0 1]
sage: D = t.decomposition_of_subspace(v); D
[
(Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 0 1], True),
(Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
```

```
[0 1 0], True)
]
sage: t.restrict(D[0][0])
[0]
sage: t.restrict(D[1][0])
[-2]
```

We do a decomposition over ZZ:

```
sage: a = matrix(ZZ,6,[0, 0, -2, 0, 2, 0, 2, -4, -2, 0, 2, 0, 0, 0, -2, -2, 0, 0, 2, 0, -2,
sage: a.decomposition_of_subspace(ZZ^6)
[
(Free module of degree 6 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0  1 -1  1 -1]
[ 0  1  0 -1  2 -1], False),
(Free module of degree 6 and rank 4 over Integer Ring
Echelon basis matrix:
[ 1  0 -1  0  1  0]
[ 0  1  0  0  0  0]
[ 0  0  0  1  0  0]
[ 0  0  0  0  0  1], False)
]
```

denominator()

Return the least common multiple of the denominators of the elements of self.

If there is no denominator function for the base field, or no LCM function for the denominators, raise a TypeError.

EXAMPLES:

```
sage: A = MatrixSpace(QQ,2)(['1/2', '1/3', '1/5', '1/7'])
sage: A.denominator()
210
```

A trivial example:

```
sage: A = matrix(QQ, 0,2)
sage: A.denominator()
1
```

Denominators are not defined for real numbers:

```
sage: A = MatrixSpace(RealField(),2)([1,2,3,4])
sage: A.denominator()
...
TypeError: denominator not defined for elements of the base ring
```

We can even compute the denominator of matrix over the fraction field of $\mathbf{Z}[x]$.

```
sage: K.<x> = Frac(ZZ['x'])
sage: A = MatrixSpace(K,2)([1/x, 2/(x+1), 1, 5/(x^3)])
sage: A.denominator()
x^4 + x^3
```

Here's an example involving a cyclotomic field:

```
sage: K.<z> = CyclotomicField(3)
sage: M = MatrixSpace(K,3,sparse=True)
sage: A = M([(1+z)/3,(2+z)/3,z/3,1,1+z,-2,1,5,-1+z])
sage: print A
[1/3*z + 1/3 1/3*z + 2/3       1/3*z]
[          1       z + 1          -2]
```

```
[          1           5       z - 1]
sage: print A.denominator()
3
```

density()

Return the density of self.

By density we understand the ration of the number of nonzero positions and the self.nrows() * self.ncols(), i.e. the number of possible nonzero positions.

EXAMPLE:

First, note that the density parameter does not ensure the density of a matrix, it is only an upper bound.

```
sage: A = random_matrix(GF(127),200,200,density=0.3)
sage: A.density()
5159/20000

sage: A = matrix(QQ,3,3,[0,1,2,3,0,0,6,7,8])
sage: A.density()
2/3

sage: a = matrix([[],[],[],[]])
sage: a.density()
0
```

derivative()

Derivative with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

EXAMPLES:

```
sage: v = vector([1,x,x^2])
sage: v.derivative(x)
(0, 1, 2*x)
sage: type(v.derivative(x)) == type(v)
True
sage: v = vector([1,x,x^2], sparse=True)
sage: v.derivative(x)
(0, 1, 2*x)
sage: type(v.derivative(x)) == type(v)
True
sage: v.derivative(x,x)
(0, 0, 2)
```

det()

Synonym for self.determinant(...).

EXAMPLES:

```
sage: A = MatrixSpace(Integers(8),3)([1,7,3, 1,1,1, 3,4,5])
sage: A.det()
6
```

determinant()

Return the determinant of self.

ALGORITHM: For small matrices (n4), this is computed using the naive formula For integral domains, the charpoly is computed (using hessenberg form) Otherwise this is computed using the very stupid expansion by minors stupid *naive generic algorithm*. For matrices over more most rings more sophisticated algorithms can be used. (Type `A.determinant?` to see what is done for a specific matrix A.)

EXAMPLES:

```
sage: A = MatrixSpace(Integers(8),3)([1,7,3, 1,1,1, 3,4,5])
sage: A.determinant()
6
sage: A.determinant() is A.determinant()
True
sage: A[0,0] = 10
sage: A.determinant()
7
```

We compute the determinant of the arbitrary 3x3 matrix:

```
sage: R = PolynomialRing(QQ,9,'x')
sage: A = matrix(R,3,R.gens())
sage: A
[x0 x1 x2]
[x3 x4 x5]
[x6 x7 x8]
sage: A.determinant()
-x2*x4*x6 + x1*x5*x6 + x2*x3*x7 - x0*x5*x7 - x1*x3*x8 + x0*x4*x8
```

We create a matrix over $\mathbf{Z}[x, y]$ and compute its determinant.

```
sage: R.<x,y> = PolynomialRing(IntegerRing(),2)
sage: A = MatrixSpace(R,2)([x, y, x**2, y**2])
sage: A.determinant()
-x^2*y + x*y^2
```

TEST:

```
sage: A = matrix(5, 5, [next_prime(i^2) for i in range(25)])
sage: B = MatrixSpace(ZZ['x'], 5, 5)(A)
sage: A.det() - B.det()
0
```

We verify that trac 5569 is resolved (otherwise the following will hang for hours):

```
sage: d = random_matrix(GF(next_prime(10^20)),50).det()
sage: d = random_matrix(Integers(10^50),50).det()
```

echelon_form()

Return the echelon form of self.

INPUT:

- `matrix` - an element A of a MatrixSpace

OUTPUT:

- `matrix` - The reduced row echelon form of A, as an immutable matrix. Note that self is *not* changed by this command. Use A.echelonize() to change A in place.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(19),2,3)
sage: C = MS.matrix([1,2,3,4,5,6])
sage: C.rank()
2
sage: C.nullity()
0
sage: C.echelon_form()
[ 1  0 18]
[ 0  1  2]
```

echelonize()

Transform self into a matrix in echelon form over the same base ring as self.

INPUT:

- `algorithm` - string, which algorithm to use (default: 'default')
- `'default'` - use a default algorithm, chosen by Sage
- `'strassen'` - use a Strassen divide and conquer algorithm (if available)
- `cutoff` - integer; only used if the Strassen algorithm is selected.

EXAMPLES:

```
sage: a = matrix(QQ,3,range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: a.echelonize()
sage: a
[ 1  0 -1]
[ 0  1  2]
[ 0  0  0]
```

An immutable matrix cannot be transformed into echelon form. Use `self.echelon_form()` instead:

```
sage: a = matrix(QQ,3,range(9)); a.set_immutable()
sage: a.echelonize()
...
ValueError: matrix is immutable; please change a copy instead (use self.copy()).
sage: a.echelon_form()
[ 1  0 -1]
[ 0  1  2]
[ 0  0  0]
```

Echelon form over the integers is what is also classically often known as Hermite normal form:

```
sage: a = matrix(ZZ,3,range(9))
sage: a.echelonize(); a
[ 3  0 -3]
[ 0  1  2]
[ 0  0  0]
```

We compute an echelon form both over a domain and fraction field:

```
sage: R.<x,y> = QQ[]
sage: a = matrix(R, 2, [x,y,x,y])
sage: a.echelon_form()                # not very useful? -- why two copies of the same row?
[x y]
[x y]

sage: b = a.change_ring(R.fraction_field())
sage: b.echelon_form()                # potentially useful
[  1 y/x]
[  0   0]
```

Echelon form is not defined over arbitrary rings:

```
sage: a = matrix(Integers(9),3,range(9))
sage: a.echelon_form()
...
NotImplementedError: Echelon form not implemented over 'Ring of integers modulo 9'.
```

Involving a sparse matrix:

```
sage: m = matrix(3,[1, 1, 1, 1, 0, 2, 1, 2, 0], sparse=True); m
[1 1 1]
[1 0 2]
```

```
[1 2 0]
sage: m.echelon_form()
[ 1  0  2]
[ 0  1 -1]
[ 0  0  0]
sage: m.echelonize(); m
[ 1  0  2]
[ 0  1 -1]
[ 0  0  0]
```

eigenmatrix_left()

Return matrices D and P, where D is a diagonal matrix of eigenvalues and P is the corresponding matrix where the rows are corresponding eigenvectors (or zero vectors) so that P*self = D*P.

EXAMPLES:

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: D, P = A.eigenmatrix_left()
sage: D
[                  0                   0                   0]
[                  0 -1.348469228349535?                   0]
[                  0                   0  13.34846922834954?]
sage: P
[                   1                   -2                    1]
[                   1  0.3101020514433644? -0.3797958971132713?]
[                   1   1.289897948556636?   1.5797958971132712?]
sage: P*A == D*P
True
```

Because P is invertible, A is diagonalizable.

```
sage: A == (~P)*D*P
True
```

The matrix P may contain zero rows corresponding to eigenvalues for which the algebraic multiplicity is greater than the geometric multiplicity. In these cases, the matrix is not diagonalizable.

```
sage: A = jordan_block(2,3); A
[2 1 0]
[0 2 1]
[0 0 2]
sage: A = jordan_block(2,3)
sage: D, P = A.eigenmatrix_left()
sage: D
[2 0 0]
[0 2 0]
[0 0 2]
sage: P
[0 0 1]
[0 0 0]
[0 0 0]
sage: P*A == D*P
True
```

eigenmatrix_right()

Return matrices D and P, where D is a diagonal matrix of eigenvalues and P is the corresponding matrix where the columns are corresponding eigenvectors (or zero vectors) so that self*P = P*D.

EXAMPLES:

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: D, P = A.eigenmatrix_right()
sage: D
[                  0                   0                   0]
[                  0 -1.348469228349535?                   0]
[                  0                   0  13.348469228349543?]
sage: P
[                   1                    1                    1]
[                  -2   0.1303061543300932?   3.069693845669907?]
[                   1 -0.7393876913398137?   5.139387691339814?]
sage: A*P == P*D
True
```

Because P is invertible, A is diagonalizable.

```
sage: A == P*D*(~P)
True
```

The matrix P may contain zero columns corresponding to eigenvalues for which the algebraic multiplicity is greater than the geometric multiplicity. In these cases, the matrix is not diagonalizable.

```
sage: A = jordan_block(2,3); A
[2 1 0]
[0 2 1]
[0 0 2]
sage: A = jordan_block(2,3)
sage: D, P = A.eigenmatrix_right()
sage: D
[2 0 0]
[0 2 0]
[0 0 2]
sage: P
[1 0 0]
[0 0 0]
[0 0 0]
sage: A*P == P*D
True
```

eigenspaces()

Deprecated: Instead of eigenspaces, use eigenspaces_left

eigenspaces_left()

Compute left eigenspaces of a matrix.

If algebraic_multiplicity=False, return a list of pairs (e, V) where e runs through all eigenvalues (up to Galois conjugation) of this matrix, and V is the corresponding left eigenspace.

If algebraic_multiplicity=True, return a list of pairs (e, V, n) where e and V are as above and n is the algebraic multiplicity of the eigenvalue. If the eigenvalues are given symbolically, as roots of an irreducible factor of the characteristic polynomial, then the algebraic multiplicity returned is the multiplicity of each conjugate eigenvalue.

The eigenspaces are returned sorted by the corresponding characteristic polynomials, where polynomials are sorted in dictionary order starting with constant terms.

INPUT:

- `var` - variable name used to represent elements of the root field of each irreducible factor of the characteristic polynomial I.e., if var='a', then the root fields will be in terms of a0, a1, a2, ..., ak.

Warning: Uses a somewhat naive algorithm (simply factors the characteristic polynomial and computes kernels directly over the extension field).

TODO:

Maybe implement the better algorithm that is in dual_eigenvector in sage/modular/hecke/module.py.

EXAMPLES: We compute the left eigenspaces of a 3×3 rational matrix.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenspaces_left(); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with defining polynomi
User basis matrix:
[            1 1/15*a1 + 2/5 2/15*a1 - 1/5])
]
sage: es = A.eigenspaces_left(algebraic_multiplicity=True); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1], 1),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with defining polynomi
User basis matrix:
[            1 1/15*a1 + 2/5 2/15*a1 - 1/5], 1)
]
sage: e, v, n = es[0]; v = v.basis()[0]
sage: delta = e*v - v*A
sage: abs(abs(delta)) < 1e-10
True
```

The same computation, but with implicit base change to a field:

```
sage: A = matrix(ZZ,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.eigenspaces_left()
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with defining polynomi
User basis matrix:
[            1 1/15*a1 + 2/5 2/15*a1 - 1/5])
]
```

We compute the left eigenspaces of the matrix of the Hecke operator T_2 on level 43 modular symbols.

```
sage: A = ModularSymbols(43).T(2).matrix(); A
[ 3  0  0  0  0  0 -1]
[ 0 -2  1  0  0  0  0]
[ 0 -1  1  1  0 -1  0]
[ 0 -1  0 -1  2 -1  1]
[ 0 -1  0  1  1 -1  1]
[ 0  0 -2  0  2 -2  1]
```

```
[ 0  0 -1  0  1  0 -1]
sage: A.base_ring()
Rational Field
sage: f = A.charpoly(); f
x^7 + x^6 - 12*x^5 - 16*x^4 + 36*x^3 + 52*x^2 - 32*x - 48
sage: factor(f)
(x - 3) * (x + 2)^2 * (x^2 - 2)^2
sage: A.eigenspaces_left(algebraic_multiplicity=True)
[
(3, Vector space of degree 7 and dimension 1 over Rational Field
User basis matrix:
[   1    0  1/7    0 -1/7    0 -2/7], 1),
(-2, Vector space of degree 7 and dimension 2 over Rational Field
User basis matrix:
[ 0  1  0  1 -1  1 -1]
[ 0  0  1  0 -1  2 -1], 2),
(a2, Vector space of degree 7 and dimension 2 over Number Field in a2 with defining polynomi
User basis matrix:
[      0       1       0      -1 -a2 - 1       1      -1]
[      0       0       1       0      -1       0 -a2 + 1], 2)
]
```

Next we compute the left eigenspaces over the finite field of order 11:

```
sage: A = ModularSymbols(43, base_ring=GF(11), sign=1).T(2).matrix(); A
[ 3  9  0  0]
[ 0  9  0  1]
[ 0 10  9  2]
[ 0  9  0  2]
sage: A.base_ring()
Finite Field of size 11
sage: A.charpoly()
x^4 + 10*x^3 + 3*x^2 + 2*x + 1
sage: A.eigenspaces_left(var = 'beta')
[
(9, Vector space of degree 4 and dimension 1 over Finite Field of size 11
User basis matrix:
[0 0 1 5]),
(3, Vector space of degree 4 and dimension 1 over Finite Field of size 11
User basis matrix:
[1 6 0 6]),
(beta2, Vector space of degree 4 and dimension 1 over Univariate Quotient Polynomial Ring ir
User basis matrix:
[           0            1            0 5*beta2 + 10])
]
```

TESTS:

Warnings are issued if the generic algorithm is used over inexact fields. Garbage may result in these cases because of numerical precision issues.

```
sage: R=RealField(30)
sage: M=matrix(R,2,[2,1,1,1])
sage: M.eigenspaces_left() # random output from numerical issues
[
(2.6180340, Vector space of degree 2 and dimension 0 over Real Field with 30 bits of precisi
User basis matrix:
[]),
(0.38196601, Vector space of degree 2 and dimension 0 over Real Field with 30 bits of precis
User basis matrix:
```

```
[])
]
sage: R=ComplexField(30)
sage: N=matrix(R,2,[2,1,1,1])
sage: N.eigenspaces_left() # random output from numerical issues
[
(2.6180340, Vector space of degree 2 and dimension 0 over Complex Field with 30 bits of prec
User basis matrix:
[]),
(0.38196601, Vector space of degree 2 and dimension 0 over Complex Field with 30 bits of pre
User basis matrix:
[])
]
```

eigenspaces_right()

Compute right eigenspaces of a matrix.

If algebraic_multiplicity=False, return a list of pairs (e, V) where e runs through all eigenvalues (up to Galois conjugation) of this matrix, and V is the corresponding right eigenspace.

If algebraic_multiplicity=True, return a list of pairs (e, V, n) where e and V are as above and n is the algebraic multiplicity of the eigenvalue. If the eigenvalues are given symbolically, as roots of an irreducible factor of the characteristic polynomial, then the algebraic multiplicity returned is the multiplicity of each conjugate eigenvalue.

The eigenspaces are returned sorted by the corresponding characteristic polynomials, where polynomials are sorted in dictionary order starting with constant terms.

INPUT:

- `var` - variable name used to represent elements of the root field of each irreducible factor of the characteristic polynomial I.e., if var='a', then the root fields will be in terms of a0, a1, a2, ..., ak.

Warning: Uses a somewhat naive algorithm (simply factors the characteristic polynomial and computes kernels directly over the extension field).

TODO: Maybe implement the better algorithm that is in dual_eigenvector in sage/modular/hecke/module.py.

EXAMPLES: We compute the right eigenspaces of a 3×3 rational matrix.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenspaces_right(); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with defining polynomi
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5])
]
sage: es = A.eigenspaces_right(algebraic_multiplicity=True); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1], 1),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with defining polynomi
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5], 1)
```

```
]
sage: e, v, n = es[0]; v = v.basis()[0]
sage: delta = v*e - A*v
sage: abs(abs(delta)) < 1e-10
True
```

The same computation, but with implicit base change to a field:

```
sage: A = matrix(ZZ,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.eigenspaces_right()
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with defining polynomi
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5])
]
```

TESTS: Warnings are issued if the generic algorithm is used over inexact fields. Garbage may result in these cases because of numerical precision issues.

```
sage: R=RealField(30)
sage: M=matrix(R,2,[2,1,1,1])
sage: M.eigenspaces_right() # random output from numerical issues
[(2.6180340,
Vector space of degree 2 and dimension 0 over Real Field with 30 bits of precision
User basis matrix:
[]),
(0.38196601,
Vector space of degree 2 and dimension 0 over Real Field with 30 bits of precision
User basis matrix:
[])]
sage: R=ComplexField(30)
sage: N=matrix(R,2,[2,1,1,1])
sage: N.eigenspaces_right() # random output from numerical issues
[(2.6180340,
Vector space of degree 2 and dimension 0 over Complex Field with 30 bits of precision
User basis matrix:
[]),
(0.38196601,
Vector space of degree 2 and dimension 0 over Complex Field with 30 bits of precision
User basis matrix:
[])]
```

eigenvalues()

Return a sequence of the eigenvalues of a matrix, with multiplicity. If the eigenvalues are roots of polynomials in QQ, then QQbar elements are returned that represent each separate root.

EXAMPLES:

```
sage: a = matrix(QQ, 4, range(16)); a
[ 0  1  2  3]
[ 4  5  6  7]
[ 8  9 10 11]
[12 13 14 15]
sage: sorted(a.eigenvalues(), reverse=True)
[32.464249196572982?, 0, 0, -2.464249196572981?]
```

```
sage: a=matrix([(1, 9, -1, -1), (-2, 0, -10, 2), (-1, 0, 15, -2), (0, 1, 0, -1)])
sage: a.eigenvalues()
[-0.9386318578049146?, 15.506554353532585?, 0.2160387521361705? - 4.713151979747493?*I, 0.216
```

A symmetric matrix a+a.transpose() should have real eigenvalues

```
sage: b=a+a.transpose()
sage: ev = b.eigenvalues(); ev
[-8.350660860579577?, -1.1072479013493797?, 5.718651326708515?, 33.739257435220437?]
```

The eigenvalues are elements of QQbar, so they really represent exact roots of polynomials, not just approximations.

```
sage: e = ev[0]; e
-8.350660860579577?
sage: p = e.minpoly(); p
x^4 - 30*x^3 - 171*x^2 + 1460*x + 1784
sage: p(e) == 0
True
```

To perform computations on the eigenvalue as an element of a number field, you can always convert back to a number field element.

```
sage: e.as_number_field_element()
(Number Field in a with defining polynomial y^4 - 2*y^3 - 507*y^2 + 4988*y - 8744,
-a + 8,
Ring morphism:
From: Number Field in a with defining polynomial y^4 - 2*y^3 - 507*y^2 + 4988*y - 8744
To:   Algebraic Real Field
Defn: a |--> 16.350660860579577?)
```

eigenvectors_left()

Compute the left eigenvectors of a matrix.

For each distinct eigenvalue, returns a list of the form (e,V,n) where e is the eigenvalue, V is a list of eigenvectors forming a basis for the corresponding left eigenspace, and n is the algebraic multiplicity of the eigenvalue.

EXAMPLES: We compute the left eigenvectors of a 3×3 rational matrix.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenvectors_left(); es
[(0, [
(1, -2, 1)
], 1),
(-1.348469228349535?, [(1, 0.3101020514433644?, -0.37975897113271 3?)], 1),
(13.34846922834954?, [(1, 1.289897948556636?, 1.579795897113272?)], 1)]
sage: eval, [evec], mult = es[0]
sage: delta = eval*evec - evec*A
sage: abs(abs(delta)) < 1e-10
True
```

eigenvectors_right()

Compute the right eigenvectors of a matrix.

For each distinct eigenvalue, returns a list of the form (e,V,n) where e is the eigenvalue, V is a list of eigenvectors forming a basis for the corresponding right eigenspace, and n is the algebraic multiplicity of the eigenvalue.

EXAMPLES: We compute the right eigenvectors of a 3×3 rational matrix.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenvectors_right(); es
[(0, [
(1, -2, 1)
], 1),
(-1.348469228349535?, [(1, 0.1303061543300932?, -0.7393876913398137?)], 1),
(13.34846922834954?, [(1, 3.069693845669907?, 5.139387691339814?)], 1)]
sage: eval, [evec], mult = es[0]
sage: delta = eval*evec - A*evec
sage: abs(abs(delta)) < 1e-10
True
```

elementary_divisors()

If self is a matrix over a principal ideal domain R, return elements d_i for $1 \leq i \leq k = \min(r, s)$ where r and s are the number of rows and columns of self, such that the cokernel of self is isomorphic to

$$R/(d_1) \oplus R/(d_2) \oplus R/(d_k)$$

with $d_i \mid d_{i+1}$ for all i. These are the diagonal entries of the Smith form of self (see smith_form()).

EXAMPLES:

```
sage: OE = EquationOrder(x^2 - x + 2, 'w')
sage: w = OE.ring_generators()[0]
sage: m = Matrix([ [1, w],[w,7]])
sage: m.elementary_divisors()
[1, -w + 9]
```

See Also:

smith_form()

exp()

Calculate the exponential of this matrix X, which is the matrix

$$e^X = \sum_{k=0}^{\infty} \frac{X^k}{k!}.$$

This function depends on maxima's matrix exponentiation function, which does not deal well with floating point numbers. If the matrix has floating point numbers, they will be rounded automatically to rational numbers during the computation. If you want approximations to the exponential that are calculated numerically, you may get better results by first converting your matrix to RDF or CDF, as shown in the last example.

EXAMPLES:

```
sage: a=matrix([[1,2],[3,4]])
sage: a.exp()
[-1/22*((sqrt(33) - 11)*e^sqrt(33) - sqrt(33) - 11)*e^(-1/2*sqrt(33) + 5/2)              2/3
[             1/11*(sqrt(33)*e^sqrt(33) - sqrt(33))*e^(-1/2*sqrt(33) + 5/2)  1/22*((sqrt(33)
sage: type(a.exp())
<type 'sage.matrix.matrix_symbolic_dense.Matrix_symbolic_dense'>

sage: a=matrix([[1/2,2/3],[3/4,4/5]])
sage: a.exp()
[-1/418*((3*sqrt(209) - 209)*e^(1/10*sqrt(209)) - 3*sqrt(209) - 209)*e^(-1/20*sqrt(209) + 13
[                  15/418*(sqrt(209)*e^(1/10*sqrt(209)) - sqrt(209))*e^(-1/20*sqrt(209) + 13
```

```
sage: a=matrix(RR,[[1,pi.n()],[1e2,1e-2]])
sage: a.exp()
[ 1/416296432702*((297*sqrt(382784569869489) + 208148216351)*e^(1/551700*sqrt(3827845698694
[                                      30000/208148216351*(sqrt(382784569869489)*e^(1/551700*
sage: a.change_ring(RDF).exp()
[42748127.3153 7368259.24416]
[234538976.138 40426191.4516]
```

fcp()

Return the factorization of the characteristic polynomial of self.

INPUT:

- `var` - (default: 'x') name of variable of charpoly

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,9,-7,4/5,4,3,6,4,3])
sage: A.fcp()
x^3 - 8*x^2 + 209/5*x - 286
sage: A = M([3, 0, -2, 0, -2, 0, 0, 0, 0])
sage: A.fcp('T')
(T - 3) * T * (T + 2)
```

find()

Find elements in this matrix satisfying the constraints in the function f. The function is evaluated on each element of the matrix .

INPUT:

- `f` - a function that is evaluated on each element of this matrix.
- `indices` - whether or not to return the indices and elements of this matrix that satisfy the function.

OUTPUT: If `indices` is not specified, return a matrix with 1 where f is satisfied and 0 where it is not. If `indices` is specified, return a dictionary with containing the elements of this matrix satisfying f.

EXAMPLES:

```
sage: M = matrix(4,3,[1, -1/2, -1, 1, -1, -1/2, -1, 0, 0, 2, 0, 1])
sage: M.find(lambda entry:entry==0)
[0 0 0]
[0 0 0]
[0 1 1]
[0 1 0]

sage: M.find(lambda u:u<0)
[0 1 1]
[0 1 1]
[1 0 0]
[0 0 0]

sage: M = matrix(4,3,[1, -1/2, -1, 1, -1, -1/2, -1, 0, 0, 2, 0, 1])
sage: len(M.find(lambda u:u<1 and u>-1,indices=True))
5

sage: M.find(lambda u:u!=1/2)
[1 1 1]
[1 1 1]
[1 1 1]
[1 1 1]
```

```
sage: M.find(lambda u:u>1.2)
[0 0 0]
[0 0 0]
[0 0 0]
[1 0 0]

sage: sorted(M.find(lambda u:u!=0,indices=True).keys()) == M.nonzero_positions()
True
```

get_subdivisions()

Returns the current subdivision of self.

EXAMPLES:

```
sage: M = matrix(5, 5, range(25))
sage: M.get_subdivisions()
([], [])
sage: M.subdivide(2,3)
sage: M.get_subdivisions()
([2], [3])
sage: N = M.parent()(1)
sage: N.subdivide(M.get_subdivisions()); N
[1 0 0|0 0]
[0 1 0|0 0]
[-----+---]
[0 0 1|0 0]
[0 0 0|1 0]
[0 0 0|0 1]
```

gram_schmidt()

Return the matrix G whose rows are obtained from the rows of self (=A) by applying the Gram-Schmidt orthogonalization process. Also return the coefficients mu ij, i.e., a matrix mu such that `(mu + 1)*G == A`.

OUTPUT:

- `G` - a matrix whose rows are orthogonal
- `mu` - a matrix that gives the transformation, via the relation `(mu + 1)*G == self`

EXAMPLES:

```
sage: A = matrix(ZZ, 3, [-1, 2, 5, -11, 1, 1, 1, -1, -3]); A
[ -1   2   5]
[-11   1   1]
[  1  -1  -3]
sage: G, mu = A.gram_schmidt()
sage: G
[     -1       2       5]
[ -52/5    -1/5      -2]
[  2/187  36/187 -14/187]
sage: mu
[     0      0      0]
[   3/5      0      0]
[  -3/5 -7/187      0]
sage: G.row(0) * G.row(1)
0
sage: G.row(0) * G.row(2)
0
sage: G.row(1) * G.row(2)
0
```

The relation between mu and A is as follows:

```
sage: (mu + 1)*G == A
True
```

hadamard_bound()

Return an int n such that the absolute value of the determinant of this matrix is at most 10^n.

This is got using both the row norms and the column norms.

This function only makes sense when the base field can be coerced to the real double field RDF or the MPFR Real Field with 53-bits precision.

EXAMPLES:

```
sage: a = matrix(ZZ, 3, [1,2,5,7,-3,4,2,1,123])
sage: a.hadamard_bound()
4
sage: a.det()
-2014
sage: 10^4
10000
```

In this example the Hadamard bound has to be computed (automatically) using mpfr instead of doubles, since doubles overflow:

```
sage: a = matrix(ZZ, 2, [2^10000,3^10000,2^50,3^19292])
sage: a.hadamard_bound()
12215
sage: len(str(a.det()))
12215
```

hessenberg_form()

Return Hessenberg form of self.

If the base ring is merely an integral domain (and not a field), the Hessenberg form will (in general) only be defined over the fraction field of the base ring.

EXAMPLES:

```
sage: A = matrix(ZZ,4,[2, 1, 1, -2, 2, 2, -1, -1, -1,1,2,3,4,5,6,7])
sage: h = A.hessenberg_form(); h
[    2  -7/2 -19/5    -2]
[    2   1/2 -17/5    -1]
[    0  25/4  15/2  5/2]
[    0     0  58/5     3]
sage: parent(h)
Full MatrixSpace of 4 by 4 dense matrices over Rational Field
sage: A.hessenbergize()
...
TypeError: Hessenbergize only possible for matrices over a field
```

hessenbergize()

Transform self to Hessenberg form.

The hessenberg form of a matrix A is a matrix that is similar to A, so has the same characteristic polynomial as A, and is upper triangular except possible for entries right below the diagonal.

ALGORITHM: See Henri Cohen's first book.

EXAMPLES:

```
sage: A = matrix(QQ,3, [2, 1, 1, -2, 2, 2, -1, -1, -1])
sage: A.hessenbergize(); A
[  2 3/2   1]
[ -2   3   2]
[  0  -3  -2]
```

```
sage: A = matrix(QQ,4, [2, 1, 1, -2, 2, 2, -1, -1, -1,1,2,3,4,5,6,7])
sage: A.hessenbergize(); A
[    2  -7/2 -19/5    -2]
[    2   1/2 -17/5    -1]
[    0  25/4  15/2   5/2]
[    0     0  58/5     3]
```

You can't Hessenbergize an immutable matrix:

```
sage: A = matrix(QQ, 3, [1..9])
sage: A.set_immutable()
sage: A.hessenbergize()
...
ValueError: matrix is immutable; please change a copy instead (use self.copy()).
```

image()

Return the image of the homomorphism on rows defined by this matrix.

EXAMPLES:

```
sage: MS1 = MatrixSpace(ZZ,4)
sage: MS2 = MatrixSpace(QQ,6)
sage: A = MS1.matrix([3,4,5,6,7,3,8,10,14,5,6,7,2,2,10,9])
sage: B = MS2.random_element()

sage: image(A)
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[  1   0   0 426]
[  0   1   0 518]
[  0   0   1 293]
[  0   0   0 687]

sage: image(B) == B.row_module()
True
```

integer_kernel()

Return the integral kernel of this matrix.

Assume that the base field of this matrix has a numerator and denominator functions for its elements, e.g., it is the rational numbers or a fraction field. This function computes a basis over the integers for the kernel of self.

When kernels are implemented for matrices over general PID's, this function will compute kernels over PID's of matrices over the fraction field of the PID. (todo)

EXAMPLES:

```
sage: A = MatrixSpace(QQ, 4)(range(16))
sage: A.integer_kernel()
Free module of degree 4 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -3  2]
[ 0  1 -2  1]
```

The integer kernel even makes sense for matrices with fractional entries:

```
sage: A = MatrixSpace(QQ, 2)(['1/2',0,  0, 0])
sage: A.integer_kernel()
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1]
```

inverse()

Returns the inverse of self, without changing self.

Note that one can use the Python inverse operator to obtain the inverse as well.

EXAMPLES:

```
sage: m = matrix([[1,2],[3,4]])
sage: m^(-1)
[  -2    1]
[ 3/2 -1/2]
sage: m.inverse()
[  -2    1]
[ 3/2 -1/2]
sage: ~m
[  -2    1]
[ 3/2 -1/2]

sage: m = matrix([[1,2],[3,4]], sparse=True)
sage: m^(-1)
[  -2    1]
[ 3/2 -1/2]
sage: m.inverse()
[  -2    1]
[ 3/2 -1/2]
sage: ~m
[  -2    1]
[ 3/2 -1/2]
```

TESTS:

```
sage: matrix().inverse()
[]
```

is_one()

Return True if this matrix is the identity matrix.

EXAMPLES:

```
sage: m = matrix(QQ,2,range(4))
sage: m.is_one()
False
sage: m = matrix(QQ,2,[5,0,0,5])
sage: m.is_one()
False
sage: m = matrix(QQ,2,[1,0,0,1])
sage: m.is_one()
True
sage: m = matrix(QQ,2,[1,1,1,1])
sage: m.is_one()
False
```

is_scalar()

Return True if this matrix is a scalar matrix.

INPUT

- base_ring element a, which is chosen as self[0][0] if a = None

OUTPUT

- whether self is a scalar matrix (in fact the scalar matrix aI if a is input)

EXAMPLES:

```
sage: m = matrix(QQ,2,range(4))
sage: m.is_scalar(5)
False
sage: m = matrix(QQ,2,[5,0,0,5])
sage: m.is_scalar(5)
True
sage: m = matrix(QQ,2,[1,0,0,1])
sage: m.is_scalar(1)
True
sage: m = matrix(QQ,2,[1,1,1,1])
sage: m.is_scalar(1)
False
```

jordan_form()

Compute the Jordan canonical form of the matrix, if it exists.

This computation is performed in a naive way using the ranks of powers of A-xI, where x is an eigenvalue of the matrix A.

INPUT:

- `base_ring` - ring in which to compute the Jordan form.
- `sparse` - (default False) If sparse=True, return a sparse matrix.
- `subdivide` - (default True) If subdivide=True, the subdivisions for the Jordan blocks in the matrix are shown.
- `transformation` - (default False) If transformation=True, compute also the transformation matrix (see example below).

EXAMPLES:

```
sage: a = matrix(ZZ,4,[1, 0, 0, 0, 0, 1, 0, 0, 1, \
-1, 1, 0, 1, -1, 1, 2]); a
[ 1  0  0  0]
[ 0  1  0  0]
[ 1 -1  1  0]
[ 1 -1  1  2]
sage: a.jordan_form()
[2|0 0|0]
[-+---+-]
[0|1 1|0]
[0|0 1|0]
[-+---+-]
[0|0 0|1]
sage: a.jordan_form(subdivide=False)
[2 0 0 0]
[0 1 1 0]
[0 0 1 0]
[0 0 0 1]
sage: b = matrix(ZZ,3,range(9)); b
[0 1 2]
[3 4 5]
[6 7 8]
sage: b.jordan_form()
...
RuntimeError: Some eigenvalue does not exist in Integer Ring.
sage: b.jordan_form(RealField(15))
[-1.348|0.0000|0.0000]
[------+------+------]
[0.0000|0.0000|0.0000]
[------+------+------]
[0.0000|0.0000| 13.35]
```

If you need the transformation matrix as well as the Jordan form of self, then pass the option transformation=True.

```
sage: m = matrix([[5,4,2,1],[0,1,-1,-1],[-1,-1,3,0],[1,1,-1,2]]); m
[ 5  4  2  1]
[ 0  1 -1 -1]
[-1 -1  3  0]
[ 1  1 -1  2]
sage: jf, p = m.jordan_form(transformation=True)
sage: jf
[2|0|0 0]
[-+-+---]
[0|1|0 0]
[-+-+---]
[0|0|4 1]
[0|0|0 4]
sage: ~p * m * p
[2 0 0 0]
[0 1 0 0]
[0 0 4 1]
[0 0 0 4]
```

Note that for matrices over inexact rings, there can be problems computing the transformation matrix due to numerical stability issues in computing a basis for a kernel.

```
sage: b = matrix(ZZ,3,3,range(9))
sage: jf, p = b.jordan_form(RealField(15), transformation=True)
...
ValueError: cannot compute the transformation matrix due to lack of precision
```

TESTS:

```
sage: c = matrix(ZZ, 3, [1]*9); c
[1 1 1]
[1 1 1]
[1 1 1]
sage: c.jordan_form(subdivide=False)
[3 0 0]
[0 0 0]
[0 0 0]

sage: evals = [(i,i) for i in range(1,6)]
sage: n = sum(range(1,6))
sage: jf = block_diagonal_matrix([jordan_block(ev,size) for ev,size in evals])
sage: p = random_matrix(ZZ,n,n)
sage: while p.rank() != n: p = random_matrix(ZZ,n,n)
sage: m = p * jf * ~p
sage: mjf, mp = m.jordan_form(transformation=True)
sage: mjf == jf
True
sage: m = diagonal_matrix([1,1,0,0])
sage: jf,P = m.jordan_form(transformation=True)
sage: jf == ~P*m*P
True
```

kernel()

Return the (left) kernel of this matrix, as a vector space. This is the space of vectors x such that x*self=0. Use self.right_kernel() for the right kernel, while self.left_kernel() is equivalent to self.kernel().

INPUT: all additional arguments to the kernel function are passed directly onto the echelon call.

By convention if self has 0 rows, the kernel is of dimension 0, whereas the kernel is whole domain if self

has 0 columns.

Note: For information on algorithms used, see the documentation of `right_kernel()` in this class, or versions of right and left kernels in derived classes which override the ones here.

EXAMPLES:

A kernel of dimension one over **Q**:

```
sage: A = MatrixSpace(QQ, 3)(range(9))
sage: A.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

A trivial kernel:

```
sage: A = MatrixSpace(QQ, 2)([1,2,3,4])
sage: A.kernel()
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

Kernel of a zero matrix:

```
sage: A = MatrixSpace(QQ, 2)(0)
sage: A.kernel()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
```

Kernel of a non-square matrix:

```
sage: A = MatrixSpace(QQ,3,2)(range(6))
sage: A.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

The 2-dimensional kernel of a matrix over a cyclotomic field:

```
sage: K = CyclotomicField(12); a=K.0
sage: M = MatrixSpace(K,4,2)([1,-1, 0,-2, 0,-a**2-1, 0,a**2-1])
sage: M
[             1             -1]
[             0             -2]
[             0 -zeta12^2 - 1]
[             0  zeta12^2 - 1]
sage: M.kernel()
Vector space of degree 4 and dimension 2 over Cyclotomic Field of order 12 and degree 4
Basis matrix:
[               0                1                0     -2*zeta12^2]
[               0                0                1 -2*zeta12^2 + 1]
```

A nontrivial kernel over a complicated base field.

```
sage: K = FractionField(PolynomialRing(QQ, 2, 'x'))
sage: M = MatrixSpace(K, 2)([[K.1, K.0], [K.1, K.0]])
sage: M
[x1 x0]
[x1 x0]
sage: M.kernel()
Vector space of degree 2 and dimension 1 over Fraction Field of Multivariate Polynomial Ring
```

```
Basis matrix:
[ 1 -1]
```

We test a trivial left kernel over ZZ:

```
sage: id = matrix(ZZ, 2, 2, [[1, 0], [0, 1]])
sage: id.left_kernel()
Free module of degree 2 and rank 0 over Integer Ring
Echelon basis matrix:
[]
```

Another matrix over ZZ.

```
sage: a = matrix(ZZ,3,1,[1,2,3])
sage: a.left_kernel()
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  1 -1]
[ 0  3 -2]
```

Kernel of a large dense rational matrix, which will invoke the fast IML routines in matrix_integer_dense class. Timing on a 64-bit 3 GHz dual-core machine is about 3 seconds to setup and about 1 second for the kernel() call. Timings that are one or two orders of magnitude larger indicate problems with reaching specialized derived classes.

```
sage: entries = [[1/(i+j+1) for i in srange(500)] for j in srange(500)]
sage: a = matrix(QQ, entries)
sage: a.kernel()
Vector space of degree 500 and dimension 0 over Rational Field
Basis matrix:
0 x 500 dense matrix over Rational Field
```

kernel_on()

Return the kernel of self restricted to the invariant subspace V. The result is a vector subspace of V, which is also a subspace of the ambient space.

INPUT:

- V - vector subspace
- check - (optional) default: True; whether to check that V is invariant under the action of self.
- poly - (optional) default: None; if not None, compute instead the kernel of poly(self) on V.

OUTPUT:

- a subspace

Warning: This function does *not* check that V is in fact invariant under self if check is False. With check False this function is much faster.

EXAMPLES:

```
sage: t = matrix(QQ, 4, [39, -10, 0, -12, 0, 2, 0, -1, 0, 1, -2, 0, 0, 2, 0, -2]); t
[ 39 -10   0 -12]
[  0   2   0  -1]
[  0   1  -2   0]
[  0   2   0  -2]
sage: t.fcp()
(x - 39) * (x + 2) * (x^2 - 2)
sage: s = (t-39)*(t^2-2)
sage: V = s.kernel(); V
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
```

```
[1 0 0 0]
[0 1 0 0]
[0 0 0 1]
sage: s.restrict(V)
[0 0 0]
[0 0 0]
[0 0 0]
sage: s.kernel_on(V)
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 0 1]
sage: k = t-39
sage: k.restrict(V)
[  0 -10 -12]
[  0 -37  -1]
[  0   2 -41]
sage: ker = k.kernel_on(V); ker
Vector space of degree 4 and dimension 1 over Rational Field
Basis matrix:
[   1 -2/7    0 -2/7]
sage: ker.0 * k
(0, 0, 0, 0)
```

left_eigenmatrix()

Return matrices D and P, where D is a diagonal matrix of eigenvalues and P is the corresponding matrix where the rows are corresponding eigenvectors (or zero vectors) so that P*self = D*P.

EXAMPLES:

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: D, P = A.eigenmatrix_left()
sage: D
[                  0                   0                   0]
[                  0 -1.348469228349535?                   0]
[                  0                   0  13.34846922834954?]
sage: P
[                   1                   -2                    1]
[                   1  0.3101020514433644? -0.3797958971132713?]
[                   1   1.289897948556636?   1.579795897113272?]
sage: P*A == D*P
True
```

Because P is invertible, A is diagonalizable.

```
sage: A == (~P)*D*P
True
```

The matrix P may contain zero rows corresponding to eigenvalues for which the algebraic multiplicity is greater than the geometric multiplicity. In these cases, the matrix is not diagonalizable.

```
sage: A = jordan_block(2,3); A
[2 1 0]
[0 2 1]
[0 0 2]
sage: A = jordan_block(2,3)
sage: D, P = A.eigenmatrix_left()
```

```
sage: D
[2 0 0]
[0 2 0]
[0 0 2]
sage: P
[0 0 1]
[0 0 0]
[0 0 0]
sage: P*A == D*P
True
```

left_eigenvectors()

Compute the left eigenvectors of a matrix.

For each distinct eigenvalue, returns a list of the form (e,V,n) where e is the eigenvalue, V is a list of eigenvectors forming a basis for the corresponding left eigenspace, and n is the algebraic multiplicity of the eigenvalue.

EXAMPLES: We compute the left eigenvectors of a 3×3 rational matrix.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenvectors_left(); es
[(0, [
(1, -2, 1)
], 1),
(-1.348469228349535?, [(1, 0.3101020514433644?, -0.3797958971132713?)], 1),
(13.34846922834954?, [(1, 1.289897948556636?, 1.579795897113272?)], 1)]
sage: eval, [evec], mult = es[0]
sage: delta = eval*evec - evec*A
sage: abs(abs(delta)) < 1e-10
True
```

left_kernel()

Return the left kernel of this matrix, as a vector space. This is the space of vectors x such that x*self=0. This is identical to self.kernel(). For a right kernel, use self.right_kernel().

INPUT:

- all additional arguments to the kernel function are passed directly onto the echelon call.

By convention if self has 0 columns, the kernel is of dimension 0, whereas the kernel is whole domain if self has 0 rows.

Note: For information on algorithms used, see the documentation of right_kernel() in this class, or versions of right and left kernels in derived classes which override the ones here.

EXAMPLES:

A left kernel of dimension one over $\mathbf{Q}$:

```
sage: A = MatrixSpace(QQ, 3)(range(9))
sage: A.left_kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

A trivial left kernel:

```
sage: A = MatrixSpace(QQ, 2)([1,2,3,4])
sage: A.left_kernel()
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

Left kernel of a zero matrix:

```
sage: A = MatrixSpace(QQ, 2)(0)
sage: A.left_kernel()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
```

Left kernel of a non-square matrix:

```
sage: A = MatrixSpace(QQ,3,2)(range(6))
sage: A.left_kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

The 2-dimensional left kernel of a matrix over a cyclotomic field:

```
sage: K = CyclotomicField(12); a=K.0
sage: M = MatrixSpace(K,4,2)([1,-1, 0,-2, 0,-a**2-1, 0,a**2-1])
sage: M
[             1             -1]
[             0             -2]
[             0 -zeta12^2 - 1]
[             0  zeta12^2 - 1]
sage: M.left_kernel()
Vector space of degree 4 and dimension 2 over Cyclotomic Field of order 12 and degree 4
Basis matrix:
[               0                1                0     -2*zeta12^2]
[               0                0                1 -2*zeta12^2 + 1]
```

A nontrivial left kernel over a complicated base field.

```
sage: K = FractionField(PolynomialRing(QQ, 2, 'x'))
sage: M = MatrixSpace(K, 2)([[K.1, K.0], [K.1, K.0]])
sage: M
[x1 x0]
[x1 x0]
sage: M.left_kernel()
Vector space of degree 2 and dimension 1 over Fraction Field of Multivariate Polynomial Ring
Basis matrix:
[ 1 -1]
```

Left kernel of a large dense rational matrix, which will invoke the fast IML routines in matrix_integer_dense class. Timing on a 64-bit 3 GHz dual-core machine is about 3 seconds to setup and about 1 second for the kernel() call. Timings that are one or two orders of magnitude larger indicate problems with reaching specialized derived classes.

```
sage: entries = [[1/(i+j+1) for i in srange(500)] for j in srange(500)]
sage: a = matrix(QQ, entries)
sage: a.left_kernel()
Vector space of degree 500 and dimension 0 over Rational Field
Basis matrix:
0 x 500 dense matrix over Rational Field
```

left_nullity()

Return the (left) nullity of this matrix, which is the dimension of the (left) kernel of this matrix acting from the right on row vectors.

EXAMPLES:

```
sage: M = Matrix(QQ,[[1,0,0,1],[0,1,1,0],[1,1,1,0]])
sage: M.nullity()
0
sage: M.left_nullity()
0

sage: A = M.transpose()
sage: A.nullity()
1
sage: A.left_nullity()
1

sage: M = M.change_ring(ZZ)
sage: M.nullity()
0
sage: A = M.transpose()
sage: A.nullity()
1
```

matrix_window()

Return the requested matrix window.

EXAMPLES:

```
sage: A = matrix(QQ, 3, range(9))
sage: A.matrix_window(1,1, 2, 1)
Matrix window of size 2 x 1 at (1,1):
[0 1 2]
[3 4 5]
[6 7 8]
```

We test the optional check flag.

```
sage: matrix([1]).matrix_window(0,1,1,1, check=False)
Matrix window of size 1 x 1 at (0,1):
[1]
sage: matrix([1]).matrix_window(0,1,1,1)
...
IndexError: matrix window index out of range
```

Another test of bounds checking:

```
sage: matrix([1]).matrix_window(1,1,1,1)
...
IndexError: matrix window index out of range
```

maxspin()

Computes the largest integer n such that the list of vectors $S = [v, v*A, ..., v*A^n]$ are linearly independent, and returns that list.

INPUT:

- `self` - Matrix
- `v` - Vector

OUTPUT:

- `list` - list of Vectors

ALGORITHM: The current implementation just adds vectors to a vector space until the dimension doesn't grow. This could be optimized by directly using matrices and doing an efficient Echelon form. Also, when the base is Q, maybe we could simultaneously keep track of what is going on in the reduction modulo p, which might make things much faster.

EXAMPLES:

```
sage: t = matrix(QQ, 3, range(9)); t
[0 1 2]
[3 4 5]
[6 7 8]
sage: v = (QQ^3).0
sage: t.maxspin(v)
[(1, 0, 0), (0, 1, 2), (15, 18, 21)]
sage: k = t.kernel(); k
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
sage: t.maxspin(k.0)
[(1, -2, 1)]
```

minimal_polynomial()

This is a synonym for `self.minpoly`

EXAMPLES:

```
sage: a = matrix(QQ, 4, range(16))
sage: a.minimal_polynomial('z')
z^3 - 30*z^2 - 80*z
sage: a.minpoly()
x^3 - 30*x^2 - 80*x
```

minors()

Return the list of all k-minors of self.

Let A be an m x n matrix and k an integer with 0 < k, k <= m, and k <= n. A k x k minor of A is the determinant of a k x k matrix obtained from A by deleting m - k rows and n - k columns.

The returned list is sorted in lexicographical row major ordering, e.g., if A is a 3 x 3 matrix then the minors returned are with for these rows/columns: [[0, 1]x[0, 1], [0, 1]x[0, 2], [0, 1]x[1, 2], [0, 2]x[0, 1], [0, 2]x[0, 2], [0, 2]x[1, 2], [1, 2]x[0, 1], [1, 2]x[0, 2], [1, 2]x[1, 2]].

INPUT:

- k - integer

EXAMPLE:

```
sage: A = Matrix(ZZ,2,3,[1,2,3,4,5,6]); A
[1 2 3]
[4 5 6]
sage: A.minors(2)
[-3, -6, -3]

sage: k = GF(37)
sage: P.<x0,x1,x2> = PolynomialRing(k)
sage: A = Matrix(P,2,3,[x0*x1, x0, x1, x2, x2 + 16, x2 + 5*x1 ])
sage: A.minors(2)
[x0*x1*x2 + 16*x0*x1 - x0*x2, 5*x0*x1^2 + x0*x1*x2 - x1*x2, 5*x0*x1 + x0*x2 - x1*x2 - 16*x1]
```

minpoly()

Return the minimal polynomial of self.

This uses a simplistic - and potentially very very slow - algorithm that involves computing kernels to determine the powers of the factors of the charpoly that divide the minpoly.

EXAMPLES:

```
sage: A = matrix(GF(9,'c'), 4, [1, 1, 0,0, 0,1,0,0, 0,0,5,0, 0,0,0,5])
sage: factor(A.minpoly())
(x + 1) * (x + 2)^2
sage: A.minpoly()(A) == 0
```

```
True
sage: factor(A.charpoly())
(x + 1)^2 * (x + 2)^2
```

The default variable name is x, but you can specify another name:

```
sage: factor(A.minpoly('y'))
(y + 1) * (y + 2)^2
```

norm()

Return the p-norm of this matrix, where p can be 1, 2, ∞, or the Frobenius norm.

INPUT:

- `self` - a matrix whose entries are coercible into CDF
- `p` - one of the following options:
- `1` - the largest column-sum norm
- `2 (default)` - the Euclidean norm
- `Infinity` - the largest row-sum norm
- `'frob'` - the Frobenius (sum of squares) norm

OUTPUT: RDF number

EXAMPLES:

```
sage: A = matrix(ZZ, [[1,2,4,3],[-1,0,3,-10]])
sage: A.norm(1)
13.0
sage: A.norm(Infinity)
14.0
sage: B = random_matrix(QQ, 20, 21)
sage: B.norm(Infinity) == (B.transpose()).norm(1)
True

sage: Id = identity_matrix(12)
sage: Id.norm(2)
1.0
sage: A = matrix(RR, 2, 2, [13,-4,-4,7])
sage: A.norm()
15.0

sage: A = matrix(CDF, 2, 3, [3*I,4,1-I,1,2,0])
sage: A.norm('frob')
5.65685424949
sage: A.norm(2)
5.47068444321
sage: A.norm(1)
6.0
sage: A.norm(Infinity)
8.41421356237
sage: a = matrix([[],[],[],[]])
sage: a.norm()
0.0
sage: a.norm(Infinity) == a.norm(1)
True
```

nullity()

Return the (left) nullity of this matrix, which is the dimension of the (left) kernel of this matrix acting from the right on row vectors.

EXAMPLES:

```
sage: M = Matrix(QQ,[[1,0,0,1],[0,1,1,0],[1,1,1,0]])
sage: M.nullity()
0
sage: M.left_nullity()
0

sage: A = M.transpose()
sage: A.nullity()
1
sage: A.left_nullity()
1

sage: M = M.change_ring(ZZ)
sage: M.nullity()
0
sage: A = M.transpose()
sage: A.nullity()
1
```

numerical_approx()

Return a numerical approximation of `self` as either a real or complex number with at least the requested number of bits or digits of precision.

INPUT:

- `prec` - an integer: the number of bits of precision
- `digits` - an integer: digits of precision

OUTPUT: A matrix coerced to a real or complex field with prec bits of precision.

EXAMPLES:

```
sage: d = matrix([[3, 0],[0,sqrt(2)]]) ;
sage: b = matrix([[1, -1], [2, 2]]) ; e = b * d * b.inverse();e
[ 1/2*sqrt(2) + 3/2 -1/4*sqrt(2) + 3/4]
[      -sqrt(2) + 3  1/2*sqrt(2) + 3/2]

sage: e.numerical_approx(53)
[ 2.20710678118655 0.396446609406726]
[ 1.58578643762690  2.20710678118655]

sage: e.numerical_approx(20)
[ 2.2071 0.39645]
[ 1.5858  2.2071]

sage: (e-I).numerical_approx(20)
[2.2071 - 1.0000*I           0.39645]
[           1.5858 2.2071 - 1.0000*I]

sage: M=matrix(QQ,4,[i/(i+1) for i in range(12)]);M
[    0   1/2   2/3]
[  3/4   4/5   5/6]
[  6/7   7/8   8/9]
[ 9/10 10/11 11/12]

sage: M.numerical_approx()
[0.000000000000000 0.500000000000000 0.666666666666667]
[0.750000000000000 0.800000000000000 0.833333333333333]
[0.857142857142857 0.875000000000000 0.888888888888889]
[0.900000000000000 0.909090909090909 0.916666666666667]
```

```
sage: matrix(SR, 2, 2, range(4)).n()
[0.000000000000000  1.00000000000000]
[ 2.00000000000000  3.00000000000000]
```

permanent()

Calculate and return the permanent of this $m \times n$ matrix using Ryser's algorithm.

Let $A = (a_{i,j})$ be an $m \times n$ matrix over any commutative ring, with $m \leq n$. The permanent of A is

$$\mathrm{per}(A) = \sum_{\pi} a_{1,\pi(1)} a_{2,\pi(2)} \cdots a_{m,\pi(m)}$$

where the summation extends over all one-to-one functions π from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$.

The product $a_{1,\pi(1)} a_{2,\pi(2)} \cdots a_{m,\pi(m)}$ is called diagonal product. So the permanent of an $m \times n$ matrix A is the sum of all the diagonal products of A.

Modification of theorem 7.1.1. from Brualdi and Ryser: Combinatorial Matrix Theory. Instead of deleting columns from A, we choose columns from A and calculate the product of the row sums of the selected submatrix.

INPUT:

- A - matrix of size m x n with m = n

OUTPUT: permanent of matrix A

EXAMPLES:

```
sage: M = MatrixSpace(ZZ,4,4)
sage: A = M([1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1])
sage: A.permanent()
24

sage: M = MatrixSpace(QQ,3,6)
sage: A = M([1,1,1,1,0,0,0,1,1,1,1,0,0,0,1,1,1,1])
sage: A.permanent()
36

sage: M = MatrixSpace(RR,3,6)
sage: A = M([1.0,1.0,1.0,1.0,0,0,0,1.0,1.0,1.0,1.0,0,0,0,1.0,1.0,1.0,1.0])
sage: A.permanent()
36.0000000000000
```

See Sloane's sequence OEIS A079908(3) = 36, "The Dancing School Problems"

```
sage: print sloane_sequence(79908)                # optional (internet connection)
Searching Sloane's online database...
[79908, 'Solution to the Dancing School Problem with 3 girls and n+3 boys: f(3,n).', [1, 4,

sage: M = MatrixSpace(ZZ,4,5)
sage: A = M([1,1,0,1,1,0,1,1,1,1,1,0,1,0,1,1,1,0,1,0])
sage: A.permanent()
32
```

See Minc: Permanents, Example 2.1, p. 5.

```
sage: M = MatrixSpace(QQ,2,2)
sage: A = M([1/5,2/7,3/2,4/5])
sage: A.permanent()
103/175

sage: R.<a> = PolynomialRing(ZZ)
sage: A = MatrixSpace(R,2)([[a,1], [a,a+1]])
sage: A.permanent()
a^2 + 2*a
```

```
sage: R.<x,y> = PolynomialRing(ZZ,2)
sage: A = MatrixSpace(R,2)([x, y, x^2, y^2])
sage: A.permanent()
x^2*y + x*y^2
```

AUTHORS:

- Jaap Spies (2006-02-16)
- Jaap Spies (2006-02-21): added definition of permanent

permanental_minor()

Calculates the permanental k-minor of a $m \times n$ matrix.

This is the sum of the permanents of all possible k by k submatrices of A.

See Brualdi and Ryser: Combinatorial Matrix Theory, p. 203. Note the typo $p_0(A) = 0$ in that reference! For applications see Theorem 7.2.1 and Theorem 7.2.4.

Note that the permanental m-minor equals $per(A)$.

For a (0,1)-matrix A the permanental k-minor counts the number of different selections of k 1's of A with no two of the 1's on the same line.

INPUT:

- `self` - matrix of size m x n with m = n

OUTPUT: permanental k-minor of matrix A

EXAMPLES:

```
sage: M = MatrixSpace(ZZ,4,4)
sage: A = M([1,0,1,0,1,0,1,0,1,0,10,10,1,0,1,1])
sage: A.permanental_minor(2)
114

sage: M = MatrixSpace(ZZ,3,6)
sage: A = M([1,1,1,1,0,0,0,1,1,1,1,0,0,0,1,1,1,1])
sage: A.permanental_minor(0)
1
sage: A.permanental_minor(1)
12
sage: A.permanental_minor(2)
40
sage: A.permanental_minor(3)
36
```

Note that if k == m the permanental k-minor equals per(A)

```
sage: A.permanent()
36

sage: A.permanental_minor(5)
0
```

For C the "complement" of A:

```
sage: M = MatrixSpace(ZZ,3,6)
sage: C = M([0,0,0,0,1,1,1,0,0,0,0,1,1,1,0,0,0,0])
sage: m, n = 3, 6
sage: sum([(-1)^k * C.permanental_minor(k)*factorial(n-k)/factorial(n-m) for k in range(m+1)
36
```

See Theorem 7.2.1 of Brualdi: and Ryser: Combinatorial Matrix Theory: per(A)

AUTHORS:

- Jaap Spies (2006-02-19)

pivot_rows()

Return the pivot row positions for this matrix, which are a topmost subset of the rows that span the row space and are linearly independent.

OUTPUT:

- `list` - a list of integers

EXAMPLES:

```
sage: A = matrix(QQ,3,3, [0,0,0,1,2,3,2,4,6]); A
[0 0 0]
[1 2 3]
[2 4 6]
sage: A.pivot_rows()
[1]
```

plot()

A plot of this matrix.

Each (ith, jth) matrix element is given a different color value depending on its relative size compared to the other elements in the matrix.

The tick marks drawn on the frame axes denote the (ith, jth) element of the matrix.

This method just calls `matrix_plot`. `*args` and `**kwds` are passed to `matrix_plot`.

EXAMPLES:

A matrix over ZZ colored with different grey levels:

```
sage: A = matrix([[1,3,5,1],[2,4,5,6],[1,3,5,7]])
sage: A.plot()
```

Here we make a random matrix over RR and use cmap='hsv' to color the matrix elements different RGB colors:

```
sage: A = random_matrix(RDF, 50)
sage: plot(A, cmap='hsv')
```

Another random plot, but over GF(389):

```
sage: A = random_matrix(GF(389), 10)
sage: A.plot(cmap='Oranges')
```

prod_of_row_sums()

Calculate the product of all row sums of a submatrix of A for a list of selected columns `cols`.

EXAMPLES:

```
sage: a = matrix(QQ, 2,2, [1,2,3,2]); a
[1 2]
[3 2]
sage: a.prod_of_row_sums([0,1])
15
```

Another example:

```
sage: a = matrix(QQ, 2,3, [1,2,3,2,5,6]); a
[1 2 3]
[2 5 6]
sage: a.prod_of_row_sums([1,2])
55
```

AUTHORS:

- Jaap Spies (2006-02-18)

randomize()

Randomize density proportion of the entries of this matrix, leaving the rest unchanged.

Note: We actually choose at random density proportion of entries of the matrix and set them to random elements. It's possible that the same position can be chosen multiple times, especially for a very small matrix.

INPUT:

- `density` - integer (default: 1) rough measure of the proportion of nonzero entries in the random matrix
- `*args, **kwds` - rest of parameters may be passed to the random_element function of the base ring.

EXAMPLES: We construct the zero matrix over a polynomial ring.

```
sage: a = matrix(QQ['x'], 3); a
[0 0 0]
[0 0 0]
[0 0 0]
```

We then randomize roughly half the entries:

```
sage: a.randomize(0.5)
sage: a
[      1/2*x^2 - x - 12 1/2*x^2 - 1/95*x - 1/2                         0]
[-5/2*x^2 + 2/3*x - 1/4                         0                         0]
[           -x^2 + 2/3*x                         0                         0]
```

Now we randomize all the entries of the resulting matrix:

```
sage: a.randomize()
sage: a
[     1/3*x^2 - x + 1              -x^2 + 1               x^2 - x]
[ -1/14*x^2 - x - 1/4           -4*x - 1/5 -1/4*x^2 - 1/2*x + 4]
[ 1/9*x^2 + 5/2*x - 3     -x^2 + 3/2*x + 1   -2/7*x^2 - x - 1/2]
```

We create the zero matrix over the integers:

```
sage: a = matrix(ZZ, 2); a
[0 0]
[0 0]
```

Then we randomize it; the x and y parameters, which determine the size of the random elements, are passed onto the ZZ random_element method.

```
sage: a.randomize(x=-2^64, y=2^64)
sage: a
[-12401200298100116246   1709403521783430739]
[ -4417091203680573707  17094769731745295000]
```

restrict()

Returns the matrix that defines the action of self on the chosen basis for the invariant subspace V. If V is an ambient, returns self (not a copy of self).

INPUT:

- `V` - vector subspace
- `check` - (optional) default: True; if False may not check that V is invariant (hence can be faster).

OUTPUT: a matrix

Warning: This function returns an nxn matrix, where V has dimension n. It does *not* check that V is in fact invariant under self, unless check is True.

EXAMPLES:

```
sage: V = VectorSpace(QQ, 3)
sage: M = MatrixSpace(QQ, 3)
sage: A = M([1,2,0, 3,4,0, 0,0,0])
sage: W = V.subspace([[1,0,0], [0,1,0]])
sage: A.restrict(W)
[1 2]
[3 4]
sage: A.restrict(W, check=True)
[1 2]
[3 4]
```

We illustrate the warning about invariance not being checked by default, by giving a non-invariant subspace. With the default check=False this function returns the 'restriction' matrix, which is meaningless as check=True reveals.

```
sage: W2 = V.subspace([[1,0,0], [0,1,1]])
sage: A.restrict(W2, check=False)
[1 2]
[3 4]
sage: A.restrict(W2, check=True)
...
ArithmeticError: subspace is not invariant under matrix
```

restrict_codomain()

Suppose that self defines a linear map from some domain to a codomain that contains V and that the image of self is contained in V. This function returns a new matrix A that represents this linear map but as a map to V, in the sense that if x is in the domain, then xA is the linear combination of the elements of the basis of V that equals v*self.

INPUT:

- V - vector space (space of degree `self.ncols()`) that contains the image of self.

See Also:

`restrict()`, `restrict_domain()`

EXAMPLES:

```
sage: A = matrix(QQ,3,[1..9])
sage: V = (QQ^3).span([[1,2,3], [7,8,9]]); V
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: z = vector(QQ,[1,2,5])
sage: B = A.restrict_codomain(V); B
[1 2]
[4 5]
[7 8]
sage: z*B
(44, 52)
sage: z*A
(44, 52, 60)
sage: 44*V.0 + 52*V.1
(44, 52, 60)
```

restrict_domain()

Compute the matrix relative to the basis for V on the domain obtained by restricting self to V, but not changing the codomain of the matrix. This is the matrix whose rows are the images of the basis for V.

INPUT:

- V - vector space (subspace of ambient space on which self acts)

See Also:

restrict()

EXAMPLES:

```
sage: V = QQ^3
sage: A = matrix(QQ,3,[1,2,0, 3,4,0, 0,0,0])
sage: W = V.subspace([[1,0,0], [1,2,3]])
sage: A.restrict_domain(W)
[1 2 0]
[3 4 0]
sage: W2 = V.subspace_with_basis([[1,0,0], [1,2,3]])
sage: A.restrict_domain(W2)
[ 1  2  0]
[ 7 10  0]
```

right_eigenmatrix()

Return matrices D and P, where D is a diagonal matrix of eigenvalues and P is the corresponding matrix where the columns are corresponding eigenvectors (or zero vectors) so that self*P = P*D.

EXAMPLES:

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: D, P = A.eigenmatrix_right()
sage: D
[                  0                   0                   0]
[                  0 -1.348469228349535?                   0]
[                  0                   0  13.34846922834954?]
sage: P
[                   1                    1                    1]
[                  -2   0.1303061543300932?   3.069693845669907?]
[                   1 -0.7393876913398137?   5.139387691339814?]
sage: A*P == P*D
True
```

Because P is invertible, A is diagonalizable.

```
sage: A == P*D*(~P)
True
```

The matrix P may contain zero columns corresponding to eigenvalues for which the algebraic multiplicity is greater than the geometric multiplicity. In these cases, the matrix is not diagonalizable.

```
sage: A = jordan_block(2,3); A
[2 1 0]
[0 2 1]
[0 0 2]
sage: A = jordan_block(2,3)
sage: D, P = A.eigenmatrix_right()
sage: D
[2 0 0]
[0 2 0]
[0 0 2]
sage: P
[1 0 0]
[0 0 0]
[0 0 0]
sage: A*P == P*D
True
```

right_eigenspaces()

Compute right eigenspaces of a matrix.

If algebraic_multiplicity=False, return a list of pairs (e, V) where e runs through all eigenvalues (up to Galois conjugation) of this matrix, and V is the corresponding right eigenspace.

If algebraic_multiplicity=True, return a list of pairs (e, V, n) where e and V are as above and n is the algebraic multiplicity of the eigenvalue. If the eigenvalues are given symbolically, as roots of an irreducible factor of the characteristic polynomial, then the algebraic multiplicity returned is the multiplicity of each conjugate eigenvalue.

The eigenspaces are returned sorted by the corresponding characteristic polynomials, where polynomials are sorted in dictionary order starting with constant terms.

INPUT:

- `var` - variable name used to represent elements of the root field of each irreducible factor of the characteristic polynomial I.e., if var='a', then the root fields will be in terms of a0, a1, a2, ..., ak.

Warning: Uses a somewhat naive algorithm (simply factors the characteristic polynomial and computes kernels directly over the extension field).

TODO: Maybe implement the better algorithm that is in dual_eigenvector in sage/modular/hecke/module.py.

EXAMPLES: We compute the right eigenspaces of a 3×3 rational matrix.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenspaces_right(); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with defining polynomi
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5])
]
sage: es = A.eigenspaces_right(algebraic_multiplicity=True); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1], 1),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with defining polynomi
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5], 1)
]
sage: e, v, n = es[0]; v = v.basis()[0]
sage: delta = v*e - A*v
sage: abs(abs(delta)) < 1e-10
True
```

The same computation, but with implicit base change to a field:

```
sage: A = matrix(ZZ,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.eigenspaces_right()
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
```

```
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with defining polynomi
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5])
]
```

TESTS: Warnings are issued if the generic algorithm is used over inexact fields. Garbage may result in these cases because of numerical precision issues.

```
sage: R=RealField(30)
sage: M=matrix(R,2,[2,1,1,1])
sage: M.eigenspaces_right() # random output from numerical issues
[(2.6180340,
Vector space of degree 2 and dimension 0 over Real Field with 30 bits of precision
User basis matrix:
[]),
(0.38196601,
Vector space of degree 2 and dimension 0 over Real Field with 30 bits of precision
User basis matrix:
[])]
sage: R=ComplexField(30)
sage: N=matrix(R,2,[2,1,1,1])
sage: N.eigenspaces_right() # random output from numerical issues
[(2.6180340,
Vector space of degree 2 and dimension 0 over Complex Field with 30 bits of precision
User basis matrix:
[]),
(0.38196601,
Vector space of degree 2 and dimension 0 over Complex Field with 30 bits of precision
User basis matrix:
[])]
```

right_eigenvectors()

Compute the right eigenvectors of a matrix.

For each distinct eigenvalue, returns a list of the form (e,V,n) where e is the eigenvalue, V is a list of eigenvectors forming a basis for the corresponding right eigenspace, and n is the algebraic multiplicity of the eigenvalue.

EXAMPLES: We compute the right eigenvectors of a 3×3 rational matrix.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenvectors_right(); es
[(0, [
(1, -2, 1)
], 1),
(-1.348469228349535?, [(1, 0.1303061543300932?, -0.7393876913398137?)], 1),
(13.34846922834954?, [(1, 3.069693845669907?, 5.139387691339814?)], 1)]
sage: eval, [evec], mult = es[0]
sage: delta = eval*evec - A*evec
sage: abs(abs(delta)) < 1e-10
True
```

right_kernel()

Return the right kernel of this matrix, as a vector space. This is the space of vectors x such that self*x=0. A left kernel can be found with self.left_kernel() or just self.kernel().

INPUT: all additional arguments to the kernel function are passed directly onto the echelon call.

By convention if self has 0 columns, the kernel is of dimension 0, whereas the kernel is whole domain if self has 0 rows.

ALGORITHM:

Elementary row operations do not change the right kernel, since they are left multiplication by an invertible matrix, so we instead compute the kernel of the row echelon form. More precisely, there is a basis vector of the kernel that corresponds to each non-pivot column. That vector has a 1 at the non-pivot column, 0's at all other non-pivot columnss, and for each pivot column, the negative of the entry at the non-pivot column in the row with that pivot element.

Note: Preference is given to left kernels in that the generic method name `kernel()` returns a left kernel. However most computations of kernels are implemented as right kernels.

EXAMPLES:

A right kernel of dimension one over $\mathbb{Q}$:

```
sage: A = MatrixSpace(QQ, 3)(range(9))
sage: A.right_kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

A trivial right kernel:

```
sage: A = MatrixSpace(QQ, 2)([1,2,3,4])
sage: A.right_kernel()
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
```

Right kernel of a zero matrix:

```
sage: A = MatrixSpace(QQ, 2)(0)
sage: A.right_kernel()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
```

Right kernel of a non-square matrix:

```
sage: A = MatrixSpace(QQ,2,3)(range(6))
sage: A.right_kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
```

The 2-dimensional right kernel of a matrix over a cyclotomic field:

```
sage: K = CyclotomicField(12); a=K.0
sage: M = MatrixSpace(K,2,4)([1,-1, 0,-2, 0,-a**2-1, 0,a**2-1])
sage: M
[            1            -1             0            -2]
[            0 -zeta12^2 - 1             0  zeta12^2 - 1]
sage: M.right_kernel()
Vector space of degree 4 and dimension 2 over Cyclotomic Field of order 12 and degree 4
Basis matrix:
[      1  4/13*zeta12^2 - 1/13      0 -2/13*zeta12^2 + 7/13]
[      0                    0      1                    0]
```

A nontrivial right kernel over a complicated base field.

```
sage: K = FractionField(PolynomialRing(QQ, 2, 'x'))
sage: M = MatrixSpace(K, 2)([[K.1, K.0], [K.1, K.0]])
sage: M
[x1 x0]
[x1 x0]
sage: M.right_kernel()
Vector space of degree 2 and dimension 1 over Fraction Field of Multivariate Polynomial Ring
Basis matrix:
[ 1 x1/(-x0)]
```

Right kernel of a large dense rational matrix, which will invoke the fast IML routines in matrix_integer_dense class. Timing on a 64-bit 3 GHz dual-core machine is about 3 seconds to setup and about 1 second for the kernel() call. Timings that are one or two orders of magnitude larger indicate problems with reaching specialized derived classes.

```
sage: entries = [[1/(i+j+1) for i in srange(500)] for j in srange(500)]
sage: a = matrix(QQ, entries)
sage: a.right_kernel()
Vector space of degree 500 and dimension 0 over Rational Field
Basis matrix:
0 x 500 dense matrix over Rational Field
```

right_nullity()

Return the right nullity of this matrix, which is the dimension of the right kernel.

EXAMPLES:

```
sage: A = MatrixSpace(QQ,3,2)(range(6))
sage: A.right_nullity()
0

sage: A = matrix(ZZ,3,range(9))
sage: A.right_nullity()
1
```

rook_vector()

Returns rook vector of this matrix.

Let A be a general m by n (0,1)-matrix with $m \leq n$. We identify A with a chessboard where rooks can be placed on the fields corresponding with $a_{ij} = 1$. The number $r_k = p_k(A)$ (the permanental k-minor) counts the number of ways to place k rooks on this board so that no two rooks can attack another.

The rook vector is the list consisting of $r_0, r_1, \ldots, r_m$.

The rook polynomial is defined by $r(x) = \sum_{k=0}^{m} r_k x^k$.

INPUT:

- `self` - m by n matrix with m = n
- `check` - True or False (default), optional

OUTPUT: rook vector

EXAMPLES:

```
sage: M = MatrixSpace(ZZ,3,6)
sage: A = M([1,1,1,1,0,0,0,1,1,1,1,0,0,0,1,1,1,1])
sage: A.rook_vector()
[1, 12, 40, 36]

sage: R.<x> = PolynomialRing(ZZ)
sage: rv = A.rook_vector()
sage: rook_polynomial = sum([rv[k] * x^k for k in range(len(rv))])
sage: rook_polynomial
36*x^3 + 40*x^2 + 12*x + 1
```

AUTHORS:

- Jaap Spies (2006-02-24)

row_module()

Return the free module over the base ring spanned by the rows of self.

EXAMPLES:

```
sage: A = MatrixSpace(IntegerRing(), 2)([1,2,3,4])
sage: A.row_module()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 0]
[0 2]
```

row_space()

Return the row space of this matrix. (Synonym for self.row_module().)

EXAMPLES:

```
sage: t = matrix(QQ, 3, range(9)); t
[0 1 2]
[3 4 5]
[6 7 8]
sage: t.row_space()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]

sage: m = Matrix(Integers(5),2,2,[2,2,2,2]);
sage: m.row_space()
Vector space of degree 2 and dimension 1 over Ring of integers modulo 5
Basis matrix:
[1 1]
```

set_block()

Sets the sub-matrix of self, with upper left corner given by row, col to block.

EXAMPLES:

```
sage: A = matrix(QQ, 3, 3, range(9))/2
sage: B = matrix(ZZ, 2, 1, [100,200])
sage: A.set_block(0, 1, B)
sage: A
[  0 100   1]
[3/2 200 5/2]
[  3 7/2   4]
```

We test that an exception is raised when the block is out of bounds:

```
sage: matrix([1]).set_block(0,1,matrix([1]))
...
IndexError: matrix window index out of range
```

smith_form()

If self is a matrix over a principal ideal domain R, return matrices D, U, V over R such that D = U * self * V, U and V have unit determinant, and D is diagonal with diagonal entries the ordered elementary divisors of self, ordered so that $D_i \mid D_{i+1}$. Note that U and V are not uniquely defined in general, and D is defined only up to units.

INPUT:

- `self` - a matrix over an integral domain. If the base ring is not a PID, the routine might work, or else it will fail having found an example of a non-principal ideal. Note that we do not call any methods

to check whether or not the base ring is a PID, since this might be quite expensive (e.g. for rings of integers of number fields of large degree).

ALGORITHM: Lifted wholesale from http://en.wikipedia.org/wiki/Smith_normal_form

See Also:

`elementary_divisors()`

AUTHORS:

- David Loeffler (2008-12-05)

EXAMPLES:

An example over the ring of integers of a number field (of class number 1):

```
sage: OE = NumberField(x^2 - x + 2,'w').ring_of_integers()
sage: w = OE.ring_generators()[0]
sage: m = Matrix([ [1, w],[w,7]])
sage: d, u, v = m.smith_form()
sage: (d, u, v)
([     1      0]
[     0 -w + 9], [ 1  0]
[-w  1], [ 1 -w]
[ 0  1])
sage: u * m * v == d
True
sage: u.base_ring() == v.base_ring() == d.base_ring() == OE
True
sage: u.det().is_unit() and v.det().is_unit()
True
```

An example over the polynomial ring QQ[x]:

```
sage: R.<x> = QQ[]; m=x*matrix(R,2,2,1) - matrix(R, 2,2,[3,-4,1,-1]); m.smith_form()
([            1             0]
[            0 x^2 - 2*x + 1], [    0    -1]
[    1 x - 3], [    1 x + 1]
[    0     1])
```

An example over a field:

```
sage: m = matrix( GF(17), 3, 3, [11,5,1,3,6,8,1,16,0]); d,u,v = m.smith_form()
sage: d
[1 0 0]
[0 1 0]
[0 0 0]
sage: u*m*v == d
True
```

Some examples over non-PID's work anyway:

```
sage: R = EquationOrder(x^2 + 5, 's') # class number 2
sage: s = R.ring_generators()[0]
sage: matrix(R, 2, 2, [s-1,-s,-s,2*s+1]).smith_form()
([     1      0]
[     0 -s - 6],
[   -1    -1]
[    s s - 1],
[    1 s + 1]
[    0     1])
```

Others don't, but they fail quite constructively:

```
sage: matrix(R,2,2,[s-1,-s-2,-2*s,-s-2]).smith_form()
...
ArithmeticError: Ideal Fractional ideal (2, s + 1) not principal
```

Empty matrices are handled safely:

```
sage: m = MatrixSpace(OE, 2,0)(0); d,u,v=m.smith_form(); u*m*v == d
True
sage: m = MatrixSpace(OE, 0,2)(0); d,u,v=m.smith_form(); u*m*v == d
True
sage: m = MatrixSpace(OE, 0,0)(0); d,u,v=m.smith_form(); u*m*v == d
True
```

Some pathological cases that crashed earlier versions:

```
sage: m = Matrix(OE, [[2*w,2*w-1,-w+1],[2*w+2,-2*w-1,w-1],[-2*w-1,-2*w-2,2*w-1]]); d, u, v =
True
sage: m = matrix(OE, 3, 3, [-5*w-1,-2*w-2,4*w-10,8*w,-w,w-1,-1,1,-8]); d,u,v = m.smith_form(
True
```

solve_left()

If self is a matrix A, then this function returns a vector or matrix X such that $XA = B$. If B is a vector then X is a vector and if B is a matrix, then X is a matrix.

INPUT:

- B - a matrix
- check - bool (default: True) - if False and self is nonsquare, may not raise an error message even if there is no solution. This is faster but more dangerous.

EXAMPLES:

```
sage: A = matrix(QQ,4,2, [0, -1, 1, 0, -2, 2, 1, 0])
sage: B = matrix(QQ,2,2, [1, 0, 1, -1])
sage: X = A.solve_left(B)
sage: X*A == B
True
```

solve_right()

If self is a matrix A, then this function returns a vector or matrix X such that $AX = B$. If B is a vector then X is a vector and if B is a matrix, then X is a matrix.

Note: In Sage one can also write A \backslash B for A.solve_right(B), i.e., Sage implements the "the MATLAB/Octave backslash operator".

INPUT:

- B - a matrix or vector
- check - bool (default: True) - if False and self is nonsquare, may not raise an error message even if there is no solution. This is faster but more dangerous.

OUTPUT: a matrix or vector

See Also:

solve_left()

EXAMPLES:

```
sage: A = matrix(QQ, 3, [1,2,3,-1,2,5,2,3,1])
sage: b = vector(QQ,[1,2,3])
sage: x = A \ b; x
(-13/12, 23/12, -7/12)
sage: A * x
(1, 2, 3)
```

We solve with A nonsquare:

```
sage: A = matrix(QQ,2,4, [0, -1, 1, 0, -2, 2, 1, 0]); B = matrix(QQ,2,2, [1, 0, 1, -1])
sage: X = A.solve_right(B); X
[-3/2  1/2]
[  -1    0]
[   0    0]
[   0    0]
sage: A*X == B
True
```

Another nonsingular example:

```
sage: A = matrix(QQ,2,3, [1,2,3,2,4,6]); v = vector([-1/2,-1])
sage: x = A \ v; x
(-1/2, 0, 0)
sage: A*x == v
True
```

Same example but over $\mathbf{Z}$:

```
sage: A = matrix(ZZ,2,3, [1,2,3,2,4,6]); v = vector([-1,-2])
sage: A \ v
(-1, 0, 0)
```

An example in which there is no solution:

```
sage: A = matrix(QQ,2,3, [1,2,3,2,4,6]); v = vector([1,1])
sage: A \ v
...
ValueError: matrix equation has no solutions
```

A ValueError is raised if the input is invalid:

```
sage: A = matrix(QQ,4,2, [0, -1, 1, 0, -2, 2, 1, 0])
sage: B = matrix(QQ,2,2, [1, 0, 1, -1])
sage: X = A.solve_right(B)
...
ValueError: number of rows of self must equal number of rows of B
```

We solve with A singular:

```
sage: A = matrix(QQ,2,3, [1,2,3,2,4,6]); B = matrix(QQ,2,2, [6, -6, 12, -12])
sage: X = A.solve_right(B); X
[ 6 -6]
[ 0  0]
[ 0  0]
sage: A*X == B
True
```

We illustrate left associativity, etc., of the backslash operator.

```
sage: A = matrix(QQ, 2, [1,2,3,4])
sage: A \ A
[1 0]
[0 1]
sage: A \ A \ A
[1 2]
[3 4]
sage: A.parent()(1) \ A
[1 2]
[3 4]
sage: A \ (A \ A)
[  -2    1]
[ 3/2 -1/2]
```

```
sage: X = A \ (A - 2); X
[ 5 -2]
[-3  2]
sage: A * X
[-1  2]
[ 3  2]
```

Solving over a polynomial ring:

```
sage: x = polygen(QQ, 'x')
sage: A = matrix(2, [x,2*x,-5*x^2+1,3])
sage: v = vector([3,4*x - 2])
sage: X = A \ v
sage: X
((-8*x^2 + 4*x + 9)/(10*x^3 + x), (19*x^2 - 2*x - 3)/(10*x^3 + x))
sage: A * X == v
True
```

Solving a system over the p-adics:

```
sage: k = Qp(5,4)
sage: a = matrix(k, 3, [1,7,3,2,5,4,1,1,2]); a
[    1 + O(5^4) 2 + 5 + O(5^4)     3 + O(5^4)]
[    2 + O(5^4)     5 + O(5^5)     4 + O(5^4)]
[    1 + O(5^4)     1 + O(5^4)     2 + O(5^4)]
sage: v = vector(k, 3, [1,2,3])
sage: x = a \ v; x
(4 + 5 + 5^2 + 3*5^3 + O(5^4), 2 + 5 + 3*5^2 + 5^3 + O(5^4), 1 + 5 + O(5^4))
sage: a * x == v
True
```

subdivide()

Divides self into logical submatrices which can then be queried and extracted. If a subdivision already exists, this method forgets the previous subdivision and flushes the cache.

INPUT:

- `row_lines` - None, an integer, or a list of integers
- `col_lines` - None, an integer, or a list of integers

OUTPUT: changes self

Note: One may also pass a tuple into the first argument which will be interpreted as (row_lines, col_lines)

EXAMPLES:

```
sage: M = matrix(5, 5, prime_range(100))
sage: M.subdivide(2,3); M
[ 2  3  5| 7 11]
[13 17 19|23 29]
[--------+-----]
[31 37 41|43 47]
[53 59 61|67 71]
[73 79 83|89 97]
sage: M.subdivision(0,0)
[ 2  3  5]
[13 17 19]
sage: M.subdivision(1,0)
[31 37 41]
[53 59 61]
[73 79 83]
sage: M.subdivision_entry(1,0,0,0)
31
```

```
sage: M.get_subdivisions()
([2], [3])
sage: M.subdivide(None, [1,3]); M
[ 2| 3  5| 7 11]
[13|17 19|23 29]
[31|37 41|43 47]
[53|59 61|67 71]
[73|79 83|89 97]
```

Degenerate cases work too.

```
sage: M.subdivide([2,5], [0,1,3]); M
[| 2| 3  5| 7 11]
[|13|17 19|23 29]
[+--+-----+-----]
[|31|37 41|43 47]
[|53|59 61|67 71]
[|73|79 83|89 97]
[+--+-----+-----]
sage: M.subdivision(0,0)
[]
sage: M.subdivision(0,1)
[ 2]
[13]
sage: M.subdivide([2,2,3], [0,0,1,1]); M
[|| 2|| 3  5  7 11]
[||13||17 19 23 29]
[++--++-----------]
[++--++-----------]
[||31||37 41 43 47]
[++--++-----------]
[||53||59 61 67 71]
[||73||79 83 89 97]
sage: M.subdivision(0,0)
[]
sage: M.subdivision(2,4)
[37 41 43 47]
```

AUTHORS:

- Robert Bradshaw (2007-06-14)

subdivision()

Returns in immutable copy of the (i,j)th submatrix of self, according to a previously set subdivision.

Before a subdivision is set, the only valid arguments are (0,0) which returns self.

EXAMPLE:

```
sage: M = matrix(3, 4, range(12))
sage: M.subdivide(1,2); M
[ 0  1| 2  3]
[-----+-----]
[ 4  5| 6  7]
[ 8  9|10 11]
sage: M.subdivision(0,0)
[0 1]
sage: M.subdivision(0,1)
[2 3]
sage: M.subdivision(1,0)
[4 5]
[8 9]
```

It handles size-zero subdivisions as well.

```
sage: M = matrix(3, 4, range(12))
sage: M.subdivide([0],[0,2,2,4]); M
[+-----++-----+]
[| 0  1|| 2  3|]
[| 4  5|| 6  7|]
[| 8  9||10 11|]
sage: M.subdivision(0,0)
[]
sage: M.subdivision(1,1)
[0 1]
[4 5]
[8 9]
sage: M.subdivision(1,2)
[]
sage: M.subdivision(1,0)
[]
sage: M.subdivision(0,1)
[]
```

subdivision_entry()

Returns the x,y entry of the i,j submatrix of self.

EXAMPLES:

```
sage: M = matrix(5, 5, range(25))
sage: M.subdivide(3,3); M
[ 0  1  2| 3  4]
[ 5  6  7| 8  9]
[10 11 12|13 14]
[--------+-----]
[15 16 17|18 19]
[20 21 22|23 24]
sage: M.subdivision_entry(0,0,1,2)
7
sage: M.subdivision(0,0)[1,2]
7
sage: M.subdivision_entry(0,1,0,0)
3
sage: M.subdivision_entry(1,0,0,0)
15
sage: M.subdivision_entry(1,1,1,1)
24
```

Even though this entry exists in the matrix, the index is invalid for the submatrix.

```
sage: M.subdivision_entry(0,0,4,0)
...
IndexError: Submatrix 0,0 has no entry 4,0
```

subs()

EXAMPLES:

```
sage: var('a,b,d,e')
(a, b, d, e)
sage: m = matrix([[a,b], [d,e]])
sage: m.substitute(a=1)
[1 b]
[d e]
sage: m.subs(a=b, b=d)
```

```
[b d]
[d e]
```

symplectic_form()

Find a symplectic form for self if self is an anti-symmetric, alternating matrix defined over a field.

Returns a pair (F, C) such that the rows of C form a symplectic basis for self and F = C * self * C.transpose().

Raises a ValueError if not over a field, or self is not anti-symmetric, or self is not alternating.

Anti-symmetric means that $M = -M^t$. Alternating means that the diagonal of M is identically zero.

A symplectic basis is a basis of the form $e_1, \ldots, e_j, f_1, \ldots f_j, z_1, \ldots, z_k$ such that

- $z_i M v^t = 0$ for all vectors v
- $e_i M {e_j}^t = 0$ for all i, j
- $f_i M {f_j}^t = 0$ for all i, j
- $e_i M {f_i}^t = 1$ for all i
- $e_i M {f_j}^t = 0$ for all i not equal j.

See the example for a pictorial description of such a basis.

EXAMPLES:

```
sage: E = matrix(QQ, 8, 8, [0, -1/2, -2, 1/2, 2, 0, -2, 1, 1/2, 0, -1, -3, 0, 2, 5/2, -3, 2,
[   0 -1/2   -2  1/2    2    0   -2    1]
[ 1/2    0   -1   -3    0    2  5/2   -3]
[   2    1    0  3/2   -1    0   -1   -2]
[-1/2    3 -3/2    0    1  3/2 -1/2 -1/2]
[  -2    0    1   -1    0    0    1   -1]
[   0   -2    0 -3/2    0    0  1/2   -2]
[   2 -5/2    1  1/2   -1 -1/2    0   -1]
[  -1    3    2  1/2    1    2    1    0]
sage: F, C = E.symplectic_form(); F
[ 0  0  0  0  1  0  0  0]
[ 0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  1  0]
[ 0  0  0  0  0  0  0  1]
[-1  0  0  0  0  0  0  0]
[ 0 -1  0  0  0  0  0  0]
[ 0  0 -1  0  0  0  0  0]
[ 0  0  0 -1  0  0  0  0]
sage: F == C * E * C.transpose()
True
```

tensor_product()

Returns the tensor product of two matrices.

EXAMPLES:

```
sage: M1=Matrix(QQ,[[-1,0],[-1/2,-1]])
sage: M2=Matrix(ZZ,[[1,-1,2],[-2,4,8]])
sage: M1.tensor_product(M2)
[  -1    1   -2|   0    0    0]
[   2   -4   -8|   0    0    0]
[--------------+--------------]
[-1/2  1/2   -1|  -1    1   -2]
[   1   -2   -4|   2   -4   -8]
sage: M2.tensor_product(M1)
[  -1    0|   1    0|  -2    0]
[-1/2   -1| 1/2    1|  -1   -2]
[---------+---------+---------]
```

```
[   2    0|  -4    0|  -8    0]
[   1    2|  -2   -4|  -4   -8]
```

trace()

Return the trace of self, which is the sum of the diagonal entries of self.

INPUT:

- `self` - a square matrix

OUTPUT: element of the base ring of self

EXAMPLES:

```
sage: a = matrix(3,range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: a.trace()
12
sage: a = matrix({(1,1):10, (2,1):-3, (2,2):4/3}); a
[  0   0   0]
[  0  10   0]
[  0  -3 4/3]
sage: a.trace()
34/3
```

visualize_structure()

Write a PNG image to 'filename' which visualizes self by putting black pixels in those positions which have nonzero entries.

White pixels are put at positions with zero entries. If 'maxsize' is given, then the maximal dimension in either x or y direction is set to 'maxsize' depending on which is bigger. If the image is scaled, the darkness of the pixel reflects how many of the represented entries are nonzero. So if e.g. one image pixel actually represents a 2x2 submatrix, the dot is darker the more of the four values are nonzero.

INPUT:

- `filename` - either a path or None in which case a filename in the current directory is chosen automatically (default:None)

maxsize - maximal dimension in either x or y direction of the resulting image. If None or a maxsize larger than max(self.nrows(),self.ncols()) is given the image will have the same pixelsize as the matrix dimensions (default: 512)

EXAMPLE:

```
sage: M = random_matrix(CC, 4)
sage: M.visualize_structure()
```

wiedemann()

Application of Wiedemann's algorithm to the i-th standard basis vector.

INPUT:

- `i` - an integer
- `t` - an integer (default: 0) if t is nonzero, use only the first t linear recurrence relations.

IMPLEMENTATION: This is a toy implementation.

EXAMPLES:

```
sage: t = matrix(QQ, 3, range(9)); t
[0 1 2]
[3 4 5]
[6 7 8]
sage: t.wiedemann(0)
```

```
x^2 - 12*x - 18
sage: t.charpoly()
x^3 - 12*x^2 - 18*x
```

cmp_pivots()
Compare two sequences of pivot columns.

- If x is shorter than y, return -1, i.e., x < y, "not as good".
- If x is longer than y, x > y, "better".
- If the length is the same then x is better, i.e., x > y if the entries of x are correspondingly >= those of y with one being greater.

decomp_seq()
This function is used internally be the decomposition matrix method. It takes a list of tuples and produces a sequence that is correctly sorted and prints with carriage returns.

EXAMPLES:

```
sage: from sage.matrix.matrix2 import decomp_seq
sage: V = [(QQ^3, 2), (QQ^2, 1)]
sage: decomp_seq(V)
[
(Vector space of dimension 2 over Rational Field, 1),
(Vector space of dimension 3 over Rational Field, 2)
]
```

32.8 Generic Asymptotically Fast Strassen Algorithms

Sage implements asymptotically fast echelon form and matrix multiplication algorithms.

class int_range()
Useful class for dealing with pivots in the strassen echelon, could have much more general application

AUTHORS:

- Robert Bradshaw

intervals()

to_list()

strassen_echelon()
Compute echelon form, in place. Internal function, call with M.echelonize(algorithm="strassen") Based on work of Robert Bradshaw and David Harvey at MSRI workshop in 2006.

INPUT:

- A - matrix window
- cutoff - size at which algorithm reverts to naive gaussian elemination and multiplication must be at least 1.

OUTPUT: The list of pivot columns

EXAMPLE:

```
sage: A = matrix(QQ, 7, [5, 0, 0, 0, 0, -1, 0, 0, 1, 0, 0, 0, 0, 0, -1, 3, 1, 0, -1, 0, 0, -1
sage: B = A.copy(); B._echelon_strassen(1); B
[ 1  0  0  0  0  0  0]
[ 0  1  0 -1  0  1  0]
[ 0  0  1  0  0  0  0]
[ 0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  1]
[ 0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0]
sage: C = A.copy(); C._echelon_strassen(2); C == B
True
sage: C = A.copy(); C._echelon_strassen(4); C == B
True


sage: n = 32; A = matrix(Integers(389),n,range(n^2))
sage: B = A.copy(); B._echelon_in_place_classical()
sage: C = A.copy(); C._echelon_strassen(2)
sage: B == C
True
```

TESTS:

```
sage: A = matrix(Integers(7), 4, 4, [1,2,0,3,0,0,1,0,0,1,0,0,0,0,0,1])
sage: B = A.copy(); B._echelon_in_place_classical()
sage: C = A.copy(); C._echelon_strassen(2)
sage: B == C
True


sage: A = matrix(Integers(7), 4, 4, [1,0,5,0,2,0,3,6,5,1,2,6,4,6,1,1])
sage: B = A.copy(); B._echelon_in_place_classical()
sage: C = A.copy(); C._echelon_strassen(2)
sage: B == C
True
```

AUTHORS:

- Robert Bradshaw

strassen_window_multiply()

Multiplies the submatrices specified by A and B, places result in C. Assumes that A and B have compatible dimensions to be multiplied, and that C is the correct size to receive the product, and that they are all defined over the same ring.

Uses strassen multiplication at high levels and then uses MatrixWindow methods at low levels. EXAMPLES: The following matrix dimensions are chosen especially to exercise the eight possible parity combinations that ocould ccur while subdividing the matrix in the strassen recursion. The base case in both cases will be a (4x5) matrix times a (5x6) matrix.

```
sage: A = MatrixSpace(Integers(2^65), 64, 83).random_element()
sage: B = MatrixSpace(Integers(2^65), 83, 101).random_element()
sage: A._multiply_classical(B) == A._multiply_strassen(B, 3)
True
```

AUTHORS:

- David Harvey

32.9 Minimal Polynomials of Linear Recurrence Sequences

AUTHORS:

- William Stein

berlekamp_massey(a)

Use the Berlekamp-Massey algorithm to find the minimal polynomial of a linearly recurrence sequence a.

The minimal polynomial of a linear recurrence $\{a_r\}$ is by definition the unique monic polynomial g, such that if $\{a_r\}$ satisfies a linear recurrence $a_{j+k} + b_{j-1}a_{j-1+k} + \cdots + b_0 a_k = 0$ (for all $k \geq 0$), then g divides the polynomial $x^j + \sum_{i=0}^{j-1} b_i x^i$.

INPUT:

- `a` - a list of even length of elements of a field (or domain)

OUTPUT:

- `Polynomial` - the minimal polynomial of the sequence (as a polynomial over the field in which the entries of a live)

EXAMPLES:

```
sage: berlekamp_massey([1,2,1,2,1,2])
x^2 - 1
sage: berlekamp_massey([GF(7)(1),19,1,19])
x^2 + 6
sage: berlekamp_massey([2,2,1,2,1,191,393,132])
x^4 - 36727/11711*x^3 + 34213/5019*x^2 + 7024942/35133*x - 335813/1673
sage: berlekamp_massey(prime_range(2,38))
x^6 - 14/9*x^5 - 7/9*x^4 + 157/54*x^3 - 25/27*x^2 - 73/18*x + 37/9
```

32.10 Base class for dense matrices

TESTS:

```
sage: R.<a,b> = QQ[]
sage: m = matrix(R,2,[0,a,b,b^2])
sage: loads(dumps(m)) == m
True
```

class **Matrix_dense**()

antitranspose()

Returns the anittranspose of self, without changing self.

EXAMPLES:

```
sage: A = matrix(2,3,range(6)); A
[0 1 2]
[3 4 5]
sage: A.antitranspose()
[5 2]
[4 1]
[3 0]
```

```
sage: A.subdivide(1,2); A
[0 1|2]
[---+-]
[3 4|5]
sage: A.antitranspose()
[5|2]
[-+-]
[4|1]
[3|0]
```

apply_map()

Apply the given map phi (an arbitrary Python function or callable object) to this dense matrix. If R is not given, automatically determine the base ring of the resulting matrix.

INPUT: sparse – True to make the output a sparse matrix; default False

- phi - arbitrary Python function or callable object
- R - (optional) ring

OUTPUT: a matrix over R

EXAMPLES:

```
sage: m = matrix(ZZ, 3, range(9))
sage: k.<a> = GF(9)
sage: f = lambda x: k(x)
sage: n = m.apply_map(f); n
[0 1 2]
[0 1 2]
[0 1 2]
sage: n.parent()
Full MatrixSpace of 3 by 3 dense matrices over Finite Field in a of size 3^2
```

In this example, we explicitly specify the codomain.

```
sage: s = GF(3)
sage: f = lambda x: s(x)
sage: n = m.apply_map(f, k); n
[0 1 2]
[0 1 2]
[0 1 2]
sage: n.parent()
Full MatrixSpace of 3 by 3 dense matrices over Finite Field in a of size 3^2
```

If self is subdivided, the result will be as well:

```
sage: m = matrix(2, 2, srange(4))
sage: m.subdivide(None, 1); m
[0|1]
[2|3]
sage: m.apply_map(lambda x: x*x)
[0|1]
[4|9]
```

If the map sends most of the matrix to zero, then it may be useful to get the result as a sparse matrix.

```
sage: m = matrix(ZZ, 3, 3, range(1, 10))
sage: n = m.apply_map(lambda x: 1//x, sparse=True); n
[1 0 0]
[0 0 0]
[0 0 0]
sage: n.parent()
Full MatrixSpace of 3 by 3 sparse matrices over Integer Ring
```

TESTS:

```
sage: m = matrix([])
sage: m.apply_map(lambda x: x*x) == m
True

sage: m.apply_map(lambda x: x*x, sparse=True).parent()
Full MatrixSpace of 0 by 0 sparse matrices over Integer Ring
```

apply_morphism()

Apply the morphism phi to the coefficients of this dense matrix.

The resulting matrix is over the codomain of phi.

INPUT:

- `phi` - a morphism, so phi is callable and phi.domain() and phi.codomain() are defined. The codomain must be a ring.

OUTPUT: a matrix over the codomain of phi

EXAMPLES:

```
sage: m = matrix(ZZ, 3, range(9))
sage: phi = ZZ.hom(GF(5))
sage: m.apply_morphism(phi)
[0 1 2]
[3 4 0]
[1 2 3]
sage: parent(m.apply_morphism(phi))
Full MatrixSpace of 3 by 3 dense matrices over Finite Field of size 5
```

We apply a morphism to a matrix over a polynomial ring:

```
sage: R.<x,y> = QQ[]
sage: m = matrix(2, [x,x^2 + y, 2/3*y^2-x, x]); m
[          x     x^2 + y]
[2/3*y^2 - x           x]
sage: phi = R.hom([y,x])
sage: m.apply_morphism(phi)
[          y     y^2 + x]
[2/3*x^2 - y           y]
```

transpose()

Returns the transpose of self, without changing self.

EXAMPLES: We create a matrix, compute its transpose, and note that the original matrix is not changed.

```
sage: M = MatrixSpace(QQ,  2)
sage: A = M([1,2,3,4])
sage: B = A.transpose()
sage: print B
[1 3]
[2 4]
sage: print A
[1 2]
[3 4]

sage: A.subdivide(None, 1); A
[1|2]
[3|4]
sage: A.transpose()
[1 3]
[---]
[2 4]
```

32.11 Base class for sparse matrices

class Matrix_sparse()

antitranspose()

apply_map()

Apply the given map phi (an arbitrary Python function or callable object) to this matrix. If R is not given, automatically determine the base ring of the resulting matrix.

INPUT: sparse – False to make the output a dense matrix; default True

- phi - arbitrary Python function or callable object
- R - (optional) ring

OUTPUT: a matrix over R

EXAMPLES:

```
sage: m = matrix(ZZ, 10000, {(1,2): 17}, sparse=True)
sage: k.<a> = GF(9)
sage: f = lambda x: k(x)
sage: n = m.apply_map(f)
sage: n.parent()
Full MatrixSpace of 10000 by 10000 sparse matrices over Finite Field in a of size 3^2
sage: n[1,2]
2
```

An example where the codomain is explicitly specified.

```
sage: n = m.apply_map(lambda x:x%3, GF(3))
sage: n.parent()
Full MatrixSpace of 10000 by 10000 sparse matrices over Finite Field of size 3
sage: n[1,2]
2
```

If we didn't specify the codomain, the resulting matrix in the above case ends up over ZZ again:

```
sage: n = m.apply_map(lambda x:x%3)
sage: n.parent()
Full MatrixSpace of 10000 by 10000 sparse matrices over Integer Ring
sage: n[1,2]
2
```

If self is subdivided, the result will be as well:

```
sage: m = matrix(2, 2, [0, 0, 3, 0])
sage: m.subdivide(None, 1); m
[0|0]
[3|0]
sage: m.apply_map(lambda x: x*x)
[0|0]
[9|0]
```

If the map sends zero to a non-zero value, then it may be useful to get the result as a dense matrix.

```
sage: m = matrix(ZZ, 3, 3, [0] * 7 + [1,2], sparse=True); m
[0 0 0]
[0 0 0]
[0 1 2]
sage: parent(m)
Full MatrixSpace of 3 by 3 sparse matrices over Integer Ring
sage: n = m.apply_map(lambda x: x+polygen(QQ), sparse=False); n
```

```
[    x     x     x]
[    x     x     x]
[    x x + 1 x + 2]
sage: parent(n)
Full MatrixSpace of 3 by 3 dense matrices over Univariate Polynomial Ring in x over Rational
```

TESTS:

```
sage: m = matrix([], sparse=True)
sage: m.apply_map(lambda x: x*x) == m
True

sage: m.apply_map(lambda x: x*x, sparse=False).parent()
Full MatrixSpace of 0 by 0 dense matrices over Integer Ring
```

Check that we don't unnecessarily apply phi to 0 in the sparse case:

```
sage: m = matrix(QQ, 2, 2, range(1, 5), sparse=True)
sage: m.apply_map(lambda x: 1/x)
[  1 1/2]
[1/3 1/4]
```

Test subdivisions when phi maps 0 to non-zero:

```
sage: m = matrix(2, 2, [0, 0, 3, 0])
sage: m.subdivide(None, 1); m
[0|0]
[3|0]
sage: m.apply_map(lambda x: x+1)
[1|1]
[4|1]
```

apply_morphism()

Apply the morphism phi to the coefficients of this sparse matrix.

The resulting matrix is over the codomain of phi.

INPUT:

- `phi` - a morphism, so phi is callable and phi.domain() and phi.codomain() are defined. The codomain must be a ring.

OUTPUT: a matrix over the codomain of phi

EXAMPLES:

```
sage: m = matrix(ZZ, 3, range(9), sparse=True)
sage: phi = ZZ.hom(GF(5))
sage: m.apply_morphism(phi)
[0 1 2]
[3 4 0]
[1 2 3]
sage: m.apply_morphism(phi).parent()
Full MatrixSpace of 3 by 3 sparse matrices over Finite Field of size 5
```

change_ring()

Return the matrix obtained by coercing the entries of this matrix into the given ring.

Always returns a copy (unless self is immutable, in which case returns self).

EXAMPLES:

sage: A = matrix(QQ['x,y'], 2, [0,-1,2*x,-2], sparse=True); A [0 -1] [2*x -2] sage: A.change_ring(QQ['x,y,z']) [0 -1] [2*x -2]

Subdivisions are preserved when changing rings:

```
sage: A.subdivide([2],[]); A
[  0  -1]
[2*x  -2]
[-------]
sage: A.change_ring(RR['x,y'])
[                  0  -1.00000000000000]
[2.00000000000000*x  -2.00000000000000]
[-----------------------------------]
```

charpoly()

Return the characteristic polynomial of this matrix.

Note - the generic sparse charpoly implementation in Sage is to just compute the charpoly of the corresponding dense matrix, so this could use a lot of memory. In particular, for this matrix, the charpoly will be computed using a dense algorithm.

EXAMPLES:

```
sage: A = matrix(ZZ, 4, range(16), sparse=True)
sage: A.charpoly()
x^4 - 30*x^3 - 80*x^2
sage: A.charpoly('y')
y^4 - 30*y^3 - 80*y^2
sage: A.charpoly()
x^4 - 30*x^3 - 80*x^2
```

transpose()

Returns the transpose of self, without changing self.

EXAMPLES: We create a matrix, compute its transpose, and note that the original matrix is not changed.

```
sage: M = MatrixSpace(QQ,  2, sparse=True)
sage: A = M([1,2,3,4])
sage: B = A.transpose()
sage: print B
[1 3]
[2 4]
sage: print A
[1 2]
[3 4]
```

32.12 Dense Matrices over a general ring

class **Matrix_generic_dense**()

The Matrix_generic_dense class derives from Matrix, and defines functionality for dense matrices over any base ring. Matrices are represented by a list of elements in the base ring, and element access operations are implemented in this class.

EXAMPLES:

```
sage: A = random_matrix(Integers(25)['x'],2); A
[    x^2 + 12*x + 2   4*x^2 + 13*x + 8]
[ 22*x^2 + 2*x + 17 19*x^2 + 22*x + 14]
sage: type(A)
<type 'sage.matrix.matrix_generic_dense.Matrix_generic_dense'>
sage: A == loads(dumps(A))
True
```

32.13 Sparse Matrices over a general ring

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: M = MatrixSpace(QQ['x'],2,3,sparse=True); M
Full MatrixSpace of 2 by 3 sparse matrices over Univariate Polynomial Ring in x over Rational Field
sage: a = M(range(6)); a
[0 1 2]
[3 4 5]
sage: b = M([x^n for n in range(6)]); b
[  1   x x^2]
[x^3 x^4 x^5]
sage: a * b.transpose()
[            2*x^2 + x           2*x^5 + x^4]
[      5*x^2 + 4*x + 3 5*x^5 + 4*x^4 + 3*x^3]
sage: pari(a)*pari(b.transpose())
[2*x^2 + x, 2*x^5 + x^4; 5*x^2 + 4*x + 3, 5*x^5 + 4*x^4 + 3*x^3]
sage: c = copy(b); c
[  1   x x^2]
[x^3 x^4 x^5]
sage: c[0,0] = 5; c
[  5   x x^2]
[x^3 x^4 x^5]
sage: b[0,0]
1
sage: c.dict()
{(0, 1): x, (1, 2): x^5, (0, 0): 5, (1, 0): x^3, (0, 2): x^2, (1, 1): x^4}
sage: c.list()
[5, x, x^2, x^3, x^4, x^5]
sage: c.rows()
[(5, x, x^2), (x^3, x^4, x^5)]
sage: loads(dumps(c)) == c
True
sage: d = c.change_ring(CC['x']); d
[    5.00000000000000   1.00000000000000*x 1.00000000000000*x^2]
[1.00000000000000*x^3 1.00000000000000*x^4 1.00000000000000*x^5]
sage: latex(c)
\left(\begin{array}{rrr}
5 & x & x^{2} \\
x^{3} & x^{4} & x^{5}
\end{array}\right)
sage: c.sparse_rows()
[(5, x, x^2), (x^3, x^4, x^5)]
sage: d = c.dense_matrix(); d
[  5   x x^2]
[x^3 x^4 x^5]
sage: parent(d)
Full MatrixSpace of 2 by 3 dense matrices over Univariate Polynomial Ring in x over Rational Field
sage: c.sparse_matrix() is c
True
sage: c.is_sparse()
True
```

class Matrix_generic_sparse()

The Matrix_generic_sparse class derives from Matrix, and defines functionality for sparse matrices over any base ring. A generic sparse matrix is represented using a dictionary with keys pairs (i, j) and values in

the base ring.

The values of the dictionary must never be zero.

Matrix_sparse_from_rows()

INPUT:

- X - nonempty list of SparseVector rows

OUTPUT: Sparse_matrix with those rows.

EXAMPLES:

```
sage: V = VectorSpace(QQ,20,sparse=True)
sage: v = V(0)
sage: v[9] = 4
sage: from sage.matrix.matrix_generic_sparse import Matrix_sparse_from_rows
sage: Matrix_sparse_from_rows([v])
[0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0]
sage: Matrix_sparse_from_rows([v, v, v, V(0)])
[0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

32.14 Dense matrices over $\mathbb{Z}/n\mathbb{Z}$ for n small.

AUTHORS:

- William Stein
- Robert Bradshaw

This is a compiled implementation of dense matrices over $\mathbf{Z}/n\mathbf{Z}$ for n small.

EXAMPLES:

```
sage: a = matrix(Integers(37),3,range(9),sparse=False); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: a.rank()
2
sage: type(a)
<type 'sage.matrix.matrix_modn_dense.Matrix_modn_dense'>
sage: a[0,0] = 5
sage: a.rank()
3
sage: parent(a)
Full MatrixSpace of 3 by 3 dense matrices over Ring of integers modulo 37


sage: a^2
[ 3 23 31]
[20 17 29]
[25 16  0]
sage: a+a
[10  2  4]
```

```
[ 6  8 10]
[12 14 16]


sage: b = a.new_matrix(2,3,range(6)); b
[0 1 2]
[3 4 5]
sage: a*b
...
TypeError: unsupported operand parent(s) for '*': 'Full MatrixSpace of 3 by 3 dense matrices over Ri
sage: b*a
[15 18 21]
[20 17 29]


sage: a == loads(dumps(a))
True
sage: b == loads(dumps(b))
True


sage: a.echelonize(); a
[1 0 0]
[0 1 0]
[0 0 1]
sage: b.echelonize(); b
[ 1  0 36]
[ 0  1  2]
```

We create a matrix group and coerce it to GAP:

```
sage: M = MatrixSpace(GF(3),3,3)
sage: G = MatrixGroup([M([[0,1,0],[0,0,1],[1,0,0]]), M([[0,1,0],[1,0,0],[0,0,1]])])
sage: G
Matrix group over Finite Field of size 3 with 2 generators:
 [[[0, 1, 0], [0, 0, 1], [1, 0, 0]], [[0, 1, 0], [1, 0, 0], [0, 0, 1]]]
sage: gap(G)
Group(
[ [ [ 0*Z(3), Z(3)^0, 0*Z(3) ], [ 0*Z(3), 0*Z(3), Z(3)^0 ], [ Z(3)^0, 0*Z(3),
          0*Z(3) ] ],
  [ [ 0*Z(3), Z(3)^0, 0*Z(3) ], [ Z(3)^0, 0*Z(3), 0*Z(3) ],
      [ 0*Z(3), 0*Z(3), Z(3)^0 ] ] ])
```

TESTS:

```
sage: M = MatrixSpace(GF(5),2,2)
sage: A = M([1,0,0,1])
sage: A - int(-1)
[2 0]
[0 2]
sage: B = M([4,0,0,1])
sage: B - int(-1)
[0 0]
[0 2]
sage: Matrix(GF(5),0,0, sparse=False).inverse()
[]
```

class **Matrix_modn_dense**()

charpoly()

Returns the characteristic polynomial of self.

INPUT:

- `var` - a variable name
- `algorithm` - 'generic' 'linbox' (default)

EXAMPLES:

```
sage: A = Mat(GF(7),3,3)(range(3)*3)
sage: A.charpoly()
x^3 + 4*x^2

sage: A = Mat(Integers(6),3,3)(range(9))
sage: A.charpoly()
x^3
```

ALGORITHM: Uses LinBox if self.base_ring() is a field, otherwise use Hessenberg form algorithm.

determinant()

Return the determinant of this matrix.

EXAMPLES:

```
sage: m = matrix(GF(101),5,range(25))
sage: m.det()
0

sage: m = matrix(Integers(4), 2, [2,2,2,2])
sage: m.det()
0
```

TESTS:

```
sage: m = random_matrix(GF(3), 3, 4)
sage: m.determinant()
...
ArithmeticError: self must be a square matrix
```

echelonize()

Puts self in row echelon form.

INPUT:

- `self` - a mutable matrix
- `algorithm` - 'linbox' - uses the C++ linbox library
- `'gauss'` - uses a custom slower $O(n^3)$ Gauss elimination implemented in Sage.
- `'all'` - compute using both algorithms and verify that the results are the same (for the paranoid).
- `**kwds` - these are all ignored

OUTPUT:

- self is put in reduced row echelon form.
- the rank of self is computed and cached
- the pivot columns of self are computed and cached.
- the fact that self is now in echelon form is recorded and cached so future calls to echelonize return immediately.

EXAMPLES:

```
sage: a = matrix(GF(97),3,4,range(12))
sage: a.echelonize(); a
[ 1  0 96 95]
[ 0  1  2  3]
[ 0  0  0  0]
sage: a.pivots()
[0, 1]
```

hessenbergize()
Transforms self in place to its Hessenberg form.

lift()
Return the lift of this matrix to the integers.
EXAMPLES:

```
sage: a = matrix(GF(7),2,3,[1..6])
sage: a.lift()
[1 2 3]
[4 5 6]
sage: a.lift().parent()
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
```

Subdivisions are preserved when lifting:

```
sage: a.subdivide([], [1,1]); a
[1||2 3]
[4||5 6]
sage: a.lift()
[1||2 3]
[4||5 6]
```

list()
Return list of elements of self.
EXAMPLES:

```
sage: w = matrix(GF(19), 2, 3, [1..6])
sage: w.list()
[1, 2, 3, 4, 5, 6]
sage: w.list()[0].parent()
Finite Field of size 19
```

TESTS:

```
sage: w = random_matrix(GF(3),100)
sage: w.parent()(w.list()) == w
True
```

matrix_window()
Return the requested matrix window.
EXAMPLES:

```
sage: a = matrix(GF(7),3,range(9)); a
[0 1 2]
[3 4 5]
[6 0 1]
sage: type(a)
<type 'sage.matrix.matrix_modn_dense.Matrix_modn_dense'>
```

We test the optional check flag.

```
sage: matrix(GF(7),[1]).matrix_window(0,1,1,1)
...
IndexError: matrix window index out of range
sage: matrix(GF(7),[1]).matrix_window(0,1,1,1, check=False)
Matrix window of size 1 x 1 at (0,1):
[1]
```

minpoly()

Returns the minimal polynomial of self.

INPUT:

- var - a variable name
- algorithm - 'generic' 'linbox' (default)

randomize()

Randomize density proportion of the entries of this matrix, leaving the rest unchanged.

EXAMPLES:

```
sage: A = matrix(GF(5), 5, 5, 0)
sage: A.randomize(0.5); A
[0 0 0 2 0]
[0 3 0 0 2]
[4 0 0 0 0]
[4 0 0 0 0]
[0 1 0 0 0]
sage: A.randomize(); A
[3 3 2 1 2]
[4 3 3 2 2]
[0 3 3 3 3]
[3 3 2 2 4]
[2 2 2 1 4]
```

rank()

Return the rank of this matrix.

EXAMPLES:

```
sage: m = matrix(GF(7),5,range(25))
sage: m.rank()
2
```

Rank is not implemented over the integers modulo a composite yet.

```
sage: m = matrix(Integers(4), 2, [2,2,2,2])
sage: m.rank()
...
NotImplementedError: Echelon form not implemented over 'Ring of integers modulo 4'.
```

32.15 Sparse matrices over $\mathbb{Z}/n\mathbb{Z}$ for n small.

This is a compiled implementation of sparse matrices over $\mathbf{Z}/n\mathbf{Z}$ for n small.

TODO: - move vectors into a pyrex vector class - add _add_ and _mul_ methods.

EXAMPLES:

```
sage: a = matrix(Integers(37),3,3,range(9),sparse=True); a
[0 1 2]
[3 4 5]
```

```
[6 7 8]
sage: type(a)
<type 'sage.matrix.matrix_modn_sparse.Matrix_modn_sparse'>
sage: parent(a)
Full MatrixSpace of 3 by 3 sparse matrices over Ring of integers modulo 37
sage: a^2
[15 18 21]
[ 5 17 29]
[32 16  0]
sage: a+a
[ 0  2  4]
[ 6  8 10]
[12 14 16]
sage: b = a.new_matrix(2,3,range(6)); b
[0 1 2]
[3 4 5]
sage: a*b
...
TypeError: unsupported operand parent(s) for '*': 'Full MatrixSpace of 3 by 3 sparse matrices over Ri
sage: b*a
[15 18 21]
[ 5 17 29]


sage: a == loads(dumps(a))
True
sage: b == loads(dumps(b))
True


sage: a.echelonize(); a
[ 1  0 36]
[ 0  1  2]
[ 0  0  0]
sage: b.echelonize(); b
[ 1  0 36]
[ 0  1  2]
sage: a.pivots()
[0, 1]
sage: b.pivots()
[0, 1]
sage: a.rank()
2
sage: b.rank()
2
sage: a[2,2] = 5
sage: a.rank()
3
```

TESTS: sage: matrix(Integers(37),0,0,sparse=True).inverse() []

class Matrix_modn_sparse()

density()

Return the density of self, i.e., the ratio of the number of nonzero entries of self to the total size of self.

EXAMPLES:

```
sage: A = matrix(QQ,3,3,[0,1,2,3,0,0,6,7,8],sparse=True)
sage: A.density()
2/3
```

Notice that the density parameter does not ensure the density of a matrix; it is only an upper bound.

```
sage: A = random_matrix(GF(127),200,200,density=0.3, sparse=True)
sage: A.density()
257/1000
```

lift()

Return lift of this matrix to a sparse matrix over the integers.

EXAMPLES: sage: a = matrix(GF(7),2,3,[1..6], sparse=True) sage: a.lift() [1 2 3] [4 5 6] sage: a.lift().parent() Full MatrixSpace of 2 by 3 sparse matrices over Integer Ring

Subdivisions are preserved when lifting:

```
sage: a.subdivide([], [1,1]); a
[1|2 3]
[4|5 6]
sage: a.lift()
[1|2 3]
[4|5 6]
```

matrix_from_columns()

Return the matrix constructed from self using columns with indices in the columns list.

EXAMPLES:

```
sage: M = MatrixSpace(GF(127),3,3,sparse=True)
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.matrix_from_columns([2,1])
[2 1]
[5 4]
[8 7]
```

matrix_from_rows()

Return the matrix constructed from self using rows with indices in the rows list.

INPUT:

- rows - list or tuple of row indices

EXAMPLE:

```
sage: M = MatrixSpace(GF(127),3,3,sparse=True)
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.matrix_from_rows([2,1])
[6 7 8]
[3 4 5]
```

p

rank()

Compute the rank of self.

INPUT:

- `gauss` - if True LinBox' Gaussian elimination is used. If False 'Symbolic Reordering' as implemented in LinBox is used. If 'native' the native Sage implementation is used. (default: False)

EXAMPLE:

```
sage: A = random_matrix(GF(127),200,200,density=0.01,sparse=True)
sage: r1 = A.rank(gauss=False)
sage: r2 = A.rank(gauss=True)
sage: r3 = A.rank(gauss='native')
sage: r1 == r2 == r3
True
sage: r1
155
```

ALGORITHM: Uses LinBox or native implementation.

REFERENCES:

- Jean-Guillaume Dumas and Gilles Villars. 'Computing the Rank of Large Sparse Matrices over Finite Fields'. Proc. CASC'2002, The Fifth International Workshop on Computer Algebra in Scientific Computing, Big Yalta, Crimea, Ukraine, 22-27 sept. 2002, Springer-Verlag, http://perso.ens-lyon.fr/gilles.villard/BIBLIOGRAPHIE/POSTSCRIPT/rankjgd.ps

Note: For very sparse matrices Gaussian elimination is faster because it barly has anything to do. If the fill in needs to be considered, 'Symbolic Reordering' is usually much faster.

swap_rows()

transpose()

Return the transpose of self.

EXAMPLE:

```
sage: A = matrix(GF(127),3,3,[0,1,0,2,0,0,3,0,0],sparse=True)
sage: A
[0 1 0]
[2 0 0]
[3 0 0]
sage: A.transpose()
[0 2 3]
[1 0 0]
[0 0 0]
```

visualize_structure()

Write a PNG image to 'filename' which visualizes self by putting black pixels in those positions which have nonzero entries.

White pixels are put at positions with zero entries. If 'maxsize' is given, then the maximal dimension in either x or y direction is set to 'maxsize' depending on which is bigger. If the image is scaled, the darkness of the pixel reflects how many of the represented entries are nonzero. So if e.g. one image pixel actually represents a 2x2 submatrix, the dot is darker the more of the four values are nonzero.

INPUT:

- `filename` - either a path or None in which case a filename in the current directory is chosen automatically (default:None)
- `maxsize` - maximal dimension in either x or y direction of the resulting image. If None or a maxsize larger than max(self.nrows(),self.ncols()) is given the image will have the same pixelsize as the matrix dimensions (default: 512)

32.16 Dense matrices over the integer ring.

AUTHORS:

- William Stein
- Robert Bradshaw

EXAMPLES:

```
sage: a = matrix(ZZ, 3,3, range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: a.det()
0
sage: a[0,0] = 10; a.det()
-30
sage: a.charpoly()
x^3 - 22*x^2 + 102*x + 30
sage: b = -3*a
sage: a == b
False
sage: b < a
True
```

TESTS:

```
sage: a = matrix(ZZ,2,range(4), sparse=False)
sage: loads(dumps(a)) == a
True
sage: Matrix(ZZ,0,0).inverse()
[]
```

class Matrix_integer_dense()

Matrix over the integers.

On a 32-bit machine, they can have at most $2^{32} - 1$ rows or columns. On a 64-bit machine, matrices can have at most $2^{64} - 1$ rows or columns.

EXAMPLES:

```
sage: a = MatrixSpace(ZZ,3)(2); a
[2 0 0]
[0 2 0]
[0 0 2]
sage: a = matrix(ZZ,1,3, [1,2,-3]); a
[ 1  2 -3]
sage: a = MatrixSpace(ZZ,2,4)(2); a
...
TypeError: nonzero scalar matrix must be square
```

BKZ()

Block Korkin-Zolotarev reduction.

INPUT:

- `fp` - 'fp' - double precision: NTL's FP or fpLLL's double
- `'qd'` - quad doubles: NTL's QP
- `'qd1'` - quad doubles: uses quad_float precision to compute Gram-Schmidt, but uses double precision in the search phase of the block reduction algorithm. This seems adequate for most purposes, and is faster than 'qd', which uses quad_float precision uniformly throughout.
- `'xd'` - extended exponent: NTL's XD

- 'rr' - arbitrary precision (default)
- block_size - specifies the size of the blocks in the reduction. High values yield shorter vectors, but the running time increases exponentially with block_size. block_size should be between 2 and the number of rows of self (default: 10)
- prune - The optional parameter prune can be set to any positive number to invoke the Volume Heuristic from [Schnorr and Horner, Eurocrypt '95]. This can significantly reduce the running time, and hence allow much bigger block size, but the quality of the reduction is of course not as good in general. Higher values of prune mean better quality, and slower running time. When prune == 0, pruning is disabled. Recommended usage: for block_size = 30, set 10 = prune = 15.
- use_givens - use Given's orthogonalization. This is a bit slower, but generally much more stable, and is really the preferred orthogonalization strategy. For a nice description of this, see Chapter 5 of [G. Golub and C. van Loan, Matrix Computations, 3rd edition, Johns Hopkins Univ. Press, 1996].

EXAMPLE:

```
sage: A = Matrix(ZZ,3,3,range(1,10))
sage: A.BKZ()
[ 0  0  0]
[ 2  1  0]
[-1  1  3]
sage: A = Matrix(ZZ,3,3,range(1,10))
sage: A.BKZ(use_givens=True)
[ 0  0  0]
[ 2  1  0]
[-1  1  3]

sage: A = Matrix(ZZ,3,3,range(1,10))
sage: A.BKZ(fp="fp")
[ 0  0  0]
[ 2  1  0]
[-1  1  3]
```

LLL()

Returns LLL reduced or approximated LLL reduced lattice R for this matrix interpreted as a lattice.

A lattice $(b_1, b_2, ..., b_d)$ is (δ, η) -LLL-reduced if the two following conditions hold:

- For any $i > j$, we have $|mu_{i,j}| <= \eta$,
- For any $i < d$, we have $\delta |b_i^*|^2 <= |b_{i+1}^* + mu_{i+1,i} b_i^*|^2$,

where $mu_{i,j} =< b_i, b_j^* > / < b_j^*, b_j^* >$ and b_i^* is the i-th vector of the Gram-Schmidt orthogonalisation of $(b_1, b_2, ..., b_d)$.

The default reduction parameters are $\delta = 3/4$ and $eta = 0.501$. The parameters δ and η must satisfy: $0.25 < \delta <= 1.0$ and $0.5 <= \eta < sqrt(\delta)$. Polynomial time complexity is only guaranteed for $\delta < 1$.

The lattice is returned as a matrix. Also the rank (and the determinant) of self are cached if those are computed during the reduction. Note that in general this only happens when self.rank() == self.ncols() and the exact algorithm is used.

INPUT:

- delta - parameter as described above (default: 3/4)
- eta - parameter as described above (default: 0.501), ignored by NTL
- algorithm - string (default: "fpLLL:wrapper") one of the algorithms mentioned below
- fp
  - None - NTL's exact reduction or fpLLL's wrapper
  - 'fp' - double precision: NTL's FP or fpLLL's double
  - 'qd' - quad doubles: NTL's QP
  - 'xd' - extended exponent: NTL's XD or fpLLL's dpe

–'rr' - arbitrary precision: NTL'RR or fpLLL's MPFR

- prec - precision, ignored by NTL (default: auto choose)
- early_red - perform early reduction, ignored by NTL (default: False)
- use_givens - use Givens orthogonalization (default: False) only applicable to approximate reductions and NTL. This is more stable but slower.
  Also, if the verbose level is = 2, some more verbose output is printed during the calculation if NTL is used.
  AVAILABLE ALGORITHMS:
- NTL:LLL - NTL's LLL + fp
- fpLLL:heuristic - fpLLL's heuristic + fp
- fpLLL:fast - fpLLL's fast
- fpLLL:wrapper - fpLLL's automatic choice (default)

OUTPUT: a matrix over the integers

EXAMPLE:

```
sage: A = Matrix(ZZ,3,3,range(1,10))
sage: A.LLL()
[ 0  0  0]
[ 2  1  0]
[-1  1  3]
```

We compute the extended GCD of a list of integers using LLL, this example is from the Magma handbook:

```
sage: Q = [ 67015143, 248934363018, 109210, 25590011055, 74631449, \
            10230248, 709487, 68965012139, 972065, 864972271 ]
sage: n = len(Q)
sage: S = 100
sage: X = Matrix(ZZ, n, n + 1)
sage: for i in xrange(n):
...       X[i,i + 1] = 1
sage: for i in xrange(n):
...       X[i,0] = S*Q[i]
sage: L = X.LLL()
sage: M = L.row(n-1).list()[1:]
sage: M
[-3, -1, 13, -1, -4, 2, 3, 4, 5, -1]
sage: add([Q[i]*M[i] for i in range(n)])
-1
```

ALGORITHM: Uses the NTL library by Victor Shoup or fpLLL library by Damien Stehle depending on the chosen algorithm.

REFERENCES:

- ntl.mat_ZZ or sage.libs.fplll.fplll for details on the used algorithms.

LLL_gram()

LLL reduction of the lattice whose gram matrix is self.

INPUT:

- M - gram matrix of a definite quadratic form

OUTPUT:

- U - unimodular transformation matrix such that U.transpose() * M * U is LLL-reduced.

ALGORITHM: Use PARI

EXAMPLES:

```
sage: M = Matrix(ZZ, 2, 2, [5,3,3,2]) ; M
[5 3]
[3 2]
sage: U = M.LLL_gram(); U
[-1  1]
[ 1 -2]
sage: U.transpose() * M * U
[1 0]
[0 1]
```

Semidefinite and indefinite forms raise a ValueError:

```
sage: Matrix(ZZ,2,2,[2,6,6,3]).LLL_gram()
...
ValueError: not a definite matrix
sage: Matrix(ZZ,2,2,[1,0,0,-1]).LLL_gram()
...
ValueError: not a definite matrix
```

BUGS: should work for semidefinite forms (PARI is ok)

antitranspose()

Returns the antitranspose of self, without changing self.

EXAMPLES:

```
sage: A = matrix(2,3,range(6))
sage: type(A)
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
sage: A.antitranspose()
[5 2]
[4 1]
[3 0]
sage: A
[0 1 2]
[3 4 5]

sage: A.subdivide(1,2); A
[0 1|2]
[---+-]
[3 4|5]
sage: A.antitranspose()
[5|2]
[-+-]
[4|1]
[3|0]
```

charpoly()

INPUT:

- `var` - a variable name
- `algorithm` - 'linbox' (default) 'generic'

Note: Linbox charpoly disabled on 64-bit machines, since it hangs in many cases.

EXAMPLES:

```
sage: A = matrix(ZZ,6, range(36))
sage: f = A.charpoly(); f
x^6 - 105*x^5 - 630*x^4
sage: f(A) == 0
True
sage: n=20; A = Mat(ZZ,n)(range(n^2))
```

```
sage: A.charpoly()
x^20 - 3990*x^19 - 266000*x^18
sage: A.minpoly()
x^3 - 3990*x^2 - 266000*x
```

decomposition()

Returns the decomposition of the free module on which this matrix A acts from the right (i.e., the action is x goes to x A), along with whether this matrix acts irreducibly on each factor. The factors are guaranteed to be sorted in the same way as the corresponding factors of the characteristic polynomial, and are saturated as ZZ modules.

INPUT:

- `self` - a matrix over the integers
- `**kwds` - these are passed onto to the decomposition over QQ command.

EXAMPLES:

```
sage: t = ModularSymbols(11,sign=1).hecke_matrix(2)
sage: w = t.change_ring(ZZ)
sage: w.list()
[3, -1, 0, -2]
```

determinant()

Return the determinant of this matrix.

INPUT:

- `algorithm`
  - **`'default'` – use 'pari' when number of rows less than 50;** otherwise, use 'padic'
  - `'padic'` - uses a p-adic / multimodular algorithm that relies on code in IML and linbox
  - `'linbox'` - calls linbox det (you *must* set proof=False to use this!)
  - `'ntl'` - calls NTL's det function
  - `'pari'` - uses PARI
- `proof` - bool or None; if None use proof.linear_algebra(); only relevant for the padic algorithm.
  Note: It would be *VERY VERY* hard for det to fail even with proof=False.
- `stabilize` - if proof is False, require det to be the same for this many CRT primes in a row. Ignored if proof is True.

ALGORITHM: The p-adic algorithm works by first finding a random vector v, then solving A*x = v and taking the denominator d. This gives a divisor of the determinant. Then we compute $\det(A)/d$ using a multimodular algorithm and the Hadamard bound, skipping primes that divide d.

TIMINGS: This is perhaps the fastest implementation of determinants in the world. E.g., for a 500x500 random matrix with 32-bit entries on a core2 duo 2.6Ghz running OS X, Sage takes 4.12 seconds, whereas Magma takes 62.87 seconds (both with proof False). With proof=True on the same problem Sage takes 5.73 seconds. For another example, a 200x200 random matrix with 1-digit entries takes 4.18 seconds in pari, 0.18 in Sage with proof True, 0.11 in Sage with proof False, and 0.21 seconds in Magma with proof True and 0.18 in Magma with proof False.

EXAMPLES:

```
sage: A = matrix(ZZ,8,8,[3..66])
sage: A.determinant()
0

sage: A = random_matrix(ZZ,20,20)
sage: D1 = A.determinant()
sage: A._clear_cache()
sage: D2 = A.determinant(algorithm='ntl')
sage: D1 == D2
True
```

Next we try the Linbox det. Note that we must have proof=False.

```
sage: A = matrix(ZZ,5,[1,2,3,4,5,4,6,3,2,1,7,9,7,5,2,1,4,6,7,8,3,2,4,6,7])
sage: A.determinant(algorithm='linbox')
...
RuntimeError: you must pass the proof=False option to the determinant command to use Linbox'
sage: A.determinant(algorithm='linbox',proof=False)
-21
sage: A._clear_cache()
sage: A.determinant()
-21
```

A bigger example:

```
sage: A = random_matrix(ZZ,30)
sage: d = A.determinant()
sage: A._clear_cache()
sage: A.determinant(algorithm='linbox',proof=False) == d
True
```

echelon_form()

Return the echelon form of this matrix over the integers, also known as the hermit normal form (HNF).

INPUT:

- `algorithm`
  - **'default' - max 10 rows or columns: pari with flag 0** max 75 rows or columns: pari with flag 1 larger – use padic algorithm
  - **'padic' - an asymptotically fast p-adic modular algorithm,** If your matrix has large coefficients and is small, you may also want to try this.
  - 'pari' - use PARI with flag 1
  - 'pari0' - use PARI with flag 0
  - 'pari4' - use PARI with flag 4 (use heuristic LLL)
  - **'ntl' - use NTL (only works for square matrices of** full rank!)
- `proof` - (default: True); if proof=False certain determinants are computed using a randomized hybrid p-adic multimodular strategy until it stabilizes twice (instead of up to the Hadamard bound). It is *incredibly* unlikely that one would ever get an incorrect result with proof=False.
- `include_zero_rows` - (default: True) if False, don't include zero rows
- `transformation` - if given, also compute transformation matrix; only valid for padic algorithm
- `D` - (default: None) if given and the algorithm is 'ntl', then D must be a multiple of the determinant and this function will use that fact.

OUTPUT:

- `matrix` - the Hermite normal form (=echelon form over ZZ) of self.

Note: The result is *not* cached.

EXAMPLES:

```
sage: A = MatrixSpace(ZZ,2)([1,2,3,4])
sage: A.echelon_form()
[1 0]
[0 2]

sage: A = MatrixSpace(ZZ,5)(range(25))
sage: A.echelon_form()
[  5   0  -5 -10 -15]
[  0   1   2   3   4]
[  0   0   0   0   0]
[  0   0   0   0   0]
[  0   0   0   0   0]
```

TESTS: Make sure the zero matrices are handled correctly:

```
sage: m = matrix(ZZ,3,3,[0]*9)
sage: m.echelon_form()
[0 0 0]
[0 0 0]
[0 0 0]
sage: m = matrix(ZZ,3,1,[0]*3)
sage: m.echelon_form()
[0]
[0]
[0]
sage: m = matrix(ZZ,1,3,[0]*3)
sage: m.echelon_form()
[0 0 0]
```

The ultimate border case!

```
sage: m = matrix(ZZ,0,0,[])
sage: m.echelon_form()
[]
```

Note: If 'ntl' is chosen for a non square matrix this function raises a ValueError.
Special cases: 0 or 1 rows:

```
sage: a = matrix(ZZ, 1,2,[0,-1])
sage: a.hermite_form()
[0 1]
sage: a.pivots()
[1]
sage: a = matrix(ZZ, 1,2,[0,0])
sage: a.hermite_form()
[0 0]
sage: a.pivots()
[]
sage: a = matrix(ZZ,1,3); a
[0 0 0]
sage: a.echelon_form(include_zero_rows=False)
[]
sage: a.echelon_form(include_zero_rows=True)
[0 0 0]
```

Illustrate using various algorithms.:

```
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='pari')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='pari0')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='pari4')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='padic')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='default')
```

```
[1 2 3]
[0 3 6]
[0 0 0]
```

The 'ntl' algorithm doesn't work on matrices that do not have full rank.:

```
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='ntl')
...
ValueError: ntl only computes HNF for square matrices of full rank.
sage: matrix(ZZ,3,[0] +[2..9]).hermite_form(algorithm='ntl')
[1 0 0]
[0 1 0]
[0 0 3]
```

elementary_divisors()

Return the elementary divisors of self, in order.

Warning: This is MUCH faster than the smith_form function.

The elementary divisors are the invariants of the finite abelian group that is the cokernel of *left* multiplication of this matrix. They are ordered in reverse by divisibility.

INPUT:

- `self` - matrix
- `algorithm` - (default: 'pari')
  - `'pari'`: works robustly, but is slower.
  - `'linbox'` - use linbox (currently off, broken)

OUTPUT: list of integers

Note: These are the invariants of the cokernel of *left* multiplication:

```
sage: M = Matrix([[3,0,1],[0,1,0]])
sage: M
[3 0 1]
[0 1 0]
sage: M.elementary_divisors()
[1, 1]
sage: M.transpose().elementary_divisors()
[1, 1, 0]
```

EXAMPLES:

```
sage: matrix(3, range(9)).elementary_divisors()
[1, 3, 0]
sage: matrix(3, range(9)).elementary_divisors(algorithm='pari')
[1, 3, 0]
sage: C = MatrixSpace(ZZ,4)([3,4,5,6,7,3,8,10,14,5,6,7,2,2,10,9])
sage: C.elementary_divisors()
[1, 1, 1, 687]

sage: M = matrix(ZZ, 3, [1,5,7, 3,6,9, 0,1,2])
sage: M.elementary_divisors()
[1, 1, 6]
```

This returns a copy, which is safe to change:

```
sage: edivs = M.elementary_divisors()
sage: edivs.pop()
6
sage: M.elementary_divisors()
[1, 1, 6]
```

See Also:

smith_form()

frobenius()

Return the Frobenius form (rational canonical form) of this matrix.

INPUT: flag -an integer:

- 0 - (default) return the Frobenius form of this matrix.
- 1 - return only the elementary divisor polynomials, as polynomials in var.
- 2 - return a two-components vector [F,B] where F is the Frobenius form and B is the basis change so that $M = B^{-1}FB$.
- var - a string (default: 'x')

INPUT:

- flag - 0 (default), 1 or 2 as described above

ALGORITHM: uses pari's matfrobenius()

EXAMPLE:

```
sage: A = MatrixSpace(ZZ, 3)(range(9))
sage: A.frobenius(0)
[ 0  0  0]
[ 1  0 18]
[ 0  1 12]
sage: A.frobenius(1)
[x^3 - 12*x^2 - 18*x]
sage: A.frobenius(1, var='y')
[y^3 - 12*y^2 - 18*y]
sage: A.frobenius(2)
([ 0  0  0]
[ 1  0 18]
[ 0  1 12],
[    -1      2     -1]
[     0  23/15 -14/15]
[     0  -2/15   1/15])
sage: a=matrix([])
sage: a.frobenius(2)
([], [])
sage: a.frobenius(0)
[]
sage: a.frobenius(1)
[]
sage: B = random_matrix(ZZ,2,3)
sage: B.frobenius()
...
ArithmeticError: frobenius matrix of non-square matrix not defined.
```

AUTHORS:

- Martin Albrect (2006-04-02)

TODO: - move this to work for more general matrices than just over Z. This will require fixing how PARI polynomials are coerced to Sage polynomials.

gcd()

Return the gcd of all entries of self; very fast.

EXAMPLES:

```
sage: a = matrix(ZZ,2, [6,15,-6,150])
sage: a.gcd()
3
```

height()

Return the height of this matrix, i.e., the max absolute value of the entries of the matrix.

OUTPUT: A nonnegative integer.

EXAMPLE:

```
sage: a = Mat(ZZ,3)(range(9))
sage: a.height()
8
sage: a = Mat(ZZ,2,3)([-17,3,-389,15,-1,0]); a
[ -17    3 -389]
[  15   -1    0]
sage: a.height()
389
```

hermite_form()

Return the Hermite normal form of self.

This is a synonym for `self.echelon_form(...)`. See the documentation for `self.echelon_form` for more details.

EXAMPLES:

```
sage: A = matrix(ZZ, 3, 5, [-1, -1, -2, 2, -2, -4, -19, -17, 1, 2, -3, 1, 1, -4, 1])
sage: E, U = A.hermite_form(transformation=True)
sage: E
[   1    0   52 -133  109]
[   0    1   19  -47   38]
[   0    0   69 -178  145]
sage: U
[-46   3  11]
[-16   1   4]
[-61   4  15]
sage: U*A
[   1    0   52 -133  109]
[   0    1   19  -47   38]
[   0    0   69 -178  145]
sage: A.hermite_form()
[   1    0   52 -133  109]
[   0    1   19  -47   38]
[   0    0   69 -178  145]
```

TESTS: This example illustrated trac 2398.

```
sage: a = matrix([(0, 0, 3), (0, -2, 2), (0, 1, 2), (0, -2, 5)])
sage: a.hermite_form()
[0 1 2]
[0 0 3]
[0 0 0]
[0 0 0]
```

index_in_saturation()

Return the index of self in its saturation.

INPUT:

- `proof` - (default: use proof.linear_algebra()); if False, the determinant calculations are done with proof=False.

OUTPUT:

- `positive integer` - the index of the row span of this matrix in its saturation

ALGORITHM: Use Hermite normal form twice to find an invertible matrix whose inverse transforms a matrix with the same row span as self to its saturation, then compute the determinant of that matrix.

EXAMPLES:

```
sage: A = matrix(ZZ, 2,3, [1..6]); A
[1 2 3]
[4 5 6]
sage: A.index_in_saturation()
3
sage: A.saturation()
[1 2 3]
[1 1 1]
```

insert_row()

Create a new matrix from self with.

INPUT:

- index - integer
- row - a vector

EXAMPLES:

```
sage: X = matrix(ZZ,3,range(9)); X
[0 1 2]
[3 4 5]
[6 7 8]
sage: X.insert_row(1, [1,5,-10])
[  0   1   2]
[  1   5 -10]
[  3   4   5]
[  6   7   8]
sage: X.insert_row(0, [1,5,-10])
[  1   5 -10]
[  0   1   2]
[  3   4   5]
[  6   7   8]
sage: X.insert_row(3, [1,5,-10])
[  0   1   2]
[  3   4   5]
[  6   7   8]
[  1   5 -10]
```

is_LLL_reduced()

Return True if this lattice is (δ, η)-LLL reduced. See self.LLL for a definition of LLL reduction.

INPUT:

- delta - parameter as described above (default: 3/4)
- eta - parameter as described above (default: 0.501)

EXAMPLE:

```
sage: A = random_matrix(ZZ, 10, 10)
sage: L = A.LLL()
sage: A.is_LLL_reduced()
False
sage: L.is_LLL_reduced()
True
```

kernel_matrix()

The options are exactly like self.kernel(...), but returns a matrix A whose rows form a basis for the left kernel, i.e., so that A*self = 0.

This is mainly useful to avoid all overhead associated with creating a free module.

EXAMPLES:

```
sage: A = matrix(ZZ, 3, 3, [1..9])
sage: A.kernel_matrix()
[-1  2 -1]
```

Note that the basis matrix returned above is not in Hermite/echelon form.

```
sage: A.kernel()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1 -2  1]
```

We compute another kernel:

```
sage: A = matrix(ZZ, 4, 2, [2, -1, 1, 1, -18, -1, -1, -5])
sage: K = A.kernel_matrix(); K
[-17 -20  -3   0]
[  7   3   1  -1]
```

K is a basis for the left kernel:

```
sage: K*A
[0 0]
[0 0]
```

We illustrate the LLL flag:

```
sage: L = A.kernel_matrix(LLL=True); L
[  7   3   1  -1]
[  4 -11   0  -3]
sage: K.hermite_form()
[ 1 64  3 12]
[ 0 89  4 17]
sage: L.hermite_form()
[ 1 64  3 12]
[ 0 89  4 17]
```

Two kernel matrices via PARI, one of full rank:

```
sage: B = matrix(ZZ, [[2,-1,1],[1,-1,0],[-5,4,-1],[-8,6,-2]])
sage: B.kernel_matrix(algorithm='pari')
[ 1  1 -1  1]
[ 0  2  2 -1]
sage: C = matrix(ZZ, [[1,1,2],[1,1,4]])
sage: D = C.kernel_matrix(algorithm='pari'); D
[]
sage: D.ncols()
2
```

minpoly()

INPUT:

- `var` - a variable name
- `algorithm` - 'linbox' (default) 'generic'

Note: Linbox charpoly disabled on 64-bit machines, since it hangs in many cases.

EXAMPLES:

```
sage: A = matrix(ZZ,6, range(36))
sage: A.minpoly()
x^3 - 105*x^2 - 630*x
sage: n=6; A = Mat(ZZ,n)([k^2 for k in range(n^2)])
sage: A.minpoly()
x^4 - 2695*x^3 - 257964*x^2 + 1693440*x
```

pivots()

Return the pivot column positions of this matrix as a list of Python integers.

This returns a list, of the position of the first nonzero entry in each row of the echelon form.

OUTPUT:

- `list` - a list of Python ints

EXAMPLES:

```
sage: n = 3; A = matrix(ZZ,n,range(n^2)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.pivots()
[0, 1]
sage: A.echelon_form()
[ 3  0 -3]
[ 0  1  2]
[ 0  0  0]
```

prod_of_row_sums()

Return the product of the sums of the entries in the submatrix of self with given columns.

INPUT:

- `cols` - a list (or set) of integers representing columns of self.

OUTPUT: an integer

EXAMPLES:

```
sage: a = matrix(ZZ,2,3,[1..6]); a
[1 2 3]
[4 5 6]
sage: a.prod_of_row_sums([0,2])
40
sage: (1+3)*(4+6)
40
sage: a.prod_of_row_sums(set([0,2]))
40
```

randomize()

Randomize density proportion of the entries of this matrix, leaving the rest unchanged.

The parameters are the same as the integer ring's random_element function.

If x and y are given, randomized entries of this matrix to be between x and y and have density 1.

INPUT:

- `self` - a mutable matrix over ZZ
- `density` - a float between 0 and 1
- `x, y` - if not None are passed to ZZ.random_element function as the upper and lower endpoints in the uniform distribution
- `distribution` - would also be passed into ZZ.random_element if given

OUTPUT:

- `None` - the matrix is modified in place

EXAMPLES:

```
sage: A = matrix(ZZ, 2,3, [1..6]); A
[1 2 3]
[4 5 6]
sage: A.randomize()
sage: A
```

```
[-8  2  0]
[ 0  1 -1]
sage: A.randomize(x=-30,y=30)
sage: A
[  5 -19  24]
[ 24  23  -9]
```

rank()

Return the rank of this matrix.

OUTPUT:

- nonnegative integer - the rank

Note: The rank is cached.

ALGORITHM: First check if the matrix has maxim posible rank by working modulo one random prime. If not call Linbox's rank function.

EXAMPLES:

```
sage: a = matrix(ZZ,2,3,[1..6]); a
[1 2 3]
[4 5 6]
sage: a.rank()
2
sage: a = matrix(ZZ,3,3,[1..9]); a
[1 2 3]
[4 5 6]
[7 8 9]
sage: a.rank()
2
```

Here's a bigger example - the rank is of course still 2:

```
sage: a = matrix(ZZ,100,[1..100^2]); a.rank()
2
```

rational_reconstruction()

Use rational reconstruction to lift self to a matrix over the rational numbers (if possible), where we view self as a matrix modulo N.

INPUT:

- N - an integer

OUTPUT:

- matrix - over QQ or raise a ValueError

EXAMPLES: We create a random 4x4 matrix over ZZ.

```
sage: A = matrix(ZZ, 4, [4, -4, 7, 1, -1, 1, -1, -12, -1, -1, 1, -1, -3, 1, 5, -1])
```

There isn't a unique rational reconstruction of it:

```
sage: A.rational_reconstruction(11)
...
ValueError: Rational reconstruction of 4 (mod 11) does not exist.
```

We throw in a denominator and reduce the matrix modulo 389 - it does rationally reconstruct.

```
sage: B = (A/3 % 389).change_ring(ZZ)
sage: B.rational_reconstruction(389) == A/3
True
```

right_kernel()

Return the right kernel of this matrix, as a module over the integers. This is the saturated ZZ-module spanned by all the column vectors v such that self*v = 0.

INPUT:

- `algorithm` - 'padic': a new p-adic based algorithm 'pari': use PARI
- `LLL` - bool (default: False); if True the basis is an LLL reduced basis; otherwise, it is an echelon basis.
- `proof` - None (default: proof.linear_algebra()); if False, impacts how determinants are computed.

By convention if self has 0 columnss, the right kernel is of dimension 0, whereas the right kernel is the whole domain if self has 0 rows.

EXAMPLES:

```
sage: M = MatrixSpace(ZZ,2,4)(range(8))
sage: M.right_kernel()
Free module of degree 4 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -3  2]
[ 0  1 -2  1]
```

saturation()

Return a saturation matrix of self, which is a matrix whose rows span the saturation of the row span of self. This is not unique.

The saturation of a $\mathbf{Z}$ module M embedded in $\mathbf{Z}^n$ is the a module S that contains M with finite index such that $\mathbf{Z}^n/S$ is torsion free. This function takes the row span M of self, and finds another matrix of full rank with row span the saturation of M.

INPUT:

- `p` - (default: 0); if nonzero given, saturate only at the prime p, i.e., return a matrix whose row span is a $\mathbf{Z}$-module S that contains self and such that the index of S in its saturation is coprime to p. If p is None, return full saturation of self.
- `proof` - (default: use proof.linear_algebra()); if False, the determinant calculations are done with proof=False.
- `max_dets` - (default: 5); technical parameter - max number of determinant to compute when bounding prime divisor of self in its saturation.

OUTPUT:

- `matrix` - a matrix over ZZ

Note: The result is *not* cached.

ALGORITHM: 1. Replace input by a matrix of full rank got from a subset of the rows. 2. Divide out any common factors from rows. 3. Check max_dets random dets of submatrices to see if their gcd (with p) is 1 - if so matrix is saturated and we're done. 4. Finally, use that if A is a matrix of full rank, then $hnf(transpose(A))^{-1} * A$ is a saturation of A.

EXAMPLES:

```
sage: A = matrix(ZZ, 3, 5, [-51, -1509, -71, -109, -593, -19, -341, 4, 86, 98, 0, -246, -11,
sage: A.echelon_form()
[      1       5    2262   20364   56576]
[      0       6   35653  320873  891313]
[      0       0   42993  386937 1074825]
sage: S = A.saturation(); S
[  -51 -1509   -71  -109  -593]
[  -19  -341     4    86    98]
[   35   994    43    51   347]
```

Notice that the saturation spans a different module than A.

```
sage: S.echelon_form()
[ 1  2  0  8 32]
[ 0  3  0 -2 -6]
[ 0  0  1  9 25]
```

```
sage: V = A.row_space(); W = S.row_space()
sage: V.is_submodule(W)
True
sage: V.index_in(W)
85986
sage: V.index_in_saturation()
85986
```

We illustrate each option:

```
sage: S = A.saturation(p=2)
sage: S = A.saturation(proof=False)
sage: S = A.saturation(max_dets=2)
```

smith_form()

Returns matrices S, U, and V such that S = U*self*V, and S is in Smith normal form. Thus S is diagonal with diagonal entries the ordered elementary divisors of S.

Warning: The elementary_divisors function, which returns the diagonal entries of S, is VASTLY faster than this function.

The elementary divisors are the invariants of the finite abelian group that is the cokernel of this matrix. They are ordered in reverse by divisibility.

EXAMPLES:

```
sage: A = MatrixSpace(IntegerRing(), 3)(range(9))
sage: D, U, V = A.smith_form()
sage: D
[1 0 0]
[0 3 0]
[0 0 0]
sage: U
[ 0  1  0]
[ 0 -1  1]
[-1  2 -1]
sage: V
[-1  4  1]
[ 1 -3 -2]
[ 0  0  1]
sage: U*A*V
[1 0 0]
[0 3 0]
[0 0 0]
```

It also makes sense for nonsquare matrices:

```
sage: A = Matrix(ZZ,3,2,range(6))
sage: D, U, V = A.smith_form()
sage: D
[1 0]
[0 2]
[0 0]
sage: U
[ 0  1  0]
[ 0 -1  1]
[-1  2 -1]
sage: V
[-1  3]
[ 1 -2]
sage: U * A * V
```

```
[1 0]
[0 2]
[0 0]
```

Empty matrices are handled sensibly (see trac #3068):

```
sage: m = MatrixSpace(ZZ, 2,0)(0); d,u,v = m.smith_form(); u*m*v == d
True
sage: m = MatrixSpace(ZZ, 0,2)(0); d,u,v = m.smith_form(); u*m*v == d
True
sage: m = MatrixSpace(ZZ, 0,0)(0); d,u,v = m.smith_form(); u*m*v == d
True
```

See Also:

elementary_divisors()

stack()

Return the matrix self on top of other: [self] [other]

EXAMPLES:

```
sage: M = Matrix(ZZ, 2, 3, range(6))
sage: N = Matrix(ZZ, 1, 3, [10,11,12])
sage: M.stack(N)
[ 0  1  2]
[ 3  4  5]
[10 11 12]
```

symplectic_form()

Find a symplectic basis for self if self is an anti-symmetric, alternating matrix.

Returns a pair (F, C) such that the rows of C form a symplectic basis for self and F = C * self * C.transpose().

Raises a ValueError if self is not anti-symmetric, or self is not alternating.

Anti-symmetric means that $M = -M^t$. Alternating means that the diagonal of M is identically zero.

A symplectic basis is a basis of the form $e_1, \ldots, e_j, f_1, \ldots f_j, z_1, \ldots, z_k$ such that

- $z_i M v^t = 0$ for all vectors v
- $e_i M {e_j}^t = 0$ for all i, j
- $f_i M {f_j}^t = 0$ for all i, j
- $e_i M {f_i}^t = 1$ for all i
- $e_i M {f_j}^t = 0$ **for all** i **not equal** j.

The ordering for the factors $d_i | d_{i+1}$ and for the placement of zeroes was chosen to agree with the output of smith_form.

See the example for a pictorial description of such a basis.

EXAMPLES:

```
sage: E = matrix(ZZ, 5, 5, [0, 14, 0, -8, -2, -14, 0, -3, -11, 4, 0, 3, 0, 0, 0, 8, 11, 0, 0
[  0  14   0  -8  -2]
[-14   0  -3 -11   4]
[  0   3   0   0   0]
[  8  11   0   0   8]
[  2  -4   0  -8   0]
sage: F, C = E.symplectic_form()
sage: F
[ 0  0  1  0  0]
[ 0  0  0  2  0]
[-1  0  0  0  0]
[ 0 -2  0  0  0]
```

```
[ 0  0  0  0  0]
sage: F == C * E * C.transpose()
True
sage: E.smith_form()[0]
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 2 0 0]
[0 0 0 2 0]
[0 0 0 0 0]
```

transpose()

Returns the transpose of self, without changing self.

EXAMPLES:

We create a matrix, compute its transpose, and note that the original matrix is not changed.

```
sage: A = matrix(ZZ,2,3,xrange(6))
sage: type(A)
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
sage: B = A.transpose()
sage: print B
[0 3]
[1 4]
[2 5]
sage: print A
[0 1 2]
[3 4 5]

sage: A.subdivide(None, 1); A
[0|1 2]
[3|4 5]
sage: A.transpose()
[0 3]
[---]
[1 4]
[2 5]
```

clear_mpz_globals()

gmp_randrange()

init_mpz_globals()

tune_multiplication()

Compare various multiplication algorithms.

INPUT:

- `k` - integer; affects numbers of trials
- `nmin` - integer; smallest matrix to use
- `nmax` - integer; largest matrix to use
- `bitmin` - integer; smallest bitsize
- `bitmax` - integer; largest bitsize

OUTPUT:

- `prints what doing then who wins` - multimodular or classical

EXAMPLES:

```
sage: from sage.matrix.matrix_integer_dense import tune_multiplication
sage: tune_multiplication(2, nmin=10, nmax=60, bitmin=2,bitmax=8)
10 2 0.2
...
```

32.17 Dense matrices over the rational field.

EXAMPLES:

We create a 3x3 matrix with rational entries and do some operations with it.

```
sage: a = matrix(QQ, 3,3, [1,2/3, -4/5, 1,1,1, 8,2, -3/19]); a
[    1   2/3  -4/5]
[    1     1     1]
[    8     2 -3/19]
sage: a.det()
2303/285
sage: a.charpoly()
x^3 - 35/19*x^2 + 1259/285*x - 2303/285
sage: b = a^(-1); b
[ -615/2303  -426/2303   418/2303]
[ 2325/2303  1779/2303  -513/2303]
[-1710/2303   950/2303    95/2303]
sage: b.det()
285/2303
sage: a == b
False
sage: a < b
False
sage: b < a
True
sage: a > b
True
sage: a*b
[1 0 0]
[0 1 0]
[0 0 1]
```

TESTS:

```
sage: a = matrix(QQ,2,range(4), sparse=False)
sage: loads(dumps(a)) == a
True
```

class MatrixWindow()

class Matrix_rational_dense()

antitranspose()

Returns the antitranspose of self, without changing self.

EXAMPLES:

```
sage: A = matrix(QQ,2,3,range(6))
sage: type(A)
<type 'sage.matrix.matrix_rational_dense.Matrix_rational_dense'>
```

```
sage: A.antitranspose()
[5 2]
[4 1]
[3 0]
sage: A
[0 1 2]
[3 4 5]

sage: A.subdivide(1,2); A
[0 1|2]
[---+-]
[3 4|5]
sage: A.antitranspose()
[5|2]
[-+-]
[4|1]
[3|0]
```

change_ring()

Create the matrix over R with entries the entries of self coerced into R.

EXAMPLES:

```
sage: a = matrix(QQ,2,[1/2,-1,2,3])
sage: a.change_ring(GF(3))
[2 2]
[2 0]
sage: a.change_ring(ZZ)
...
TypeError: matrix has denominators so can't change to ZZ.
sage: b = a.change_ring(QQ['x']); b
[1/2  -1]
[  2   3]
sage: b.parent()
Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial Ring in x over Rational
```

TESTS:

Make sure that subdivisions are preserved when changing rings:

```
sage: a = matrix(QQ, 3, range(9))
sage: a.subdivide(2,1); a
[0|1 2]
[3|4 5]
[-+---]
[6|7 8]
sage: a.change_ring(ZZ).change_ring(QQ)
[0|1 2]
[3|4 5]
[-+---]
[6|7 8]
sage: a.change_ring(GF(3))
[0|1 2]
[0|1 2]
[-+---]
[0|1 2]
```

charpoly()

Return the characteristic polynomial of this matrix.

INPUT:

- `var` - 'x' (string)
- `algorithm` - 'linbox' (default) or 'generic'

OUTPUT: a polynomial over the rational numbers.

EXAMPLES:

```
sage: a = matrix(QQ, 3, [4/3, 2/5, 1/5, 4, -3/2, 0, 0, -2/3, 3/4])
sage: f = a.charpoly(); f
x^3 - 7/12*x^2 - 149/40*x + 97/30
sage: f(a)
[0 0 0]
[0 0 0]
[0 0 0]
```

column()

Return the i-th column of this matrix as a dense vector.

INPUT:
- `i` - integer
- `from_list` - ignored

EXAMPLES:

```
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).column(1)
(-2/3, 4/9)
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).column(1,from_list=True)
(-2/3, 4/9)
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).column(-1)
(-2/3, 4/9)
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).column(-2)
(1/5, 3/4)
```

decomposition()

Returns the decomposition of the free module on which this matrix A acts from the right (i.e., the action is x goes to x A), along with whether this matrix acts irreducibly on each factor. The factors are guaranteed to be sorted in the same way as the corresponding factors of the characteristic polynomial.

Let A be the matrix acting from the on the vector space V of column vectors. Assume that A is square. This function computes maximal subspaces W_1, ..., W_n corresponding to Galois conjugacy classes of eigenvalues of A. More precisely, let f(X) be the characteristic polynomial of A. This function computes the subspace $W_i = ker(g_(A)^n)$, where g_i(X) is an irreducible factor of f(X) and g_i(X) exactly divides f(X). If the optional parameter is_diagonalizable is True, then we let W_i = ker(g(A)), since then we know that ker(g(A)) = $ker(g(A)^n)$.

If dual is True, also returns the corresponding decomposition of V under the action of the transpose of A. The factors are guarenteed to correspond.

INPUT:

- `is_diagonalizable` - ignored
- `dual` - whether to also return decompositions for the dual
- `algorithm`
  - 'default': use default algorithm for computing Echelon forms
  - 'multimodular': much better if the answers factors have small height
- `height_guess` - positive integer; only used by the multimodular algorithm
- `proof` - bool or None (default: None, see proof.linear_algebra or sage.structure.proof); only used by the multimodular algorithm. Note that the Sage global default is proof=True.

Note: IMPORTANT: If you expect that the subspaces in the answer are spanned by vectors with small height coordinates, use algorithm='multimodular' and height_guess=1; this is potentially much faster than the default. If you know for a fact the answer will be very small, use algorithm='multimodular', height_guess=bound on height, proof=False.

You can get very very fast decomposition with proof=False.

EXAMPLES:

```
sage: a = matrix(QQ,3,[1..9])
sage: a.decomposition()
[
(Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1], True),
(Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2], True)
]
```

denominator()

Return the denominator of this matrix.

OUTPUT: a Sage Integer

EXAMPLES:

```
sage: b = matrix(QQ,2,range(6)); b[0,0]=-5007/293; b
[-5007/293         1         2]
[         3         4         5]
sage: b.denominator()
293
```

determinant()

Return the determinant of this matrix.

INPUT:

- `proof` - bool or None; if None use proof.linear_algebra(); only relevant for the padic algorithm.
- `algorithm`:

  "default" – use PARI for up to 7 rows, then use integer
  "pari" – use PARI
  "integer" – clear denominators and call det on integer matrix

Note: It would be *VERY VERY* hard for det to fail even with proof=False.

ALGORITHM: Clear denominators and call the integer determinant function.

EXAMPLES:

```
sage: m = matrix(QQ,3,[1,2/3,4/5, 2,2,2, 5,3,2/5])
sage: m.determinant()
-34/15
sage: m.charpoly()
x^3 - 17/5*x^2 - 122/15*x + 34/15
```

echelon_form()

INPUT:

- `algorithm`
  - 'default' (default): use heuristic choice
  - 'padic': an algorithm based on the IML p-adic solver.
  - 'multimodular': uses a multimodular algorithm the uses linbox modulo many primes.
  - 'classical': just clear each column using Gauss elimination
- `height_guess, **kwds` - all passed to the multimodular algorithm; ignored by the p-adic algorithm.
- `proof` - bool or None (default: None, see proof.linear_algebra or sage.structure.proof). Passed to the multimodular algorithm. Note that the Sage global default is proof=True.

OUTPUT: self is no in reduced row echelon form.

EXAMPLES:

```
sage: a = matrix(QQ, 4, range(16)); a[0,0] = 1/19; a[0,1] = 1/5; a
[1/19  1/5    2    3]
[   4    5    6    7]
[   8    9   10   11]
[  12   13   14   15]
sage: a.echelon_form()
[      1       0       0 -76/157]
[      0       1       0  -5/157]
[      0       0       1 238/157]
[      0       0       0       0]
sage: a.echelon_form(algorithm='multimodular')
[      1       0       0 -76/157]
[      0       1       0  -5/157]
[      0       0       1 238/157]
[      0       0       0       0]
```

echelonize()

INPUT:

- algorithm
- 'default' (default): use heuristic choice
- 'padic': an algorithm based on the IML p-adic solver.
- 'multimodular': uses a multimodular algorithm the uses linbox modulo many primes.
- 'classical': just clear each column using Gauss elimination
- height_guess, **kwds - all passed to the multimodular algorithm; ignored by the p-adic algorithm.
- proof - bool or None (default: None, see proof.linear_algebra or sage.structure.proof). Passed to the multimodular algorithm. Note that the Sage global default is proof=True.

OUTPUT:

- matrix - the reduced row echelon for of self.

EXAMPLES:

```
sage: a = matrix(QQ, 4, range(16)); a[0,0] = 1/19; a[0,1] = 1/5; a
[1/19  1/5    2    3]
[   4    5    6    7]
[   8    9   10   11]
[  12   13   14   15]
sage: a.echelonize(); a
[      1       0       0 -76/157]
[      0       1       0  -5/157]
[      0       0       1 238/157]
[      0       0       0       0]

sage: a = matrix(QQ, 4, range(16)); a[0,0] = 1/19; a[0,1] = 1/5
sage: a.echelonize(algorithm='multimodular'); a
[      1       0       0 -76/157]
[      0       1       0  -5/157]
[      0       0       1 238/157]
[      0       0       0       0]
```

height()

Return the height of this matrix, which is the maximum of the absolute values of all numerators and denominators of entries in this matrix.

OUTPUT: an Integer

EXAMPLES:

```
sage: b = matrix(QQ,2,range(6)); b[0,0]=-5007/293; b
[-5007/293         1         2]
[         3         4         5]
sage: b.height()
5007
```

invert()

Compute the inverse of this matrix.

Warning: This function is deprecated. Use `inverse` instead.

EXAMPLES:

```
sage: a = matrix(QQ,3,range(9))
sage: a.invert()
...
ZeroDivisionError: input matrix must be nonsingular
```

minpoly()

Return the minimal polynomial of this matrix.

INPUT:

- `var` - 'x' (string)
- `algorithm` - 'linbox' (default) or 'generic'

OUTPUT: a polynomial over the rational numbers.

EXAMPLES:

```
sage: a = matrix(QQ, 3, [4/3, 2/5, 1/5, 4, -3/2, 0, 0, -2/3, 3/4])
sage: f = a.minpoly(); f
x^3 - 7/12*x^2 - 149/40*x + 97/30
sage: a = Mat(ZZ,4)(range(16))
sage: f = a.minpoly(); f.factor()
x * (x^2 - 30*x - 80)
sage: f(a) == 0
True

sage: a = matrix(QQ, 4, [1..4^2])
sage: factor(a.minpoly())
x * (x^2 - 34*x - 80)
sage: factor(a.minpoly('y'))
y * (y^2 - 34*y - 80)
sage: factor(a.charpoly())
x^2 * (x^2 - 34*x - 80)
sage: b = matrix(QQ, 4, [-1, 2, 2, 0, 0, 4, 2, 2, 0, 0, -1, -2, 0, -4, 0, 4])
sage: a = matrix(QQ, 4, [1, 1, 0,0, 0,1,0,0, 0,0,5,0, 0,0,0,5])
sage: c = b^(-1)*a*b
sage: factor(c.minpoly())
(x - 5) * (x - 1)^2
sage: factor(c.charpoly())
(x - 5)^2 * (x - 1)^2
```

prod_of_row_sums()

randomize()

Randomize density proportion of the entries of this matrix, leaving the rest unchanged.

If x and y are given, randomized entries of this matrix have numerators and denominators bounded by x and y and have density 1.

INPUT:

- density – number between 0 and 1 (default: 1)
- num_bound – numerator bound (default: 2)
- den_bound – denominator bound (default: 2)
- distribution – None or '1/n' (default: None); if '1/n' then num_bound, den_bound are ignored and numbers are chosen using the GMP function mpq_randomize_entry_recip_uniform

EXAMPLES:

```
sage: a = matrix(QQ,2,4); a.randomize(); a
[ 0 -1  2 -2]
[ 1 -1  2  1]
sage: a = matrix(QQ,2,4); a.randomize(density=0.5); a
[ -1  -2   0   0]
[  0   0 1/2   0]
sage: a = matrix(QQ,2,4); a.randomize(num_bound=100, den_bound=100); a
[ 14/27  21/25  43/42 -48/67]
[-19/55  64/67 -11/51     76]
sage: a = matrix(QQ,2,4); a.randomize(distribution='1/n'); a
[      3     1/9     1/2     1/4]
[      1    1/39       2 -1955/2]
```

rank()

Return the rank of this matrix.

EXAMPLES:: sage: matrix(QQ,3,[1..9]).rank() 2 sage: matrix(QQ,100,[1..100^2]).rank() 2

right_kernel()

Return the right kernel of this matrix, as a vector space over QQ. For a left kernel use self.left_kernel() or just self.kernel().

INPUT:

- algorithm - 'padic' (or 'default'): use IML's p-adic nullspace algorithm
- anything else - passed on to the generic echelon-form based algorithm.
- **kwds - passed onto to echelon form algorithm in the echelon case.

EXAMPLES:

A non-trivial right kernel over the rationals::

```
sage: A = matrix(QQ, [[2,1,-5,-8],[-1,-1,4,6],[1,0,-1,-2]])
sage: A.right_kernel()
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[   1    0   -2  3/2]
[   0    1    1 -1/2]
```

A trivial right kernel, plus left kernel (via superclass)::

```
sage: M=Matrix(QQ,[[1/2,3],[0,1],[1,1]])
sage: M.right_kernel()
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix:
[]
sage: M.left_kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[   1 -5/2 -1/2]
```

row()

Return the i-th row of this matrix as a dense vector.

INPUT: • `i` - integer
• `from_list` - ignored

EXAMPLES:

```
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).row(1)
(3/4, 4/9)
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).row(1,from_list=True)
(3/4, 4/9)
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).row(-2)
(1/5, -2/3)
```

set_row_to_multiple_of_row()
Set row i equal to s times row j.
EXAMPLES:

```
sage: a = matrix(QQ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.set_row_to_multiple_of_row(1,0,-3)
sage: a
[ 0  1  2]
[ 0 -3 -6]
```

transpose()
Returns the transpose of self, without changing self.
EXAMPLES:
We create a matrix, compute its transpose, and note that the original matrix is not changed.

```
sage: A = matrix(QQ,2,3,xrange(6))
sage: type(A)
<type 'sage.matrix.matrix_rational_dense.Matrix_rational_dense'>
sage: B = A.transpose()
sage: print B
[0 3]
[1 4]
[2 5]
sage: print A
[0 1 2]
[3 4 5]

sage: A.subdivide(None, 1); A
[0|1 2]
[3|4 5]
sage: A.transpose()
[0 3]
[---]
[1 4]
[2 5]
```

clear_mpz_globals()

gmp_randrange()

init_mpz_globals()

32.18 Dense matrices over the Real Double Field using NumPy

EXAMPLES:

```
sage: b=Mat(RDF,2,3).basis()
sage: b[0]
[1.0 0.0 0.0]
[0.0 0.0 0.0]
```

We deal with the case of zero rows or zero columns:

```
sage: m = MatrixSpace(RDF,0,3)
sage: m.zero_matrix()
[]
```

TESTS:

```
sage: a = matrix(RDF,2,range(4), sparse=False)
sage: loads(dumps(a)) == a
True
sage: MatrixSpace(RDF,0,0).zero_matrix().inverse()
[]
```

AUTHORS:

- Jason Grout (2008-09): switch to numpy backend, factored out the Matrix_double_dense class
- Josh Kantor
- William Stein: many bug fixes and touch ups.

class Matrix_real_double_dense()

Class that implements matrices over the real double field. These are supposed to be fast matrix operations using C doubles. Most operations are implemented using numpy which will call the underlying BLAS on the system.

EXAMPLES:

```
sage: m = Matrix(RDF, [[1,2],[3,4]])
sage: m**2
[ 7.0 10.0]
[15.0 22.0]
sage: n= m^(-1); n
[-2.0  1.0]
[ 1.5 -0.5]
```

To compute eigenvalues the use the functions left_eigenvectors or right_eigenvectors

```
sage: p,e = m.right_eigenvectors()
```

the result of eigen is a pair (p,e), where p is a list of eigenvalues and the e is a matrix whose columns are the eigenvectors.

To solve a linear system Ax = b where A = [[1,2],[3,4]] and b = [5,6].

```
sage: b = vector(RDF,[5,6])
sage: m.solve_left(b)
(-4.0, 4.5)
```

See the commands qr, lu, and svd for QR, LU, and singular value decomposition.

32.19 Dense matrices over the Complex Double Field using NumPy

EXAMPLES:

```
sage: b=Mat(CDF,2,3).basis()
sage: b[0]
[1.0   0   0]
[  0   0   0]
```

We deal with the case of zero rows or zero columns:

```
sage: m = MatrixSpace(CDF,0,3)
sage: m.zero_matrix()
[]
```

TESTS:

```
sage: a = matrix(CDF,2,[i+(4-i)*I for i in range(4)], sparse=False)
sage: loads(dumps(a)) == a
True
sage: Mat(CDF,0,0).zero_matrix().inverse()
[]
```

AUTHORS:

- Jason Grout (2008-09): switch to numpy backend
- Josh Kantor
- William Stein: many bug fixes and touch ups.

class Matrix_complex_double_dense()

Class that implements matrices over the real double field. These are supposed to be fast matrix operations using C doubles. Most operations are implemented using numpy which will call the underlying BLAS on the system.

EXAMPLES:

```
sage: m = Matrix(CDF, [[1,2*I],[3+I,4]])
sage: m**2
[-1.0 + 6.0*I       10.0*I]
[15.0 + 5.0*I 14.0 + 6.0*I]
sage: n= m^(-1); n
[  0.333333333333 + 0.333333333333*I   0.166666666667 - 0.166666666667*I]
[ -0.166666666667 - 0.333333333333*I 0.0833333333333 + 0.0833333333333*I]
```

To compute eigenvalues the use the functions left_eigenvectors or right_eigenvectors

```
sage: p,e = m.right_eigenvectors()
```

the result of eigen is a pair (p,e), where p is a list of eigenvalues and the e is a matrix whose columns are the eigenvectors.

To solve a linear system Ax = b where A = [[1,2] and b = [5,6] [3,4]]

```
sage: b = vector(CDF,[5,6])
sage: m.solve_left(b)
(2.66666666667 + 0.666666666667*I, -0.333333333333 - 1.16666666667*I)
```

See the commands qr, lu, and svd for QR, LU, and singular value decomposition.

CHAPTER

THIRTYTHREE

MODULES

33.1 Abstract base class for modules

class Module()

Generic module class.

category()

Return the category to which this module belongs.

endomorphism_ring()

Return the endomorphism ring of this module in its category.

is_atomic_repr()

True if the elements have atomic string representations, in the sense that they print if they print at s, then -s means the negative of s. For example, integers are atomic but polynomials are not.

is_Module()

Return True if x is a module.

EXAMPLES:

```
sage: from sage.modules.module import is_Module
sage: M = FreeModule(RationalField(),30)
sage: is_Module(M)
True
sage: is_Module(10)
False
```

is_VectorSpace()

Return True if x is a vector space.

EXAMPLES:

```
sage: from sage.modules.module import is_Module, is_VectorSpace
sage: M = FreeModule(RationalField(),30)
sage: is_VectorSpace(M)
True
sage: M = FreeModule(IntegerRing(),30)
sage: is_Module(M)
True
sage: is_VectorSpace(M)
False
```

33.2 Free modules

Sage supports computation with free modules over an arbitrary commutative ring. Nontrivial functionality is available over $\mathbf{Z}$ and fields. All free modules over an integral domain are equipped with an embedding in an ambient vector space and an inner product, which you can specify and change.

Create the free module of rank n over an arbitrary commutative ring R using the command `FreeModule(R,n)`. Equivalently, `R^n` also creates that free module.

The following example illustrates the creation of both a vector space and a free module over the integers and a submodule of it. Use the functions `FreeModule`, `span` and member functions of free modules to create free modules. *Do not use the FreeModulexxx constructors directly.*

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: W = V.subspace([[1,2,7], [1,1,0]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -7]
[ 0  1  7]
sage: C = VectorSpaces(FiniteField(7))
sage: C
Category of vector spaces over Finite Field of size 7
sage: C(W)
Vector space of degree 3 and dimension 2 over Finite Field of size 7
Basis matrix:
[1 0 0]
[0 1 0]


sage: M = ZZ^3
sage: C = VectorSpaces(FiniteField(7))
sage: C(M)
Vector space of dimension 3 over Finite Field of size 7
sage: W = M.submodule([[1,2,7], [8,8,0]])
sage: C(W)
Vector space of degree 3 and dimension 2 over Finite Field of size 7
Basis matrix:
[1 0 0]
[0 1 0]
```

We illustrate the exponent notation for creation of free modules.

```
sage: ZZ^4
Ambient free module of rank 4 over the principal ideal domain Integer Ring
sage: QQ^2
Vector space of dimension 2 over Rational Field
sage: RR^3
Vector space of dimension 3 over Real Field with 53 bits of precision
```

Base ring:

```
sage: R.<x,y> = QQ[]
sage: M = FreeModule(R,2)
sage: M.base_ring()
Multivariate Polynomial Ring in x, y over Rational Field
```

```
sage: VectorSpace(QQ, 10).base_ring()
Rational Field
```

TESTS: We intersect a zero-dimensional vector space with a 1-dimension submodule.

```
sage: V = (QQ^1).span([])
sage: W = ZZ^1
sage: V.intersection(W)
Free module of degree 1 and rank 0 over Integer Ring
Echelon basis matrix:
[]
```

We construct subspaces of real and complex double vector spaces and verify that the element types are correct:

```
sage: V = FreeModule(RDF, 3); V
Vector space of dimension 3 over Real Double Field
sage: V.0
(1.0, 0.0, 0.0)
sage: type(V.0)
<type 'sage.modules.vector_real_double_dense.Vector_real_double_dense'>
sage: W = V.span([V.0]); W
Vector space of degree 3 and dimension 1 over Real Double Field
Basis matrix:
[1.0 0.0 0.0]
sage: type(W.0)
<type 'sage.modules.vector_real_double_dense.Vector_real_double_dense'>
sage: V = FreeModule(CDF, 3); V
Vector space of dimension 3 over Complex Double Field
sage: type(V.0)
<type 'sage.modules.vector_complex_double_dense.Vector_complex_double_dense'>
sage: W = V.span_of_basis([CDF.0 * V.1]); W
Vector space of degree 3 and dimension 1 over Complex Double Field
User basis matrix:
[    0 1.0*I     0]
sage: type(W.0)
<type 'sage.modules.vector_complex_double_dense.Vector_complex_double_dense'>
```

Basis vectors are immutable:

```
sage: A = span([[1,2,3], [4,5,6]], ZZ)
sage: A.0
(1, 2, 3)
sage: A.0[0] = 5
...
ValueError: vector is immutable; please change a copy instead (use self.copy())
```

We can save and load submodules and elements:

```
sage: M = ZZ^3
sage: M == loads(M.dumps())
True
sage: W = M.span_of_basis([[1,2,3],[4,5,19]])
sage: W == loads(W.dumps())
True
sage: v = W.0 + W.1
```

```
sage: v == loads(v.dumps())
True
```

AUTHORS:

- William Stein (2005, 2007)
- David Kohel (2007, 2008)

class ComplexDoubleVectorSpace_class(*n*)

coordinates(*v*)

class FreeModuleFactory()

Create the free module over the given commutative ring of the given rank.

INPUT:

- `base_ring` - a commutative ring
- `rank` - a nonnegative integer
- `sparse` - bool; (default False)
- `inner_product_matrix` - the inner product matrix (default None)

OUTPUT: a free module

Note: In Sage it is the case that there is only one dense and one sparse free ambient module of rank n over R.

EXAMPLES:

First we illustrate creating free modules over various base fields. The base field affects the free module that is created. For example, free modules over a field are vector spaces, and free modules over a principal ideal domain are special in that more functionality is available for them than for completely general free modules.

```
sage: FreeModule(Integers(8),10)
Ambient free module of rank 10 over Ring of integers modulo 8
sage: FreeModule(QQ,10)
Vector space of dimension 10 over Rational Field
sage: FreeModule(ZZ,10)
Ambient free module of rank 10 over the principal ideal domain Integer Ring
sage: FreeModule(FiniteField(5),10)
Vector space of dimension 10 over Finite Field of size 5
sage: FreeModule(Integers(7),10)
Vector space of dimension 10 over Ring of integers modulo 7
sage: FreeModule(PolynomialRing(QQ,'x'),5)
Ambient free module of rank 5 over the principal ideal domain Univariate Polynomial Ring in x ov
sage: FreeModule(PolynomialRing(ZZ,'x'),5)
Ambient free module of rank 5 over the integral domain Univariate Polynomial Ring in x over Inte
```

Of course we can make rank 0 free modules:

```
sage: FreeModule(RealField(100),0)
Vector space of dimension 0 over Real Field with 100 bits of precision
```

Next we create a free module with sparse representation of elements. Functionality with sparse modules is *identical* to dense modules, but they may use less memory and arithmetic may be faster (or slower!).

```
sage: M = FreeModule(ZZ,200,sparse=True)
sage: M.is_sparse()
True
sage: type(M.0)
<type 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
```

The default is dense.

```
sage: M = ZZ^200
sage: type(M.0)
<type 'sage.modules.vector_integer_dense.Vector_integer_dense'>
```

Note that matrices associated in some way to sparse free modules are sparse by default:

```
sage: M = FreeModule(Integers(8), 2)
sage: A = M.basis_matrix()
sage: A.is_sparse()
False
sage: Ms = FreeModule(Integers(8), 2, sparse=True)
sage: M == Ms  # as mathematical objects they are equal
True
sage: Ms.basis_matrix().is_sparse()
True
```

We can also specify an inner product matrix, which is used when computing inner products of elements.

```
sage: A = MatrixSpace(ZZ,2)([[1,0],[0,-1]])
sage: M = FreeModule(ZZ,2,inner_product_matrix=A)
sage: v, w = M.gens()
sage: v.inner_product(w)
0
sage: v.inner_product(v)
1
sage: w.inner_product(w)
-1
sage: (v+2*w).inner_product(w)
-2
```

You can also specify the inner product matrix by giving anything that coerces to an appropriate matrix. This is only useful if the inner product matrix takes values in the base ring.

```
sage: FreeModule(ZZ,2,inner_product_matrix=1).inner_product_matrix()
[1 0]
[0 1]
sage: FreeModule(ZZ,2,inner_product_matrix=[1,2,3,4]).inner_product_matrix()
[1 2]
[3 4]
sage: FreeModule(ZZ,2,inner_product_matrix=[[1,2],[3,4]]).inner_product_matrix()
[1 2]
[3 4]
```

create_key(*base_ring, rank, sparse=False, inner_product_matrix=None*)

TESTS:

```
sage: loads(dumps(ZZ^6)) is ZZ^6
True
sage: loads(dumps(RDF^3)) is RDF^3
True
```

create_object(*version, key*)

class FreeModule_ambient(*base_ring, rank, sparse=False*)

Ambient free module over a commutative ring.

ambient_module()

Return self, since self is ambient.

EXAMPLES:

```
sage: A = QQ^5; A.ambient_module()
Vector space of dimension 5 over Rational Field
sage: A = ZZ^5; A.ambient_module()
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

basis()

Return a basis for this ambient free module.

OUTPUT:

- Sequence - an immutable sequence with universe this ambient free module

EXAMPLES:

```
sage: A = ZZ^3; B = A.basis(); B
[
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: B.universe()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

change_ring(*R*)

Return the ambient free module over R of the same rank as self.

EXAMPLES:

```
sage: A = ZZ^3; A.change_ring(QQ)
Vector space of dimension 3 over Rational Field
sage: A = ZZ^3; A.change_ring(GF(5))
Vector space of dimension 3 over Finite Field of size 5
```

For ambient modules any change of rings is defined.

```
sage: A = GF(5)**3; A.change_ring(QQ)
Vector space of dimension 3 over Rational Field
```

coordinate_vector(*v, check=True*)

Write v in terms of the standard basis for self and return the resulting coefficients in a vector over the fraction field of the base ring.

Returns a vector c such that if B is the basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raises an ArithmeticError exception.

EXAMPLES:

```
sage: V = Integers(16)^3
sage: v = V.coordinate_vector([1,5,9]); v
(1, 5, 9)
sage: v.parent()
Ambient free module of rank 3 over Ring of integers modulo 16
```

echelon_coordinate_vector(*v, check=True*)

Same as `self.coordinate_vector(v)`, since self is an ambient free module.

INPUT:

- `v` - vector
- `check` - bool (default: True); if True, also verify that v is really in self.

OUTPUT: list

EXAMPLES:

```
sage: V = QQ^4
sage: v = V([-1/2,1/2,-1/2,1/2])
sage: v
(-1/2, 1/2, -1/2, 1/2)
sage: V.coordinate_vector(v)
(-1/2, 1/2, -1/2, 1/2)
sage: V.echelon_coordinate_vector(v)
(-1/2, 1/2, -1/2, 1/2)
sage: W = V.submodule_with_basis([[1/2,1/2,1/2,1/2],[1,0,1,0]])
sage: W.coordinate_vector(v)
(1, -1)
sage: W.echelon_coordinate_vector(v)
(-1/2, 1/2)
```

echelon_coordinates(*v, check=True*)

Returns the coordinate vector of v in terms of the echelon basis for self.

EXAMPLES:

```
sage: U = VectorSpace(QQ,3)
sage: [ U.coordinates(v) for v in U.basis() ]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
sage: [ U.echelon_coordinates(v) for v in U.basis() ]
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
sage: V = U.submodule([[1,1,0],[0,1,1]])
sage: V
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  1]
sage: [ V.coordinates(v) for v in V.basis() ]
[[1, 0], [0, 1]]
sage: [ V.echelon_coordinates(v) for v in V.basis() ]
[[1, 0], [0, 1]]
sage: W = U.submodule_with_basis([[1,1,0],[0,1,1]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 1 0]
[0 1 1]
sage: [ W.coordinates(v) for v in W.basis() ]
[[1, 0], [0, 1]]
sage: [ W.echelon_coordinates(v) for v in W.basis() ]
[[1, 1], [0, 1]]
```

echelonized_basis()

Return a basis for this ambient free module in echelon form.

EXAMPLES:

```
sage: A = ZZ^3; A.echelonized_basis()
[
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
```

echelonized_basis_matrix()

The echelonized basis matrix of self.

EXAMPLES:

```
sage: V = ZZ^4
sage: W = V.submodule([ V.gen(i)-V.gen(0) for i in range(1,4) ])
sage: W.basis_matrix()
[ 1  0  0 -1]
[ 0  1  0 -1]
[ 0  0  1 -1]
sage: W.echelonized_basis_matrix()
[ 1  0  0 -1]
[ 0  1  0 -1]
[ 0  0  1 -1]
sage: U = V.submodule_with_basis([ V.gen(i)-V.gen(0) for i in range(1,4) ])
sage: U.basis_matrix()
[-1  1  0  0]
[-1  0  1  0]
[-1  0  0  1]
sage: U.echelonized_basis_matrix()
[ 1  0  0 -1]
[ 0  1  0 -1]
[ 0  0  1 -1]
```

is_ambient()

Return `True` since this module is an ambient module.

EXAMPLES:

```
sage: A = QQ^5; A.is_ambient()
True
sage: A = (QQ^5).span([[1,2,3,4,5]]); A.is_ambient()
False
```

linear_combination_of_basis(*v*)

Return the linear combination of the basis for self obtained from the elements of the list v.

EXAMPLES:

```
sage: V = span([[1,2,3], [4,5,6]], ZZ)
sage: V
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
sage: V.linear_combination_of_basis([1,1])
(1, 5, 9)
```

random_element(*prob=1.0, **kwds*)

Returns a random element of self.

INPUT:

- `prob` - float; probability that given coefficient is nonzero.
- `**kwds` - passed on to random_element function of base ring.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.random_element()
(-1, 2, 1)
sage: M.random_element()
(-95, -1, -2)
sage: M.random_element()
(-12, 0, 0)

sage: M = FreeModule(ZZ, 16)
sage: M.random_element()
(1, -1, 1, -1, -2, -1, 4, -4, -6, 5, 0, 0, -2, 0, 1, -4)
sage: M.random_element(prob=0.3)
(0, 0, 0, 0, 0, 0, 0, -6, 1, -1, 1, 0, 1, 0, 0, 0)
```

class FreeModule_ambient_domain(*base_ring, rank, sparse=False*)

Ambient free module over an integral domain.

ambient_vector_space()

Returns the ambient vector space, which is this free module tensored with its fraction field.

EXAMPLES:

```
sage: M = ZZ^3;
sage: V = M.ambient_vector_space(); V
Vector space of dimension 3 over Rational Field
```

If an inner product on the module is specified, then this is preserved on the ambient vector space.

```
sage: N = FreeModule(ZZ,4,inner_product_matrix=1)
sage: U = N.ambient_vector_space()
sage: U
Ambient quadratic space of dimension 4 over Rational Field
Inner product matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: P = N.submodule_with_basis([[1,-1,0,0],[0,1,-1,0],[0,0,1,-1]])
sage: P.gram_matrix()
[ 2 -1  0]
[-1  2 -1]
[ 0 -1  2]
sage: U == N.ambient_vector_space()
True
sage: U == V
False
```

base_field()

Return the fraction field of the base ring of self.

EXAMPLES:

```
sage: M = ZZ^3;  M.base_field()
Rational Field
sage: M = PolynomialRing(GF(5),'x')^3;  M.base_field()
Fraction Field of Univariate Polynomial Ring in x over Finite Field of size 5
```

coordinate_vector(*v, check=True*)

Write v in terms of the standard basis for self and return the resulting coefficients in a vector over the fraction field of the base ring.

INPUT:

- v - vector
- check - bool (default: True); if True, also verify that v is really in self.

OUTPUT: list

Returns a vector c such that if B is the basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raises an ArithmeticError exception.

EXAMPLES:

```
sage: V = ZZ^3
sage: v = V.coordinate_vector([1,5,9]); v
(1, 5, 9)
sage: v.parent()
Vector space of dimension 3 over Rational Field
```

vector_space(*base_field=None*)

Returns the vector space obtained from self by tensoring with the fraction field of the base ring and extending to the field.

EXAMPLES:

```
sage: M = ZZ^3;  M.vector_space()
Vector space of dimension 3 over Rational Field
```

class FreeModule_ambient_field(*base_field, dimension, sparse=False*)

ambient_vector_space()

Returns self as the ambient vector space.

EXAMPLES:

```
sage: M = QQ^3
sage: M.ambient_vector_space()
Vector space of dimension 3 over Rational Field
```

base_field()

Returns the base field of this vector space.

EXAMPLES:

```
sage: M = QQ^3
sage: M.base_field()
Rational Field
```

class FreeModule_ambient_pid(*base_ring, rank, sparse=False*)

Ambient free module over a principal ideal domain.

class FreeModule_generic(*base_ring, rank, degree, sparse=False*)

Base class for all free modules.

ambient_module()

Return the ambient module associated to this module.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: M = FreeModule(R,2)
sage: M.ambient_module()
Ambient free module of rank 2 over the integral domain Multivariate Polynomial Ring in x, y
```

```
sage: V = FreeModule(QQ, 4).span([[1,2,3,4], [1,0,0,0]]); V
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[  1   0   0   0]
[  0   1 3/2   2]
sage: V.ambient_module()
Vector space of dimension 4 over Rational Field
```

base_extend(*R*)

Return the base extension of self to R. This is the same as `self.change_ring(R)` except that a TypeError is raised if there is no canonical coerce map from the base ring of self to R.

INPUT:

- R - ring

EXAMPLES:

```
sage: V = ZZ^7
sage: V.base_extend(QQ)
Vector space of dimension 7 over Rational Field
```

basis()

Return the basis of this module.

EXAMPLES:

```
sage: FreeModule(Integers(12),3).basis()
[
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
```

basis_matrix()

Return the matrix whose rows are the basis for this free module.

EXAMPLES:

```
sage: FreeModule(Integers(12),3).basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]

sage: M = FreeModule(GF(7),3).span([[2,3,4],[1,1,1]]); M
Vector space of degree 3 and dimension 2 over Finite Field of size 7
Basis matrix:
[1 0 6]
[0 1 2]
sage: M.basis_matrix()
[1 0 6]
[0 1 2]

sage: M = FreeModule(GF(7),3).span_of_basis([[2,3,4],[1,1,1]]);
sage: M.basis_matrix()
[2 3 4]
[1 1 1]
```

category()

Return the category to which this free module belongs. This is the category of all free modules over the base ring.

EXAMPLES:

```
sage: FreeModule(GF(7),3).category()
Category of vector spaces over Finite Field of size 7
```

construction()

The construction functor and base ring for self.

EXAMPLES:

```
sage: R = PolynomialRing(QQ,3,'x')
sage: V = R^5
sage: V.construction()
(VectorFunctor, Multivariate Polynomial Ring in x0, x1, x2 over Rational Field)
```

coordinate_module(V)

Suppose V is a submodule of self (or a module comeasurable with self), and that self is a free module over R of rank n. Let ϕ be the map from self to R^n that sends the basis vectors of self in order to the standard basis of R^n. This function returns the image $\phi(V)$.

Warning: If there is no integer d such that dV is a submodule of self, then this function will give total nonsense.

EXAMPLES:

We illustrate this function with some $\mathbf{Z}$-submodules of $\mathbf{Q}^3$.

```
sage: V = (ZZ^3).span([[1/2,3,5], [0,1,-3]])
sage: W = (ZZ^3).span([[1/2,4,2]])
sage: V.coordinate_module(W)
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[1 4]
sage: V.0 + 4*V.1
(1/2, 4, 2)
```

In this example, the coordinate module isn't even in $\mathbf{Z}^3$.

```
sage: W = (ZZ^3).span([[1/4,2,1]])
sage: V.coordinate_module(W)
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[1/2   2]
```

The following more elaborate example illustrates using this function to write a submodule in terms of integral cuspidal modular symbols:

```
sage: M = ModularSymbols(54)
sage: S = M.cuspidal_subspace()
sage: K = S.integral_structure(); K
Free module of degree 19 and rank 8 over Integer Ring
Echelon basis matrix:
[ 0  1  0  0 -1  0  0  0  0  0  0  0  0  0  0  0  0  0]
...
sage: L = M[0].integral_structure(); L
Free module of degree 19 and rank 2 over Integer Ring
Echelon basis matrix:
[ 0  1  1  0 -2  1 -1  1 -1 -2  2  0  0  0  0  0  0  0  0]
[ 0  0  3  0 -3  2 -1  2 -1 -4  2 -1 -2  1  2  0  0 -1  1]
sage: K.coordinate_module(L)
Free module of degree 8 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  1  1 -1  1 -1  0  0]
```

```
[ 0  3  2 -1  2 -1 -1 -2]
sage: K.coordinate_module(L).basis_matrix() * K.basis_matrix()
[ 0  1  1  0 -2  1 -1  1 -1 -2  2  0  0  0  0  0  0  0]
[ 0  0  3  0 -3  2 -1  2 -1 -4  2 -1 -2  1  2  0  0 -1  1]
```

coordinate_vector(*v, check=True*)

Return the vector whose coefficients give v as a linear combination of the basis for self.

INPUT:

- v - vector
- check - bool (default: True); if True, also verify that v is really in self.

OUTPUT: list

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2); M0,M1=M.gens()
sage: W = M.submodule([M0 + M1, M0 - 2*M1])
sage: W.coordinate_vector(2*M0 - M1)
(2, -1)
```

coordinates(*v, check=True*)

Write v in terms of the basis for self.

INPUT:

- v - vector
- check - bool (default: True); if True, also verify that v is really in self.

OUTPUT: list

Returns a list c such that if B is the basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raises an ArithmeticError exception.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2); M0,M1=M.gens()
sage: W = M.submodule([M0 + M1, M0 - 2*M1])
sage: W.coordinates(2*M0-M1)
[2, -1]
```

degree()

Return the degree of this free module. This is the dimension of the ambient vector space in which it is embedded.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 10)
sage: W = M.submodule([M.gen(0), 2*M.gen(3) - M.gen(0), M.gen(0) + M.gen(3)])
sage: W.degree()
10
sage: W.rank()
2
```

dense_module()

Return corresponding dense module.

EXAMPLES:

We first illustrate conversion with ambient spaces:

```
sage: M = FreeModule(QQ,3)
sage: S = FreeModule(QQ,3, sparse=True)
```

```
sage: M.sparse_module()
Sparse vector space of dimension 3 over Rational Field
sage: S.dense_module()
Vector space of dimension 3 over Rational Field
sage: M.sparse_module() == S
True
sage: S.dense_module() == M
True
sage: M.dense_module() == M
True
sage: S.sparse_module() == S
True
```

Next we create a subspace:

```
sage: M = FreeModule(QQ,3, sparse=True)
sage: V = M.span([ [1,2,3] ] ); V
Sparse vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
sage: V.sparse_module()
Sparse vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
```

dimension()

Return the dimension of this free module.

EXAMPLES:

```
sage: M = FreeModule(FiniteField(19), 100)
sage: W = M.submodule([M.gen(50)])
sage: W.dimension()
1
```

direct_sum(*other*)

Return the direct sum of self and other as a free module.

EXAMPLES:

```
sage: V = (ZZ^3).span([[1/2,3,5], [0,1,-3]]); V
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1/2   0  14]
[  0   1  -3]
sage: W = (ZZ^3).span([[1/2,4,2]]); W
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1/2   4   2]
sage: V.direct_sum(W)
Free module of degree 6 and rank 3 over Integer Ring
Echelon basis matrix:
[1/2   0  14   0   0   0]
[  0   1  -3   0   0   0]
[  0   0   0 1/2   4   2]
```

discriminant()

Return the discriminant of this free module.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.discriminant()
1
sage: W = M.span([[1,2,3]])
sage: W.discriminant()
14
sage: W2 = M.span([[1,2,3], [1,1,1]])
sage: W2.discriminant()
6
```

echelonized_basis_matrix()

The echelonized basis matrix (not implemented for this module).

This example works because M is an ambient module. Submodule creation should exist for generic modules.

EXAMPLES:

```
sage: R = IntegerModRing(12)
sage: S.<x,y> = R[]
sage: M = FreeModule(S,3)
sage: M.echelonized_basis_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

TESTS:

```
sage: from sage.modules.free_module import FreeModule_generic
sage: FreeModule_generic.echelonized_basis_matrix(M)
...
NotImplementedError
```

element_class()

The class of elements for this free module.

EXAMPLES:

```
sage: M = FreeModule(ZZ,20,sparse=False)
sage: x = M.random_element()
sage: type(x)
<type 'sage.modules.vector_integer_dense.Vector_integer_dense'>
sage: M.element_class()
<type 'sage.modules.vector_integer_dense.Vector_integer_dense'>
sage: N = FreeModule(ZZ,20,sparse=True)
sage: y = N.random_element()
sage: type(y)
<type 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
sage: N.element_class()
<type 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
```

free_module()

Return this free module. (This is used by the `FreeModule` functor, and simply returns self.)

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.free_module()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

gen(*i=0*)

Return ith generator for self, where i is between 0 and rank-1, inclusive.

INPUT:

- i - an integer

OUTPUT: i-th basis vector for self.

EXAMPLES:

```
sage: n = 5
sage: V = QQ^n
sage: B = [ V.gen(i) for i in range(n) ]
sage: B
[(1, 0, 0, 0, 0),
(0, 1, 0, 0, 0),
(0, 0, 1, 0, 0),
(0, 0, 0, 1, 0),
(0, 0, 0, 0, 1)]
sage: V.gens() == tuple(B)
True
```

TESTS:

```
sage: (QQ^3).gen(4/3)
...
TypeError: rational is not an integer
```

gram_matrix()

Return the gram matrix associated to this free module, defined to be G = B*A*B.transpose(), where A is the inner product matrix (induced from the ambient space), and B the basis matrix.

EXAMPLES:

```
sage: V = VectorSpace(QQ,4)
sage: u = V([1/2,1/2,1/2,1/2])
sage: v = V([0,1,1,0])
sage: w = V([0,0,1,1])
sage: M = span([u,v,w], ZZ)
sage: M.inner_product_matrix() == V.inner_product_matrix()
True
sage: L = M.submodule_with_basis([u,v,w])
sage: L.inner_product_matrix() == M.inner_product_matrix()
True
sage: L.gram_matrix()
[1 1 1]
[1 2 1]
[1 1 2]
```

has_user_basis()

Return True if the basis of this free module is specified by the user, as opposed to being the default echelon form.

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.subspace([[2,'1/2', 1]])
sage: W.has_user_basis()
False
sage: W = V.subspace_with_basis([[2,'1/2',1]])
sage: W.has_user_basis()
True
```

inner_product_matrix()

Return the default identity inner product matrix associated to this module.

By definition this is the inner product matrix of the ambient space, hence may be of degree greater than the rank of the module.

TODO: Differentiate the image ring of the inner product from the base ring of the module and/or ambient space. E.g. On an integral module over ZZ the inner product pairing could naturally take values in ZZ, QQ, RR, or CC.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.inner_product_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

is_ambient()

Returns False since this is not an ambient free module.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3).span([[1,2,3]]); M
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1 2 3]
sage: M.is_ambient()
False
sage: M = (ZZ^2).span([[1,0], [0,1]])
sage: M
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 0]
[0 1]
sage: M.is_ambient()
False
sage: M == M.ambient_module()
True
```

is_dense()

Return `True` if the underlying representation of this module uses dense vectors, and False otherwise.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).is_dense()
True
sage: FreeModule(ZZ, 2, sparse=True).is_dense()
False
```

is_finite()

Returns True if the underlying set of this free module is finite.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).is_finite()
False
sage: FreeModule(Integers(8), 2).is_finite()
True
sage: FreeModule(ZZ, 0).is_finite()
True
```

is_full()

Return `True` if the rank of this module equals its degree.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).is_full()
True
sage: M = FreeModule(ZZ, 2).span([[1,2]])
```

```
sage: M.is_full()
False
```

is_sparse()

Return `True` if the underlying representation of this module uses sparse vectors, and False otherwise.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).is_sparse()
False
sage: FreeModule(ZZ, 2, sparse=True).is_sparse()
True
```

is_submodule(*other*)

Return True if self is a submodule of other.

EXAMPLES:

```
sage: M = FreeModule(ZZ,3)
sage: V = M.ambient_vector_space()
sage: X = V.span([[1/2,1/2,0],[1/2,0,1/2]], ZZ)
sage: Y = V.span([[1,1,1]], ZZ)
sage: N = X + Y
sage: M.is_submodule(X)
False
sage: M.is_submodule(Y)
False
sage: Y.is_submodule(M)
True
sage: N.is_submodule(M)
False
sage: M.is_submodule(N)
True
```

Since basis() is not implemented in general, submodule testing does not work for all PID's. However, trivial cases are already used (and useful) for coercion, e.g.

```
sage: QQ(1/2) * vector(ZZ['x']['y'],[1,2,3,4])
(1/2, 1, 3/2, 2)
sage: vector(ZZ['x']['y'],[1,2,3,4]) * QQ(1/2)
(1/2, 1, 3/2, 2)
```

matrix()

Return the basis matrix of this module, which is the matrix whose rows are a basis for this module.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2)
sage: M.matrix()
[1 0]
[0 1]
sage: M.submodule([M.gen(0) + M.gen(1), M.gen(0) - 2*M.gen(1)]).matrix()
[1 1]
[0 3]
```

ngens()

Returns the number of basis elements of this free module.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).ngens()
2
sage: FreeModule(ZZ, 0).ngens()
0
```

```
sage: FreeModule(ZZ, 2).span([[1,1]]).ngens()
1
```

nonembedded_free_module()

Returns an ambient free module that is isomorphic to this free module.

Thus if this free module is of rank n over a ring R, then this function returns R^n, as an ambient free module.

EXAMPLES:

```
sage: FreeModule(ZZ, 2).span([[1,1]]).nonembedded_free_module()
Ambient free module of rank 1 over the principal ideal domain Integer Ring
```

random_element(*prob=1.0, **kwds*)

Returns a random element of self.

INPUT:

- `prob` - float; probability that given coefficient is nonzero.
- `**kwds` - passed on to random_element function of base ring.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2).span([[1,1]])
sage: M.random_element()
(-1, -1)
sage: M.random_element()
(2, 2)
sage: M.random_element()
(1, 1)
```

rank()

Return the rank of this free module.

EXAMPLES:

```
sage: FreeModule(Integers(6), 10000000).rank()
10000000
sage: FreeModule(ZZ, 2).span([[1,1], [2,2], [3,4]]).rank()
2
```

sparse_module()

Return the corresponding sparse module with the same defining data.

EXAMPLES:

We first illustrate conversion with ambient spaces:

```
sage: M = FreeModule(Integers(8),3)
sage: S = FreeModule(Integers(8),3, sparse=True)
sage: M.sparse_module()
Ambient sparse free module of rank 3 over Ring of integers modulo 8
sage: S.dense_module()
Ambient free module of rank 3 over Ring of integers modulo 8
sage: M.sparse_module() is S
True
sage: S.dense_module() is M
True
sage: M.dense_module() is M
True
sage: S.sparse_module() is S
True
```

Next we convert a subspace:

```
sage: M = FreeModule(QQ,3)
sage: V = M.span([ [1,2,3] ] ); V
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
sage: V.sparse_module()
Sparse vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
```

uses_ambient_inner_product()

Return `True` if the inner product on this module is the one induced by the ambient inner product.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2)
sage: W = M.submodule([[1,2]])
sage: W.uses_ambient_inner_product()
True
sage: W.inner_product_matrix()
[1 0]
[0 1]

sage: W.gram_matrix()
[5]
```

zero_vector()

Returns the zero vector in this free module.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 2)
sage: M.zero_vector()
(0, 0)
sage: M(0)
(0, 0)
sage: M.span([[1,1]]).zero_vector()
(0, 0)
sage: M.zero_submodule().zero_vector()
(0, 0)
```

class **FreeModule_generic_field**(*base_field, dimension, degree, sparse=False*)

Base class for all free modules over fields.

category()

Return the category to which this vector space belongs.

EXAMPLES:

```
sage: V = QQ^4; V.category()
Category of vector spaces over Rational Field
sage: V = GF(5)**20; V.category()
Category of vector spaces over Finite Field of size 5
```

echelonized_basis_matrix()

Return basis matrix for self in row echelon form.

EXAMPLES:

```
sage: V = FreeModule(QQ, 3).span_of_basis([[1,2,3],[4,5,6]])
sage: V.basis_matrix()
[1 2 3]
[4 5 6]
```

```
sage: V.echelonized_basis_matrix()
[ 1  0 -1]
[ 0  1  2]
```

intersection(*other*)

Return the intersection of self and other, which must be R-submodules of a common ambient vector space.

EXAMPLES:

```
sage: V  = VectorSpace(QQ,3)
sage: W1 = V.submodule([V.gen(0), V.gen(0) + V.gen(1)])
sage: W2 = V.submodule([V.gen(1), V.gen(2)])
sage: W1.intersection(W2)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 1 0]
sage: W2.intersection(W1)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 1 0]
sage: V.intersection(W1)
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
sage: W1.intersection(V)
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
sage: Z = V.submodule([])
sage: W1.intersection(Z)
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
```

is_subspace(*other*)

True if this vector space is a subspace of other.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: W = V.subspace([V.gen(0), V.gen(0) + V.gen(1)])
sage: W2 = V.subspace([V.gen(1)])
sage: W.is_subspace(V)
True
sage: W2.is_subspace(V)
True
sage: W.is_subspace(W2)
False
sage: W2.is_subspace(W)
True
```

quotient(*sub, check=True*)

Return the quotient of self by the given subspace sub.

INPUT:

- `sub` - a submodule of self, or something that can be turned into one via self.submodule(sub).
- `check` - (default: True) whether or not to check that sub is a submodule.

EXAMPLES:

```
sage: A = QQ^3; V = A.span([[1,2,3], [4,5,6]])
sage: Q = V.quotient( [V.0 + V.1] ); Q
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
W: Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 1 1]
sage: Q(V.0 + V.1)
(0)
```

quotient_abstract(*sub, check=True*)

Returns an ambient free module isomorphic to the quotient space of self modulo sub, together with maps from self to the quotient, and a lifting map in the other direction.

Use `self.quotient(sub)` to obtain the quotient module as an object equipped with natural maps in both directions, and a canonical coercion.

INPUT:

- `sub` - a submodule of self, or something that can be turned into one via self.submodule(sub).
- `check` - (default: True) whether or not to check that sub is a submodule.

OUTPUT:

- `U` - the quotient as an abstract *ambient* free module
- `pi` - projection map to the quotient
- `lift` - lifting map back from quotient

EXAMPLES:

```
sage: V = GF(19)^3
sage: W = V.span_of_basis([ [1,2,3], [1,0,1] ])
sage: U,pi,lift = V.quotient_abstract(W)
sage: pi(V.2)
(18)
sage: pi(V.0)
(1)
sage: pi(V.0 + V.2)
(0)
```

Another example involving a quotient of one subspace by another.

```
sage: A = matrix(QQ,4,4,[0,1,0,0, 0,0,1,0, 0,0,0,1, 0,0,0,0])
sage: V = (A^3).kernel()
sage: W = A.kernel()
sage: U, pi, lift = V.quotient_abstract(W)
sage: [pi(v) == 0 for v in W.gens()]
[True]
sage: [pi(lift(b)) == b for b in U.basis()]
[True, True]
```

scale(*other*)

Return the product of self by the number other, which is the module spanned by other times each basis vector. Since self is a vector space this product equals self if other is nonzero, and is the zero vector space if other is 0.

EXAMPLES:

```
sage: V = QQ^4
sage: V.scale(5)
```

```
Vector space of dimension 4 over Rational Field
sage: V.scale(0)
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]

sage: W = V.span([[1,1,1,1]])
sage: W.scale(2)
Vector space of degree 4 and dimension 1 over Rational Field
Basis matrix:
[1 1 1 1]
sage: W.scale(0)
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]

sage: V = QQ^4; V
Vector space of dimension 4 over Rational Field
sage: V.scale(3)
Vector space of dimension 4 over Rational Field
sage: V.scale(0)
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

span(*gens, base_ring=None, check=True, already_echelonized=False*)

Return the K-span of the given list of gens, where K is the base field of self or the user-specified base_ring. Note that this span is a subspace of the ambient vector space, but need not be a subspace of self.

INPUT:

- `gens` - list of vectors
- `check` - bool (default: True): whether or not to coerce entries of gens into base field
- `already_echelonized` - bool (default: False): set this if you know the gens are already in echelon form

EXAMPLES:

```
sage: V = VectorSpace(GF(7), 3)
sage: W = V.subspace([[2,3,4]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
sage: W.span([[1,1,1]])
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 1 1]
```

TESTS:

```
sage: V = FreeModule(RDF,3)
sage: W = V.submodule([V.gen(0)])
sage: W.span([V.gen(1)], base_ring=GF(7))
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[0 1 0]
sage: v = V((1, pi, e)); v
(1.0, 3.14159265359, 2.71828182846)
sage: W.span([v], base_ring=GF(7))
...
```

```
ValueError: Argument gens (= [(1.0, 3.14159265359, 2.71828182846)]) is not compatible with b
sage: W = V.submodule([v])
sage: W.span([V.gen(2)], base_ring=GF(7))
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[0 0 1]
```

span_of_basis(*basis, base_ring=None, check=True, already_echelonized=False*)

Return the free K-module with the given basis, where K is the base field of self or user specified base_ring.

Note that this span is a subspace of the ambient vector space, but need not be a suspace of self.

INPUT:

- `basis` - list of vectors
- `check` - bool (default: True): whether or not to coerce entries of gens into base field
- `already_echelonized` - bool (default: False): set this if you know the gens are already in echelon form

EXAMPLES:

```
sage: V = VectorSpace(GF(7), 3)
sage: W = V.subspace([[2,3,4]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
sage: W.span_of_basis([[2,2,2], [3,3,0]])
Vector space of degree 3 and dimension 2 over Finite Field of size 7
User basis matrix:
[2 2 2]
[3 3 0]
```

The basis vectors must be linearly independent or an ArithmeticError exception is raised.

```
sage: W.span_of_basis([[2,2,2], [3,3,3]])
...
ValueError: The given basis vectors must be linearly independent.
```

subspace(*gens, check=True, already_echelonized=False*)

Return the subspace of self spanned by the elements of gens.

INPUT:

- `gens` - list of vectors
- `check` - bool (default: True) verify that gens are all in self.
- `already_echelonized` - bool (default: False) set to True if you know the gens are in Echelon form.

EXAMPLES:

First we create a 1-dimensional vector subspace of an ambient 3-dimensional space over the finite field of order 7.

```
sage: V = VectorSpace(GF(7), 3)
sage: W = V.subspace([[2,3,4]]); W
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 5 2]
```

Next we create an invalid subspace, but it's allowed since `check=False`. This is just equivalent to computing the span of the element.

```
sage: W.subspace([[1,1,0]], check=False)
Vector space of degree 3 and dimension 1 over Finite Field of size 7
```

```
Basis matrix:
[1 1 0]
```

With `check=True` (the default) the mistake is correctly detected and reported with an `ArithmeticError` exception.

```
sage: W.subspace([[1,1,0]], check=True)
...
ArithmeticError: Argument gens (= [[1, 1, 0]]) does not generate a submodule of self.
```

subspace_with_basis(*gens, check=True, already_echelonized=False*)

Same as `self.submodule_with_basis(...)`.

EXAMPLES:

We create a subspace with a user-defined basis.

```
sage: V = VectorSpace(GF(7), 3)
sage: W = V.subspace_with_basis([[2,2,2], [1,2,3]]); W
Vector space of degree 3 and dimension 2 over Finite Field of size 7
User basis matrix:
[2 2 2]
[1 2 3]
```

We then create a subspace of the subspace with user-defined basis.

```
sage: W1 = W.subspace_with_basis([[3,4,5]]); W1
Vector space of degree 3 and dimension 1 over Finite Field of size 7
User basis matrix:
[3 4 5]
```

Notice how the basis for the same subspace is different if we merely use the `subspace` command.

```
sage: W2 = W.subspace([[3,4,5]]); W2
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 6 4]
```

Nonetheless the two subspaces are equal (as mathematical objects):

```
sage: W1 == W2
True
```

subspaces(*dim*)

Iterate over all subspaces of dimension dim.

INPUT:

- `dim` - int, dimension of subspaces to be generated

EXAMPLE:

```
sage: V = VectorSpace(GF(3), 5)
sage: len(list(V.subspaces(0)))
1
sage: len(list(V.subspaces(1)))
121
sage: len(list(V.subspaces(2)))
1210
sage: len(list(V.subspaces(3)))
1210
sage: len(list(V.subspaces(4)))
121
sage: len(list(V.subspaces(5)))
1
```

```
sage: V = VectorSpace(GF(3), 5)
sage: V = V.subspace([V([1,1,0,0,0]),V([0,0,1,1,0])])
sage: list(V.subspaces(1))
[Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[1 1 0 0 0],
 Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[1 1 1 1 0],
 Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[1 1 2 2 0],
 Vector space of degree 5 and dimension 1 over Finite Field of size 3
Basis matrix:
[0 0 1 1 0]]
```

vector_space(*base_field=None*)

Return the vector space associated to self. Since self is a vector space this function simply returns self, unless the base field is different.

EXAMPLES:

```
sage: V = span([[1,2,3]],QQ); V
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
sage: V.vector_space()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[1 2 3]
```

zero_submodule()

Return the zero submodule of self.

EXAMPLES:

```
sage: (QQ^4).zero_submodule()
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

zero_subspace()

Return the zero subspace of self.

EXAMPLES:

```
sage: (QQ^4).zero_subspace()
Vector space of degree 4 and dimension 0 over Rational Field
Basis matrix:
[]
```

class **FreeModule_generic_pid**(*base_ring, rank, degree, sparse=False*)

Base class for all free modules over a PID.

base_field()

Return the base field, which is the fraction field of the base ring of this module.

EXAMPLES:

```
sage: FreeModule(GF(3), 2).base_field()
Finite Field of size 3
sage: FreeModule(ZZ, 2).base_field()
Rational Field
```

```
sage: FreeModule(PolynomialRing(GF(7),'x'), 2).base_field()
Fraction Field of Univariate Polynomial Ring in x over Finite Field of size 7
```

basis_matrix()

Return the matrix whose rows are the basis for this free module.

EXAMPLES:

```
sage: M = FreeModule(QQ,2).span_of_basis([[1,-1],[1,0]]); M
Vector space of degree 2 and dimension 2 over Rational Field
User basis matrix:
[ 1 -1]
[ 1  0]
sage: M.basis_matrix()
[ 1 -1]
[ 1  0]
```

See #3699: sage: K = FreeModule(ZZ, 2000) sage: I = K.basis_matrix()

denominator()

The denominator of the basis matrix of self (i.e. the LCM of the coordinate entries with respect to the basis of the ambient space).

EXAMPLES:

```
sage: V = QQ^3
sage: L = V.span([[1,1/2,1/3], [-1/5,2/3,3]],ZZ)
sage: L
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1/5 19/6 37/3]
[   0 23/6 46/3]
sage: L.denominator()
30
```

index_in(*other*)

Return the lattice index [other:self] of self in other, as an element of the base field. When self is contained in other, the lattice index is the usual index. If the index is infinite, then this function returns infinity.

EXAMPLES:

```
sage: L1 = span([[1,2]], ZZ)
sage: L2 = span([[3,6]], ZZ)
sage: L2.index_in(L1)
3
```

Note that the free modules being compared need not be integral.

```
sage: L1 = span([['1/2','1/3'], [4,5]], ZZ)
sage: L2 = span([[1,2], [3,4]], ZZ)
sage: L2.index_in(L1)
12/7
sage: L1.index_in(L2)
7/12
sage: L1.discriminant() / L2.discriminant()
49/144
```

The index of a lattice of infinite index is infinite.

```
sage: L1 = FreeModule(ZZ, 2)
sage: L2 = span([[1,2]], ZZ)
sage: L2.index_in(L1)
+Infinity
```

index_in_saturation()

Return the index of this module in its saturation, i.e., its intersection with R^n.

EXAMPLES:

```
sage: W = span([[2,4,6]], ZZ)
sage: W.index_in_saturation()
2
sage: W = span([[1/2,1/3]], ZZ)
sage: W.index_in_saturation()
1/6
```

intersection(*other*)

Return the intersection of self and other.

EXAMPLES:

We intersect two submodules one of which is clearly contained in the other.

```
sage: A = ZZ^2
sage: M1 = A.span([[1,1]])
sage: M2 = A.span([[3,3]])
sage: M1.intersection(M2)
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[3 3]
```

We intersection two submodules of $\mathbf{Z}^3$ of rank 2, whose intersection has rank 1.

```
sage: A = ZZ^3
sage: M1 = A.span([[1,1,1], [1,2,3]])
sage: M2 = A.span([[2,2,2], [1,0,0]])
sage: M1.intersection(M2)
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[2 2 2]
```

We compute an intersection of two $\mathbf{Z}$-modules that are not submodules of $\mathbf{Z}^2$.

```
sage: A = ZZ^2
sage: M1 = A.span([[1,2]]).scale(1/6)
sage: M2 = A.span([[1,2]]).scale(1/15)
sage: M1.intersection(M2)
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[1/3 2/3]
```

We intersect a $\mathbf{Z}$-module with a $\mathbf{Q}$-vector space.

```
sage: A = ZZ^3
sage: L = ZZ^3
sage: V = QQ^3
sage: W = L.span([[1/2,0,1/2]])
sage: K = V.span([[1,0,1], [0,0,1]])
sage: W.intersection(K)
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1/2   0 1/2]
sage: K.intersection(W)
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1/2   0 1/2]
```

is_submodule(*other*)

True if this module is a submodule of other.

EXAMPLES:

```
sage: M = FreeModule(ZZ,2)
sage: M.is_submodule(M)
True
sage: N = M.scale(2)
sage: N.is_submodule(M)
True
sage: M.is_submodule(N)
False
sage: N = M.scale(1/2)
sage: N.is_submodule(M)
False
sage: M.is_submodule(N)
True
```

saturation()

Return the saturated submodule of R^n that spans the same vector space as self.

EXAMPLES:

We create a 1-dimensional lattice that is obviously not saturated and saturate it.

```
sage: L = span([[9,9,6]], ZZ); L
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[9 9 6]
sage: L.saturation()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[3 3 2]
```

We create a lattice spanned by two vectors, and saturate. Computation of discriminants shows that the index of lattice in its saturation is 3, which is a prime of congruence between the two generating vectors.

```
sage: L = span([[1,2,3], [4,5,6]], ZZ)
sage: L.saturation()
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: L.discriminant()
54
sage: L.saturation().discriminant()
6
```

Notice that the saturation of a non-integral lattice L is defined, but the result is integral hence does not contain L:

```
sage: L = span([['1/2',1,3]], ZZ)
sage: L.saturation()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[1 2 6]
```

scale(*other*)

Return the product of this module by the number other, which is the module spanned by other times each basis vector.

EXAMPLES:

```
sage: M = FreeModule(ZZ, 3)
sage: M.scale(2)
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[2 0 0]
[0 2 0]
[0 0 2]

sage: a = QQ('1/3')
sage: M.scale(a)
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1/3   0   0]
[  0 1/3   0]
[  0   0 1/3]
```

span(*gens, base_ring=None, check=True, already_echelonized=False*)

Return the R-span of the given list of gens, where R = base_ring. The default R is the base ring of self. Note that this span need not be a submodule of self, nor even of the ambient space. It must, however, be contained in the ambient vector space, i.e., the ambient space tensored with the fraction field of R.

EXAMPLES:

```
sage: V = FreeModule(ZZ,3)
sage: W = V.submodule([V.gen(0)])
sage: W.span([V.gen(1)])
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1 0]
sage: W.submodule([V.gen(1)])
...
ArithmeticError: Argument gens (= [(0, 1, 0)]) does not generate a submodule of self.
```

span_of_basis(*basis, base_ring=None, check=True, already_echelonized=False*)

Return the free R-module with the given basis, where R is the base ring of self or user specified base_ring.

Note that this R-module need not be a submodule of self, nor even of the ambient space. It must, however, be contained in the ambient vector space, i.e., the ambient space tensored with the fraction field of R.

EXAMPLES:

```
sage: M = FreeModule(ZZ,3)
sage: W = M.span_of_basis([M([1,2,3])])
```

Next we create two free $\mathbf{Z}$-modules, neither of which is a submodule of W.

```
sage: W.span_of_basis([M([2,4,0])])
Free module of degree 3 and rank 1 over Integer Ring
User basis matrix:
[2 4 0]
```

The following module isn't in the ambient module $\mathbf{Z}^3$ but is contained in the ambient vector space $\mathbf{Q}^3$:

```
sage: V = M.ambient_vector_space()
sage: W.span_of_basis([ V([1/5,2/5,0]), V([1/7,1/7,0]) ])
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1/5 2/5   0]
[1/7 1/7   0]
```

Of course the input basis vectors must be linearly independent.

```
sage: W.span_of_basis([ [1,2,0], [2,4,0] ])
...
ValueError: The given basis vectors must be linearly independent.
```

submodule(*gens, check=True, already_echelonized=False*)

Create the R-submodule of the ambient vector space with given generators, where R is the base ring of self.

INPUT:

- `gens` - a list of free module elements or a free module
- `check` - (default: True) whether or not to verify that the gens are in self.

OUTPUT:

- `FreeModule` - the submodule spanned by the vectors in the list gens. The basis for the subspace is always put in reduced row echelon form.

EXAMPLES:

We create a submodule of $\mathbf{Z}^3$:

```
sage: M = FreeModule(ZZ, 3)
sage: B = M.basis()
sage: W = M.submodule([B[0]+B[1], 2*B[1]-B[2]])
sage: W
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  1  0]
[ 0  2 -1]
```

We create a submodule of a submodule.

```
sage: W.submodule([3*B[0] + 3*B[1]])
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[3 3 0]
```

We try to create a submodule that isn't really a submodule, which results in an ArithmeticError exception:

```
sage: W.submodule([B[0] - B[1]])
...
ArithmeticError: Argument gens (= [(1, -1, 0)]) does not generate a submodule of self.
```

Next we try to create a submodule of a free module over the principal ideal domain $\mathbf{Q}[x]$ (general HNF needed):

```
sage: R = PolynomialRing(QQ, 'x'); x = R.gen()
sage: M = FreeModule(R, 3)
sage: B = M.basis()
sage: W = M.submodule([x*B[0], 2*B[1]- x*B[2]])
...
NotImplementedError: echelon form over Univariate Polynomial Ring in x over Rational Field r
```

submodule_with_basis(*basis, check=True, already_echelonized=False*)

Create the R-submodule of the ambient vector space with given basis, where R is the base ring of self.

INPUT:

- `basis` - a list of linearly independent vectors
- `check` - whether or not to verify that each gen is in the ambient vector space

OUTPUT:

- `FreeModule` - the R-submodule with given basis

EXAMPLES:

First we create a submodule of $\mathbf{Z}^3$:

```
sage: M = FreeModule(ZZ, 3)
sage: B = M.basis()
sage: N = M.submodule_with_basis([B[0]+B[1], 2*B[1]-B[2]])
sage: N
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 1  1  0]
[ 0  2 -1]
```

A list of vectors in the ambient vector space may fail to generate a submodule.

```
sage: V = M.ambient_vector_space()
sage: X = M.submodule_with_basis([ V(B[0]+B[1])/2, V(B[1]-B[2])/2])
...
ArithmeticError: The given basis does not generate a submodule of self.
```

However, we can still determine the R-span of vectors in the ambient space, or over-ride the submodule check by setting check to False.

```
sage: X = V.span([ V(B[0]+B[1])/2, V(B[1]-B[2])/2 ], ZZ)
sage: X
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1/2    0   1/2]
[   0  1/2  -1/2]
sage: Y = M.submodule([ V(B[0]+B[1])/2, V(B[1]-B[2])/2 ], check=False)
sage: X == Y
True
```

Next we try to create a submodule of a free module over the principal ideal domain $\mathbf{Q}[x]$ (general HNF needed):

```
sage: R = PolynomialRing(QQ, 'x'); x = R.gen()
sage: M = FreeModule(R, 3)
sage: B = M.basis()
sage: W = M.submodule_with_basis([x*B[0], 2*B[1]- x*B[2]])
...
NotImplementedError: echelon form over Univariate Polynomial Ring in x over Rational Field r
```

vector_space_span(*gens, check=True*)

Create the vector subspace of the ambient vector space with given generators.

INPUT:

- `gens` - a list of vector in self
- `check` - whether or not to verify that each gen is in the ambient vector space

OUTPUT: a vector subspace

EXAMPLES:

We create a 2-dimensional subspace of $\mathbf{Q}^3$.

```
sage: V = VectorSpace(QQ, 3)
sage: B = V.basis()
sage: W = V.vector_space_span([B[0]+B[1], 2*B[1]-B[2]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0  1/2]
[   0    1 -1/2]
```

We create a subspace of a vector space over $\mathbf{Q}(i)$.

```
sage: R.<x> = QQ[]
sage: K = NumberField(x^2 + 1, 'a'); a = K.gen()
sage: V = VectorSpace(K, 3)
sage: V.vector_space_span([2*V.gen(0) + 3*V.gen(2)])
Vector space of degree 3 and dimension 1 over Number Field in a with defining polynomial x^2
Basis matrix:
[  1   0 3/2]
```

We use the `vector_space_span` command to create a vector subspace of the ambient vector space of a submodule of $\mathbf{Z}^3$.

```
sage: M = FreeModule(ZZ,3)
sage: W = M.submodule([M([1,2,3])])
sage: W.vector_space_span([M([2,3,4])])
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[  1 3/2   2]
```

vector_space_span_of_basis(*basis, check=True*)

Create the vector subspace of the ambient vector space with given basis.

INPUT:

- `basis` - a list of linearly independent vectors
- `check` - whether or not to verify that each gen is in the ambient vector space

OUTPUT: a vector subspace with user-specified basis

EXAMPLES:

```
sage: V = VectorSpace(QQ, 3)
sage: B = V.basis()
sage: W = V.vector_space_span_of_basis([B[0]+B[1], 2*B[1]-B[2]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  1  0]
[ 0  2 -1]
```

zero_submodule()

Return the zero submodule of this module.

EXAMPLES:

```
sage: V = FreeModule(ZZ,2)
sage: V.zero_submodule()
Free module of degree 2 and rank 0 over Integer Ring
Echelon basis matrix:
[]
```

class **FreeModule_submodule_field**(*ambient, gens, check=True, already_echelonized=False*)

An embedded vector subspace with echelonized basis.

EXAMPLES:

Since this is an embedded vector subspace with echelonized basis, the echelon_coordinates() and user coordinates() agree:

```
sage: V = QQ^3
sage: W = V.span([[1,2,3],[4,5,6]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
```

```
[ 1  0 -1]
[ 0  1  2]

sage: v = V([1,5,9])
sage: W.echelon_coordinates(v)
[1, 5]
sage: vector(QQ, W.echelon_coordinates(v)) * W.basis_matrix()
(1, 5, 9)
sage: v = V([1,5,9])
sage: W.coordinates(v)
[1, 5]
sage: vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

coordinate_vector(*v, check=True*)

Write v in terms of the user basis for self.

INPUT:

- v - vector
- check - bool (default: True); if True, also verify that v is really in self.

OUTPUT: list

Returns a list c such that if B is the basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raises an ArithmeticError exception.

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.span([[1,2,3],[4,5,6]]); W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: v = V([1,5,9])
sage: W.coordinate_vector(v)
(1, 5)
sage: W.coordinates(v)
[1, 5]
sage: vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)

sage: V = VectorSpace(QQ,5, sparse=True)
sage: W = V.subspace([[0,1,2,0,0], [0,-1,0,0,-1/2]])
sage: W.coordinate_vector([0,0,2,0,-1/2])
(0, 2)
```

echelon_coordinates(*v, check=True*)

Write v in terms of the echelonized basis of self.

INPUT:

- v - vector
- check - bool (default: True); if True, also verify that v is really in self.

OUTPUT: list

Returns a list c such that if B is the basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raises an `ArithmeticError` exception.

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.span([[1,2,3],[4,5,6]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]

sage: v = V([1,5,9])
sage: W.echelon_coordinates(v)
[1, 5]
sage: vector(QQ, W.echelon_coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

has_user_basis()

Return `True` if the basis of this free module is specified by the user, as opposed to being the default echelon form.

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.subspace([[2,'1/2', 1]])
sage: W.has_user_basis()
False
sage: W = V.subspace_with_basis([[2,'1/2',1]])
sage: W.has_user_basis()
True
```

class **FreeModule_submodule_pid**(*ambient, gens, check=True, already_echelonized=False*)

An R-submodule of K^n where K is the fraction field of a principal ideal domain R.

EXAMPLES:

```
sage: M = ZZ^3
sage: W = M.span_of_basis([[1,2,3],[4,5,19]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[ 1  2  3]
[ 4  5 19]
```

We can save and load submodules and elements.

```
sage: loads(W.dumps()) == W
True
sage: v = W.0 + W.1
sage: loads(v.dumps()) == v
True
```

coordinate_vector(*v, check=True*)

Write v in terms of the user basis for self.

INPUT:

- `v` - vector
- `check` - bool (default: True); if True, also verify that v is really in self.

OUTPUT: list

Returns a list c such that if B is the basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raises an `ArithmeticError` exception.

EXAMPLES:

```
sage: V = ZZ^3
sage: W = V.span_of_basis([[1,2,3],[4,5,6]])
sage: W.coordinate_vector([1,5,9])
(5, -1)
```

has_user_basis()

Return `True` if the basis of this free module is specified by the user, as opposed to being the default echelon form.

EXAMPLES:

```
sage: A = ZZ^3; A
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: A.has_user_basis()
False
sage: W = A.span_of_basis([[2,'1/2',1]])
sage: W.has_user_basis()
True
sage: W = A.span([[2,'1/2',1]])
sage: W.has_user_basis()
False
```

class FreeModule_submodule_with_basis_field(*ambient, basis, check=True, echelonize=False, echelonized_basis=None, already_echelonized=False*)

An embedded vector subspace with a distinguished user basis.

EXAMPLES:

```
sage: M = QQ^3; W = M.submodule_with_basis([[1,2,3], [4,5,19]]); W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[ 1  2  3]
[ 4  5 19]
```

Since this is an embedded vector subspace with a distinguished user basis possibly different than the echelonized basis, the echelon_coordinates() and user coordinates() do not agree:

```
sage: V = QQ^3


sage: W = V.submodule_with_basis([[1,2,3], [4,5,6]])
sage: W
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1 2 3]
[4 5 6]


sage: v = V([1,5,9])
sage: W.echelon_coordinates(v)
[1, 5]
sage: vector(QQ, W.echelon_coordinates(v)) * W.echelonized_basis_matrix()
(1, 5, 9)


sage: v = V([1,5,9])
sage: W.coordinates(v)
[5, -1]
```

```
sage: vector(QQ, W.coordinates(v)) * W.basis_matrix()
(1, 5, 9)
```

We can load and save submodules:

```
sage: loads(W.dumps()) == W
True
```

```
sage: K.<x> = FractionField(PolynomialRing(QQ,'x'))
sage: M = K^3; W = M.span_of_basis([[1,1,x]])
sage: loads(W.dumps()) == W
True
```

is_ambient()

Return False since this is not an ambient module.

EXAMPLES:

```
sage: V = QQ^3
sage: V.is_ambient()
True
sage: W = V.span_of_basis([[1,2,3],[4,5,6]])
sage: W.is_ambient()
False
```

class **FreeModule_submodule_with_basis_pid**(*ambient, basis, check=True, echelonize=False, echelonized_basis=None, already_echelonized=False*)

An R-submodule of K^n with distinguished basis, where K is the fraction field of a principal ideal domain R.

ambient_module()

Return the ambient module related to the R-module self, which was used when creating this module, and is of the form R^n. Note that self need not be contained in the ambient module, though self will be contained in the ambient vector space.

EXAMPLES:

```
sage: A = ZZ^3
sage: M = A.span_of_basis([[1,2,'3/7'],[4,5,6]])
sage: M
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[  1   2 3/7]
[  4   5   6]
sage: M.ambient_module()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: M.is_submodule(M.ambient_module())
False
```

ambient_vector_space()

Return the ambient vector space in which this free module is embedded.

EXAMPLES:

```
sage: M = ZZ^3;  M.ambient_vector_space()
Vector space of dimension 3 over Rational Field
```

```
sage: N = M.span_of_basis([[1,2,'1/5']])
sage: N
Free module of degree 3 and rank 1 over Integer Ring
User basis matrix:
[  1   2 1/5]
```

```
sage: M.ambient_vector_space()
Vector space of dimension 3 over Rational Field
sage: M.ambient_vector_space() is N.ambient_vector_space()
True
```

If an inner product on the module is specified, then this is preserved on the ambient vector space.

```
sage: M = FreeModule(ZZ,4,inner_product_matrix=1)
sage: V = M.ambient_vector_space()
sage: V
Ambient quadratic space of dimension 4 over Rational Field
Inner product matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: N = M.submodule([[1,-1,0,0],[0,1,-1,0],[0,0,1,-1]])
sage: N.gram_matrix()
[2 1 1]
[1 2 1]
[1 1 2]
sage: V == N.ambient_vector_space()
True
```

basis()

Return the user basis for this free module.

EXAMPLES:

```
sage: V = ZZ^3
sage: V.basis()
[
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
sage: M = V.span_of_basis([['1/8',2,1]])
sage: M.basis()
[
(1/8, 2, 1)
]
```

change_ring(*R*)

Return the free module over R obtained by coercing each element of self into a vector over the fraction field of R, then taking the resulting R-module. Raises a TypeError if coercion is not possible.

INPUT:

- R - a principal ideal domain

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.subspace([[2,'1/2', 1]])
sage: W.change_ring(GF(7))
Vector space of degree 3 and dimension 1 over Finite Field of size 7
Basis matrix:
[1 2 4]
```

construction()

Returns the functorial construction of self, namely, the subspace of the ambient module spanned by the given basis.

EXAMPLE:

```
sage: M = ZZ^3
sage: W = M.span_of_basis([[1,2,3],[4,5,6]]); W
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1 2 3]
[4 5 6]
sage: c, V = W.construction()
sage: c(V) == W
True
```

coordinate_vector(*v, check=True*)

Write v in terms of the user basis for self.

INPUT:

- v - vector
- check - bool (default: True); if True, also verify that v is really in self.

OUTPUT: list

Returns a vector c such that if B is the basis for self, then

If v is not in self, raises an ArithmeticError exception.

EXAMPLES:

```
sage: V = ZZ^3
sage: M = V.span_of_basis([['1/8',2,1]])
sage: M.coordinate_vector([1,16,8])
(8)
```

echelon_coordinate_vector(*v, check=True*)

Write v in terms of the user basis for self.

INPUT:

- v - vector
- check - bool (default: True); if True, also verify that v is really in self.

Returns a list c such that if B is the echelonized basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raises an ArithmeticError exception.

EXAMPLES:

```
sage: V = ZZ^3
sage: M = V.span_of_basis([['1/2',3,1], [0,'1/6',0]])
sage: B = M.echelonized_basis(); B
[
(1/2, 0, 1),
(0, 1/6, 0)
]
sage: M.echelon_coordinate_vector(['1/2', 3, 1])
(1, 18)
```

echelon_coordinates(*v, check=True*)

Write v in terms of the echelonized basis for self.

INPUT:

- v - vector
- check - bool (default: True); if True, also verify that v is really in self.

OUTPUT: list

Returns a list c such that if B is the basis for self, then

$$\sum c_i B_i = v.$$

If v is not in self, raises an `ArithmeticError` exception.

EXAMPLES:

```
sage: A = ZZ^3
sage: M = A.span_of_basis([[1,2,'3/7'],[4,5,6]])
sage: M.coordinates([8,10,12])
[0, 2]
sage: M.echelon_coordinates([8,10,12])
[8, -2]
sage: B = M.echelonized_basis(); B
[
(1, 2, 3/7),
(0, 3, -30/7)
]
sage: 8*B[0] - 2*B[1]
(8, 10, 12)
```

We do an example with a sparse vector space:

```
sage: V = VectorSpace(QQ,5, sparse=True)
sage: W = V.subspace_with_basis([[0,1,2,0,0], [0,-1,0,0,-1/2]])
sage: W.echelonized_basis()
[
(0, 1, 0, 0, 1/2),
(0, 0, 1, 0, -1/4)
]
sage: W.echelon_coordinates([0,0,2,0,-1/2])
[0, 2]
```

echelon_to_user_matrix()

Return matrix that transforms the echelon basis to the user basis of self. This is a matrix A such that if v is a vector written with respect to the echelon basis for self then vA is that vector written with respect to the user basis of self.

EXAMPLES:

```
sage: V = QQ^3
sage: W = V.span_of_basis([[1,2,3],[4,5,6]])
sage: W.echelonized_basis()
[
(1, 0, -1),
(0, 1, 2)
]
sage: A = W.echelon_to_user_matrix(); A
[-5/3  2/3]
[ 4/3 -1/3]
```

The vector $(1, 1, 1)$ has coordinates $v = (1, 1)$ with respect to the echelonized basis for self. Multiplying vA we find the coordinates of this vector with respect to the user basis.

```
sage: v = vector(QQ, [1,1]); v
(1, 1)
sage: v * A
(-1/3, 1/3)
sage: u0, u1 = W.basis()
```

```
sage: (-u0 + u1)/3
(1, 1, 1)
```

echelonized_basis()

Return the basis for self in echelon form.

EXAMPLES:

```
sage: V = ZZ^3
sage: M = V.span_of_basis([['1/2',3,1], [0,'1/6',0]])
sage: M.basis()
[
(1/2, 3, 1),
(0, 1/6, 0)
]
sage: B = M.echelonized_basis(); B
[
(1/2, 0, 1),
(0, 1/6, 0)
]
sage: V.span(B) == M
True
```

echelonized_basis_matrix()

Return basis matrix for self in row echelon form.

EXAMPLES:

```
sage: V = FreeModule(ZZ, 3).span_of_basis([[1,2,3],[4,5,6]])
sage: V.basis_matrix()
[1 2 3]
[4 5 6]
sage: V.echelonized_basis_matrix()
[1 2 3]
[0 3 6]
```

has_user_basis()

Return `True` if the basis of this free module is specified by the user, as opposed to being the default echelon form.

EXAMPLES:

```
sage: V = ZZ^3; V.has_user_basis()
False
sage: M = V.span_of_basis([[1,3,1]]); M.has_user_basis()
True
sage: M = V.span([[1,3,1]]); M.has_user_basis()
False
```

linear_combination_of_basis(*v*)

Return the linear combination of the basis for self obtained from the coordinates of v.

EXAMPLES:

```
sage: V = span([[1,2,3], [4,5,6]], ZZ); V
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1 2 3]
[0 3 6]
sage: V.linear_combination_of_basis([1,1])
(1, 5, 9)
```

user_to_echelon_matrix()

Return matrix that transforms a vector written with respect to the user basis of self to one written with respect to the echelon basis. The matrix acts from the right, as is usual in Sage.

EXAMPLES:

```
sage: A = ZZ^3
sage: M = A.span_of_basis([[1,2,3],[4,5,6]])
sage: M.echelonized_basis()
[
(1, 2, 3),
(0, 3, 6)
]
sage: M.user_to_echelon_matrix()
[ 1  0]
[ 4 -1]
```

The vector $v = (5, 7, 9)$ in M is $(1, 1)$ with respect to the user basis. Multiplying the above matrix on the right by this vector yields $(5, -1)$, which has components the coordinates of v with respect to the echelon basis.

```
sage: v0,v1 = M.basis(); v = v0+v1
sage: e0,e1 = M.echelonized_basis()
sage: v
(5, 7, 9)
sage: 5*e0 + (-1)*e1
(5, 7, 9)
```

vector_space(*base_field=None*)

Return the vector space associated to this free module via tensor product with the fraction field of the base ring.

EXAMPLES:

```
sage: A = ZZ^3; A
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: A.vector_space()
Vector space of dimension 3 over Rational Field
sage: M = A.span_of_basis([['1/3',2,'3/7'],[4,5,6]]); M
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1/3   2 3/7]
[  4   5   6]
sage: M.vector_space()
Vector space of degree 3 and dimension 2 over Rational Field
User basis matrix:
[1/3   2 3/7]
[  4   5   6]
```

class RealDoubleVectorSpace_class(*n*)

coordinates(*v*)

VectorSpace(*K, dimension, sparse=False, inner_product_matrix=None*)

EXAMPLES:

The base can be complicated, as long as it is a field.

```
sage: V = VectorSpace(FractionField(PolynomialRing(ZZ,'x')),3)
sage: V
Vector space of dimension 3 over Fraction Field of Univariate Polynomial Ring in x over Integer
sage: V.basis()
[
```

```
(1, 0, 0),
(0, 1, 0),
(0, 0, 1)
]
```

The base must be a field or a `TypeError` is raised.

```
sage: VectorSpace(ZZ,5)
...
TypeError: Argument K (= Integer Ring) must be a field.
```

basis_seq(*V, vecs*)

This converts a list vecs of vectors in V to an Sequence of immutable vectors.

Should it? I.e. in most other parts of the system the return type of basis or generators is a tuple.

EXAMPLES:

```
sage: V = VectorSpace(QQ,2)
sage: B = V.gens()
sage: B
((1, 0), (0, 1))
sage: v = B[0]
sage: v[0] = 0 # immutable
...
ValueError: vector is immutable; please change a copy instead (use self.copy())
sage: sage.modules.free_module.basis_seq(V, V.gens())
[
(1, 0),
(0, 1)
]
```

element_class(*R, is_sparse*)

The class of the vectors (elements of a free module) with base ring R and boolean is_sparse.

EXAMPLES:

```
sage: FF = FiniteField(2)
sage: P = PolynomialRing(FF,'x')
sage: sage.modules.free_module.element_class(QQ, is_sparse=True)
<type 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
sage: sage.modules.free_module.element_class(QQ, is_sparse=False)
<type 'sage.modules.vector_rational_dense.Vector_rational_dense'>
sage: sage.modules.free_module.element_class(ZZ, is_sparse=True)
<type 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
sage: sage.modules.free_module.element_class(ZZ, is_sparse=False)
<type 'sage.modules.vector_integer_dense.Vector_integer_dense'>
sage: sage.modules.free_module.element_class(FF, is_sparse=True)
<type 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
sage: sage.modules.free_module.element_class(FF, is_sparse=False)
<type 'sage.modules.vector_modn_dense.Vector_modn_dense'>
sage: sage.modules.free_module.element_class(P, is_sparse=True)
<type 'sage.modules.free_module_element.FreeModuleElement_generic_sparse'>
sage: sage.modules.free_module.element_class(P, is_sparse=False)
<type 'sage.modules.free_module_element.FreeModuleElement_generic_dense'>
```

is_FreeModule(*M*)

Return True if M inherits from from FreeModule_generic.

EXAMPLES:

```
sage: from sage.modules.free_module import is_FreeModule
sage: V = ZZ^3
sage: is_FreeModule(V)
True
sage: W = V.span([ V.random_element() for i in range(2) ])
sage: is_FreeModule(W)
True
```

span(*gens, base_ring=None, check=True, already_echelonized=False*)

Return the R-span of gens (a list of vectors) where R = base_ring.

EXAMPLES:

```
sage: V = span([[1,2,5], [2,2,2]], QQ); V
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -3]
[ 0  1  4]
sage: span([V.gen(0)], QuadraticField(-7,'a'))
Vector space of degree 3 and dimension 1 over Number Field in a with defining polynomial x^2 + 7
Basis matrix:
[ 1  0 -3]
sage: span([[1,2,3], [2,2,2], [1,2,5]], GF(2))
Vector space of degree 3 and dimension 1 over Finite Field of size 2
Basis matrix:
[1 0 1]
```

TESTS:

```
sage: span([[1,2,3], [2,2,2], [1,2/3,5]], ZZ)
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[  1   0  13]
[  0 2/3   6]
[  0   0  14]
sage: span([[1,2,3], [2,2,2], [1,2,QQ['x'].gen()]], ZZ)
...
ValueError: The elements of gens (= [[1, 2, 3], [2, 2, 2], [1, 2, x]]) must be defined over base
```

For backwards compatibility one can also give the base ring as the first argument:

```
sage: span(QQ,[[1,2],[3,4]])
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
```

Fix trac 5575:

```
sage: V = QQ^3
sage: span([V.0, V.1])
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
```

33.3 Elements of free modules

AUTHORS:

- William Stein
- Josh Kantor

TODO: Change to use a get_unsafe / set_unsafe, etc., structure exactly like with matrices, since we'll have to define a bunch of special purpose implementations of vectors easily and systematically.

EXAMPLES: We create a vector space over $\mathbf{Q}$ and a subspace of this space.

```
sage: V = QQ^5
sage: W = V.span([V.1, V.2])
```

Arithmetic operations always return something in the ambient space, since there is a canonical map from W to V but not from V to W.

```
sage: parent(W.0 + V.1)
Vector space of dimension 5 over Rational Field
sage: parent(V.1 + W.0)
Vector space of dimension 5 over Rational Field
sage: W.0 + V.1
(0, 2, 0, 0, 0)
sage: W.0 - V.0
(-1, 1, 0, 0, 0)
```

Next we define modules over $\mathbf{Z}$ and a finite field.

```
sage: K = ZZ^5
sage: M = GF(7)^5
```

Arithmetic between the $\mathbf{Q}$ and $\mathbf{Z}$ modules is defined, and the result is always over $\mathbf{Q}$, since there is a canonical coercion map to $\mathbf{Q}$.

```
sage: K.0 + V.1
(1, 1, 0, 0, 0)
sage: parent(K.0 + V.1)
Vector space of dimension 5 over Rational Field
```

Since there is no canonical coercion map to the finite field from $\mathbf{Q}$ the following arithmetic is not defined:

```
sage: V.0 + M.0
...
TypeError: unsupported operand parent(s) for '+': 'Vector space of dimension 5 over Rational Field' a
```

However, there is a map from $\mathbf{Z}$ to the finite field, so the following is defined, and the result is in the finite field.

```
sage: w = K.0 + M.0; w
(2, 0, 0, 0, 0)
sage: parent(w)
Vector space of dimension 5 over Finite Field of size 7
sage: parent(M.0 + K.0)
Vector space of dimension 5 over Finite Field of size 7
```

Matrix vector multiply:

```
sage: MS = MatrixSpace(QQ,3)
sage: A = MS([0,1,0,1,0,0,0,0,1])
sage: V = QQ^3
sage: v = V([1,2,3])
sage: v * A
(2, 1, 3)
```

TESTS:

```
sage: D = 46341
sage: u = 7
sage: R = Integers(D)
sage: p = matrix(R,[[84, 97, 55, 58, 51]])
sage: 2*p.row(0)
(168, 194, 110, 116, 102)
```

class FreeModuleElement()

An element of a generic free module.

Mod()

EXAMPLES:

```
sage: V = vector(ZZ, [5, 9, 13, 15])
sage: V.Mod(7)
(5, 2, 6, 1)
sage: parent(V.Mod(7))
Vector space of dimension 4 over Ring of integers modulo 7
```

additive_order()

Return the additive order of self.

EXAMPLES:

```
sage: v = vector(Integers(4), [1,2])
sage: v.additive_order()
4

sage: v = vector([1,2,3])
sage: v.additive_order()
+Infinity

sage: v = vector(Integers(30), [6, 15]); v
(6, 15)
sage: v.additive_order()
10
sage: 10*v
(0, 0)
```

apply_map()

Apply the given map phi (an arbitrary Python function or callable object) to this free module element. If R is not given, automatically determine the base ring of the resulting element.

INPUT: **sparse – True or False will control whether the result** is sparse. By default, the result is sparse iff self is sparse.

- `phi` - arbitrary Python function or callable object
- `R` - (optional) ring

OUTPUT: a free module element over R
EXAMPLES:

```
sage: m = vector([1,x,sin(x+1)])
sage: m.apply_map(lambda x: x^2)
(1, x^2, sin(x + 1)^2)
sage: m.apply_map(sin)
(sin(1), sin(x), sin(sin(x + 1)))

sage: m = vector(ZZ, 9, range(9))
sage: k.<a> = GF(9)
sage: m.apply_map(k)
(0, 1, 2, 0, 1, 2, 0, 1, 2)
```

In this example, we explicitly specify the codomain.

```
sage: s = GF(3)
sage: f = lambda x: s(x)
sage: n = m.apply_map(f, k); n
(0, 1, 2, 0, 1, 2, 0, 1, 2)
sage: n.parent()
Vector space of dimension 9 over Finite Field in a of size 3^2
```

If your map sends 0 to a non-zero value, then your resulting vector is not mathematically sparse:

```
sage: v = vector([0] * 6 + [1], sparse=True); v
(0, 0, 0, 0, 0, 0, 1)
sage: v2 = v.apply_map(lambda x: x+1); v2
(1, 1, 1, 1, 1, 1, 2)
```

but it's still represented with a sparse data type:

```
sage: parent(v2)
Ambient sparse free module of rank 7 over the principal ideal domain Integer Ring
```

This data type is inefficient for dense vectors, so you may want to specify sparse=False:

```
sage: v2 = v.apply_map(lambda x: x+1, sparse=False); v2
(1, 1, 1, 1, 1, 1, 2)
sage: parent(v2)
Ambient free module of rank 7 over the principal ideal domain Integer Ring
```

Or if you have a map that will result in mostly zeroes, you may want to specify sparse=True:

```
sage: v = vector(srange(10))
sage: v2 = v.apply_map(lambda x: 0 if x else 1, sparse=True); v2
(1, 0, 0, 0, 0, 0, 0, 0, 0, 0)
sage: parent(v2)
Ambient sparse free module of rank 10 over the principal ideal domain Integer Ring
```

TESTS:

```
sage: m = vector(SR,[])
sage: m.apply_map(lambda x: x*x) == m
True
```

Check that we don't unnecessarily apply phi to 0 in the sparse case:

```
sage: m = vector(ZZ, range(1, 4), sparse=True)
sage: m.apply_map(lambda x: 1/x)
(1, 1/2, 1/3)

sage: parent(vector(RDF, (), sparse=True).apply_map(lambda x: x, sparse=True))
Sparse vector space of dimension 0 over Real Double Field
```

```
sage: parent(vector(RDF, (), sparse=True).apply_map(lambda x: x, sparse=False))
Vector space of dimension 0 over Real Double Field
sage: parent(vector(RDF, (), sparse=False).apply_map(lambda x: x, sparse=True))
Sparse vector space of dimension 0 over Real Double Field
sage: parent(vector(RDF, (), sparse=False).apply_map(lambda x: x, sparse=False))
Vector space of dimension 0 over Real Double Field
```

change_ring()

Change the base ring of this vector, by coercing each element of this vector into R.

EXAMPLES:

```
sage: v = vector(QQ['x,y'], [1..5]); v.change_ring(GF(3))
(1, 2, 0, 1, 2)
```

copy()

Make a copy of this vector.

EXAMPLES:

```
sage: v = vector([1..5]); v
(1, 2, 3, 4, 5)
sage: w = v.copy()
sage: v == w
True
sage: v is w
False

sage: v = vector([1..5], sparse=True); v
(1, 2, 3, 4, 5)
sage: v.copy()
(1, 2, 3, 4, 5)
```

cross_product()

Return the cross product of self and right, which is only defined for vectors of length 3.

This product is performed under the assumption that the basis vectors are orthonormal.

EXAMPLES:

```
sage: v = vector([1,2,3]); w = vector([0,5,-9])
sage: v.cross_product(v)
(0, 0, 0)
sage: u = v.cross_product(w); u
(-33, 9, 5)
sage: u.dot_product(v)
0
sage: u.dot_product(w)
0
```

degree()

denominator()

dense_vector()

derivative()

Derivative with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

EXAMPLES:

```
sage: v = vector([1,x,x^2])
sage: v.derivative(x)
```

```
(0, 1, 2*x)
sage: type(v.derivative(x)) == type(v)
True
sage: v = vector([1,x,x^2], sparse=True)
sage: v.derivative(x)
(0, 1, 2*x)
sage: type(v.derivative(x)) == type(v)
True
sage: v.derivative(x,x)
(0, 0, 2)
```

dict()

dot_product()

Return the dot product of self and right, which is the sum of the product of the corresponding entries.

INPUT:

- `right` - vector of the same degree as self. It need not be in the same vector space as self, as long as the coefficients can be multiplied.

EXAMPLES:

```
sage: V = FreeModule(ZZ, 3)
sage: v = V([1,2,3])
sage: w = V([4,5,6])
sage: v.dot_product(w)
32

sage: W = VectorSpace(GF(3),3)
sage: w = W([0,1,2])
sage: w.dot_product(v)
2
sage: w.dot_product(v).parent()
Finite Field of size 3
```

Implicit coercion is well defined (regardless of order), so we get 2 even if we do the dot product in the other order.

```
sage: v.dot_product(w)
2
```

element()

get()

get is meant to be more efficient than getitem, because it does not do any error checking.

inner_product()

Returns the inner product of self and other, with respect to the inner product defined on the parent of self.

EXAMPLES:

```
sage: I = matrix(ZZ,3,[2,0,-1,0,2,0,-1,0,6])
sage: M = FreeModule(ZZ, 3, inner_product_matrix = I)
sage: (M.0).inner_product(M.0)
2
sage: K = M.span_of_basis([[0/2,-1/2,-1/2], [0,1/2,-1/2],[2,0,0]])
sage: (K.0).inner_product(K.0)
2
```

is_dense()

is_immutable()

Return True if this vector is immutable, i.e., the entries cannot be changed.

EXAMPLES:

```
sage: v = vector(QQ['x,y'], [1..5]); v.is_immutable()
False
sage: v.set_immutable()
sage: v.is_immutable()
True
```

is_mutable()

Return True if this vector is mutable, i.e., the entries can be changed.

EXAMPLES:

```
sage: v = vector(QQ['x,y'], [1..5]); v.is_mutable()
True
sage: v.set_immutable()
sage: v.is_mutable()
False
```

is_sparse()

is_vector()

iteritems()

lift()

EXAMPLES:

```
sage: V = vector(Integers(7), [5, 9, 13, 15]) ; V
(5, 2, 6, 1)
sage: V.lift()
(5, 2, 6, 1)
sage: parent(V.lift())
Ambient free module of rank 4 over the principal ideal domain Integer Ring
```

list()

list_from_positions()

nonzero_positions()

Return the sorted list of integers i such that self[i] != 0.

norm()

Return the p-norm of this vector, where p can be a real number ≥ 1, Infinity, or a symbolic expression. If $p = 2$ (default), this is the usual Euclidean norm; if p=Infinity, this is the maximum norm; if $p = 1$, this is the taxicab (Manhattan) norm.

EXAMPLES:

```
sage: v = vector([1,2,-3])
sage: v.norm(5)
276^(1/5)
```

The default is the usual Euclidean norm:

```
sage: v.norm()
sqrt(14)
sage: v.norm(2)
sqrt(14)
```

The infinity norm is the maximum size of any entry:

```
sage: v.norm(Infinity)
3
```

Any real or symbolic value works:

```
sage: v=vector(RDF,[1,2,3])
sage: v.norm(5)
3.07738488539
sage: v.norm(pi/2)
4.2165958647
sage: _=var('a b c d p'); v=vector([a, b, c, d])
sage: v.norm(p)
(abs(a)^p + abs(b)^p + abs(c)^p + abs(d)^p)^(1/p)
```

normalize()

Return this vector divided through by the first nonzero entry of this vector.

EXAMPLES:

```
sage: v = vector(QQ,[0,4/3,5,1,2])
sage: v.normalize()
(0, 1, 15/4, 3/4, 3/2)
```

pairwise_product()

Return the pairwise product of self and right, which is a vector of the products of the corresponding entries.

INPUT:

- `right` - vector of the same degree as self. It need not be in the same vector space as self, as long as the coefficients can be multiplied.

EXAMPLES:

```
sage: V = FreeModule(ZZ, 3)
sage: v = V([1,2,3])
sage: w = V([4,5,6])
sage: v.pairwise_product(w)
(4, 10, 18)
sage: sum(v.pairwise_product(w)) == v.dot_product(w)
True

sage: W = VectorSpace(GF(3),3)
sage: w = W([0,1,2])
sage: w.pairwise_product(v)
(0, 2, 0)
sage: w.pairwise_product(v).parent()
Vector space of dimension 3 over Finite Field of size 3
```

Implicit coercion is well defined (regardless of order), so we get 2 even if we do the dot product in the other order.

```
sage: v.pairwise_product(w).parent()
Vector space of dimension 3 over Finite Field of size 3
```

TESTS:

```
sage: x, y = var('x, y')

sage: parent(vector(ZZ,[1,2]).pairwise_product(vector(ZZ,[1,2])))
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: parent(vector(ZZ,[1,2]).pairwise_product(vector(QQ,[1,2])))
Vector space of dimension 2 over Rational Field
sage: parent(vector(QQ,[1,2]).pairwise_product(vector(ZZ,[1,2])))
Vector space of dimension 2 over Rational Field
sage: parent(vector(QQ,[1,2]).pairwise_product(vector(QQ,[1,2])))
Vector space of dimension 2 over Rational Field
```

```
sage: parent(vector(QQ,[1,2,3,4]).pairwise_product(vector(ZZ[x],[1,2,3,4])))
Ambient free module of rank 4 over the principal ideal domain Univariate Polynomial Ring in
sage: parent(vector(ZZ[x],[1,2,3,4]).pairwise_product(vector(QQ,[1,2,3,4])))
Ambient free module of rank 4 over the principal ideal domain Univariate Polynomial Ring in

sage: parent(vector(QQ,[1,2,3,4]).pairwise_product(vector(ZZ[x][y],[1,2,3,4])))
Ambient free module of rank 4 over the integral domain Univariate Polynomial Ring in y over
sage: parent(vector(ZZ[x][y],[1,2,3,4]).pairwise_product(vector(QQ,[1,2,3,4])))
Ambient free module of rank 4 over the integral domain Univariate Polynomial Ring in y over

sage: parent(vector(QQ[x],[1,2,3,4]).pairwise_product(vector(ZZ[x][y],[1,2,3,4])))
Ambient free module of rank 4 over the integral domain Univariate Polynomial Ring in y over
sage: parent(vector(ZZ[x][y],[1,2,3,4]).pairwise_product(vector(QQ[x],[1,2,3,4])))
Ambient free module of rank 4 over the integral domain Univariate Polynomial Ring in y over

sage: parent(vector(QQ[y],[1,2,3,4]).pairwise_product(vector(ZZ[x][y],[1,2,3,4])))
Ambient free module of rank 4 over the integral domain Univariate Polynomial Ring in y over
sage: parent(vector(ZZ[x][y],[1,2,3,4]).pairwise_product(vector(QQ[y],[1,2,3,4])))
Ambient free module of rank 4 over the integral domain Univariate Polynomial Ring in y over

sage: parent(vector(ZZ[x],[1,2,3,4]).pairwise_product(vector(ZZ[y],[1,2,3,4])))
...
TypeError: no common canonical parent for objects with parents: 'Ambient free module of rank
sage: parent(vector(ZZ[x],[1,2,3,4]).pairwise_product(vector(QQ[y],[1,2,3,4])))
...
TypeError: no common canonical parent for objects with parents: 'Ambient free module of rank
sage: parent(vector(QQ[x],[1,2,3,4]).pairwise_product(vector(ZZ[y],[1,2,3,4])))
...
TypeError: no common canonical parent for objects with parents: 'Ambient free module of rank
sage: parent(vector(QQ[x],[1,2,3,4]).pairwise_product(vector(QQ[y],[1,2,3,4])))
...
TypeError: no common canonical parent for objects with parents: 'Ambient free module of rank
sage: v = vector({1: 1, 3: 2})  # test sparse vectors
sage: w = vector({0: 6, 3: -4})
sage: v.pairwise_product(w)
(0, 0, 0, -8)
sage: w.pairwise_product(v) == v.pairwise_product(w)
True
```

plot()

INPUT:

- **plot_type - (default: 'arrow' if v has 3 or fewer components,** otherwise 'step') type of plot. Options are:
  - 'arrow' to draw an arrow
  - 'point' to draw a point at the coordinates specified by the vector
  - 'step' to draw a step function representing the coordinates of the vector.

  Both 'arrow' and 'point' raise exceptions if the vector has more than 3 dimensions.

EXAMPLES:

```
sage: v = vector(RDF, (1,2))
sage: eps = 0.1
sage: plot(v, plot_type='arrow')
sage: plot(v, plot_type='point')
sage: plot(v, plot_type='step') # calls v.plot_step()
sage: plot(v, plot_type='step', eps=eps, xmax=5, hue=0)
sage: v = vector(RDF, (1,2,1))
```

```
sage: plot(v) # defaults to an arrow plot
sage: plot(v, plot_type='arrow')
sage: from sage.plot.plot3d.shapes2 import frame3d
sage: plot(v, plot_type='point')+frame3d((0,0,0), v.list())
sage: plot(v, plot_type='step') # calls v.plot_step()
sage: plot(v, plot_type='step', eps=eps, xmax=5, hue=0)
sage: v = vector(RDF, (1,2,3,4))
sage: plot(v) # defaults to a step plot
```

plot_step()

INPUT:

- xmin - (default: 0) start x position to start plotting
- xmax - (default: 1) stop x position to stop plotting
- eps - (default: determined by xmax) we view this vector as defining a function at the points xmin, xmin + eps, xmin + 2*eps, ...,
- res - (default: all points) total number of points to include in the graph
- connect - (default: True) if True draws a line; otherwise draw a list of points.

EXAMPLES:

```
sage: eps=0.1
sage: v = vector(RDF, [sin(n*eps) for n in range(100)])
sage: v.plot_step(eps=eps, xmax=5, hue=0)
```

set()

set is meant to be more efficient than setitem, because it does not do any error checking or coercion. Use with care.

set_immutable()

Make this vector immutable. This operation can't be undone.

EXAMPLES:

```
sage: v = vector([1..5]); v
(1, 2, 3, 4, 5)
sage: v[1] = 10
sage: v.set_immutable()
sage: v[1] = 10
...
ValueError: vector is immutable; please change a copy instead (use self.copy())
```

sparse_vector()

support()

Return the integers i such that self[i] != 0. This is the same as the nonzero_positions function.

transpose()

EXAMPLES:

```
sage: v = vector(ZZ, [2, 12, 22])
sage: transpose(vector(v))
[ 2]
[12]
[22]

sage: transpose(vector(GF(7), v))
[2]
[5]
[1]
```

```
sage: transpose(vector(v, ZZ['x', 'y']))
[ 2]
[12]
[22]
```

class FreeModuleElement_generic_dense()

A generic dense element of a free module.

list()

n()

Returns a numerical approximation of self by calling the n() method on all of its entries.

EXAMPLES:

```
sage: v = vector(RealField(212), [1,2,3])
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 53 bits of precision
sage: v.n(prec=75)
(1.000000000000000000000, 2.000000000000000000000, 3.000000000000000000000)
sage: _.parent()
Vector space of dimension 3 over Real Field with 75 bits of precision
```

class FreeModuleElement_generic_sparse()

A generic sparse free module element is a dictionary with keys ints i and entries in the base ring.

EXAMPLES:

Pickling works:

```
sage: v = FreeModule(ZZ, 3, sparse=True).0
sage: loads(dumps(v)) == v
True
sage: v = FreeModule(Integers(8)['x,y'], 5, sparse=True).1
sage: loads(dumps(v)) - v
(0, 0, 0, 0, 0)
```

```
sage: a = vector([-1,0,1/1],sparse=True); b = vector([-1/1,0,0],sparse=True)
sage: a.parent()
Sparse vector space of dimension 3 over Rational Field
sage: b - a
(0, 0, -1)
sage: (b-a).dict()
{2: -1}
```

denominator()

dict()

get()

Like __getitem__ but with no type or bounds checking. Returns 0 if access is out of bounds.

iteritems()

list()

n()

Returns a numerical approximation of self by calling the n() method on all of its entries.

EXAMPLES:

```
sage: v = vector(RealField(200), [1,2,3], sparse=True)
sage: v.n()
(1.00000000000000, 2.00000000000000, 3.00000000000000)
sage: _.parent()
Sparse vector space of dimension 3 over Real Field with 53 bits of precision
sage: v.n(prec=75)
(1.000000000000000000000, 2.000000000000000000000, 3.000000000000000000000)
sage: _.parent()
Sparse vector space of dimension 3 over Real Field with 75 bits of precision
```

nonzero_positions()
Returns the list of numbers i such that self[i] != 0.
EXAMPLES:

```
sage: v = vector({1: 1, 3: -2})
sage: w = vector({1: 4, 3: 2})
sage: v+w
(0, 5, 0, 0)
sage: (v+w).nonzero_positions()
[1]
```

set()
Like __setitem__ but with no type or bounds checking.

free_module_element()
Return a vector over R with given entries.
CALL FORMATS:

1.vector(object)
2.vector(ring, object)
3.vector(object, ring)
4.vector(numpy_array)

In each case, give `sparse=True` or `sparse=False` as an option.
INPUT:

- `elts` - entries of a vector (either a list or dict).
- `R` - ring
- `sparse` - optional

OUTPUT: An element of the free module over R of rank len(elts).
EXAMPLES:

```
sage: v = vector([1,2,3]); v
(1, 2, 3)
sage: v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: v = vector([1,2,3/5]); v
(1, 2, 3/5)
sage: v.parent()
Vector space of dimension 3 over Rational Field
```

All entries must *canonically* coerce to some common ring:

```
sage: v = vector([17, GF(11)(5), 19/3]); v
...
TypeError: unable to find a common ring for all elements
```

```
sage: v = vector([17, GF(11)(5), 19]); v
(6, 5, 8)
sage: v.parent()
Vector space of dimension 3 over Finite Field of size 11
sage: v = vector([17, GF(11)(5), 19], QQ); v
(17, 5, 19)
sage: v.parent()
Vector space of dimension 3 over Rational Field
sage: v = vector((1,2,3), QQ); v
(1, 2, 3)
sage: v.parent()
Vector space of dimension 3 over Rational Field
sage: v = vector(QQ, (1,2,3)); v
(1, 2, 3)
sage: v.parent()
Vector space of dimension 3 over Rational Field
sage: v = vector(vector([1,2,3])); v
(1, 2, 3)
sage: v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

You can also use `free_module_element`, which is the same as `vector`.

```
sage: free_module_element([1/3, -4/5])
(1/3, -4/5)
```

Make a vector mod 3 out of a vector over **Z**:

```
sage: vector(vector([1,2,3]), GF(3))
(1, 2, 0)
```

Here we illustrate the creation of sparse vectors by using a dictionary:

```
sage: vector({1:1.1, 3:3.14})
(0.000000000000000, 1.10000000000000, 0.000000000000000, 3.14000000000000)
```

Any 1 dimensional numpy array of type float or complex may be passed to vector. The result will be a vector in the appropriate dimensional vector space over the real double field or the complex double field. The data in the array must be contiguous so columnwise slices of numpy matrices will raise an exception.

```
sage: import numpy
sage: x=numpy.random.randn(10)
sage: y=vector(x)
sage: v=numpy.random.randn(10)*numpy.complex(0,1)
sage: w=vector(v)
```

If any of the arguments to vector have Python type int, long, real, or complex, they will first be coerced to the appropriate Sage objects. This fixes trac #3847:

```
sage: v = vector([int(0)]); v
(0)
sage: v[0].parent()
Integer Ring
sage: v = vector(range(10)); v
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
sage: v[3].parent()
Integer Ring
sage: v = vector([float(23.4), int(2), complex(2+7*I), long(1)]); v
```

```
(23.4, 2.0, 2.0 + 7.0*I, 1.0)
sage: v[1].parent()
Complex Double Field
```

is_FreeModuleElement()

make_FreeModuleElement_generic_dense()

make_FreeModuleElement_generic_dense_v1()

make_FreeModuleElement_generic_sparse()

make_FreeModuleElement_generic_sparse_v1()

prepare()

prepare_dict()

EXAMPLES:

```
sage: from sage.modules.free_module_element import prepare_dict
sage: prepare_dict({3:1 , 5:3}, QQ)
([0, 0, 0, 1, 0, 3], Rational Field)
sage: prepare_dict({},QQ)
([], Rational Field)
```

vector()

Return a vector over R with given entries.

CALL FORMATS:

1.vector(object)

2.vector(ring, object)

3.vector(object, ring)

4.vector(numpy_array)

In each case, give `sparse=True` or `sparse=False` as an option.

INPUT:

- `elts` - entries of a vector (either a list or dict).
- `R` - ring
- `sparse` - optional

OUTPUT: An element of the free module over R of rank len(elts).

EXAMPLES:

```
sage: v = vector([1,2,3]); v
(1, 2, 3)
sage: v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: v = vector([1,2,3/5]); v
(1, 2, 3/5)
sage: v.parent()
Vector space of dimension 3 over Rational Field
```

All entries must *canonically* coerce to some common ring:

```
sage: v = vector([17, GF(11)(5), 19/3]); v
...
TypeError: unable to find a common ring for all elements
```

```
sage: v = vector([17, GF(11)(5), 19]); v
(6, 5, 8)
sage: v.parent()
Vector space of dimension 3 over Finite Field of size 11
sage: v = vector([17, GF(11)(5), 19], QQ); v
(17, 5, 19)
sage: v.parent()
Vector space of dimension 3 over Rational Field
sage: v = vector((1,2,3), QQ); v
(1, 2, 3)
sage: v.parent()
Vector space of dimension 3 over Rational Field
sage: v = vector(QQ, (1,2,3)); v
(1, 2, 3)
sage: v.parent()
Vector space of dimension 3 over Rational Field
sage: v = vector(vector([1,2,3])); v
(1, 2, 3)
sage: v.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

You can also use `free_module_element`, which is the same as `vector`.

```
sage: free_module_element([1/3, -4/5])
(1/3, -4/5)
```

Make a vector mod 3 out of a vector over **Z**:

```
sage: vector(vector([1,2,3]), GF(3))
(1, 2, 0)
```

Here we illustrate the creation of sparse vectors by using a dictionary:

```
sage: vector({1:1.1, 3:3.14})
(0.000000000000000, 1.10000000000000, 0.000000000000000, 3.14000000000000)
```

Any 1 dimensional numpy array of type float or complex may be passed to vector. The result will be a vector in the appropriate dimensional vector space over the real double field or the complex double field. The data in the array must be contiguous so columnwise slices of numpy matrices will raise an exception.

```
sage: import numpy
sage: x=numpy.random.randn(10)
sage: y=vector(x)
sage: v=numpy.random.randn(10)*numpy.complex(0,1)
sage: w=vector(v)
```

If any of the arguments to vector have Python type int, long, real, or complex, they will first be coerced to the appropriate Sage objects. This fixes trac #3847:

```
sage: v = vector([int(0)]); v
(0)
sage: v[0].parent()
Integer Ring
sage: v = vector(range(10)); v
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
sage: v[3].parent()
Integer Ring
sage: v = vector([float(23.4), int(2), complex(2+7*I), long(1)]); v
```

```
(23.4, 2.0, 2.0 + 7.0*I, 1.0)
sage: v[1].parent()
Complex Double Field
```

33.4 Pickling for the old CDF vector class.

AUTHORS:

- Jason Grout

TESTS:

```
sage: v = vector(CDF,[(1,-1), (2,pi), (3,5)])
sage: v
(1.0 - 1.0*I, 2.0 + 3.14159265359*I, 3.0 + 5.0*I)
sage: loads(dumps(v)) == v
True
```

33.5 Pickling for the old RDF vector class.

AUTHORS:

- Jason Grout

TESTS:

```
sage: v = vector(RDF, [1,2,3,4])
sage: loads(dumps(v)) == v
True
```

33.6 Homspaces between free modules

EXAMPLES: We create $\mathrm{End}(\mathbf{Z}^2)$ and compute a basis.

```
sage: M = FreeModule(IntegerRing(),2)
sage: E = End(M)
sage: B = E.basis()
sage: len(B)
4
sage: B[0]
Free module morphism defined by the matrix
[1 0]
[0 0]
Domain: Ambient free module of rank 2 over the principal ideal domain ...
Codomain: Ambient free module of rank 2 over the principal ideal domain ...
```

We create $\mathrm{Hom}(\mathbf{Q}^3, \mathbf{Q}^2)$ and compute a basis.

```
sage: V3 = VectorSpace(RationalField(),3)
sage: V2 = VectorSpace(RationalField(),2)
sage: H = Hom(V3,V2)
sage: H
Set of Morphisms from Vector space of dimension 3 over Rational Field
to Vector space of dimension 2 over Rational Field in Category of
vector spaces over Rational Field
sage: B = H.basis()
sage: len(B)
6
sage: B[0]
Free module morphism defined by the matrix
[1 0]
[0 0]
[0 0]...
```

TESTS:

```
sage: H = Hom(QQ^2, QQ^1)
sage: loads(dumps(H)) == H
True
```

See trac 5886:

```
sage: V = (QQ^2).span_of_basis([[1,2],[3,4]])
sage: V.hom([V.0, V.1])
Free module morphism defined by the matrix
[1 0]
[0 1]...
```

class FreeModuleHomspace(*X, Y, cat=None, check=True, base=None*)

basis()

Return a basis for this space of free module homomorphisms.

OUTPUT: • tuple

EXAMPLES:

```
sage: H = Hom(QQ^2, QQ^1)
sage: H.basis()
(Free module morphism defined by the matrix
[1]
[0]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 1 over Rational Field,
 Free module morphism defined by the matrix
[0]
[1]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 1 over Rational Field)
```

identity()

Return identity morphism in an endomorphism ring.

EXAMPLE:

```
sage: V=VectorSpace(QQ,5)
sage: H=V.Hom(V)
sage: H.identity()
Free module morphism defined by the matrix
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
Domain: Vector space of dimension 5 over Rational Field
Codomain: Vector space of dimension 5 over Rational Field
```

is_FreeModuleHomspace(*x*)

Return True if x is a Free module homspace.

EXAMPLES:

```
sage: H = Hom(QQ^3, QQ^2)
sage: sage.modules.free_module_homspace.is_FreeModuleHomspace(H)
True
sage: sage.modules.free_module_homspace.is_FreeModuleHomspace(2)
False
```

33.7 Morphisms of free modules

class FreeModuleMorphism(*parent, A*)

is_FreeModuleMorphism(*x*)

33.8 Morphisms defined by a matrix.

A matrix morphism is a morphism that is defined by multiplication by a matrix. Elements of domain must either have a method `vector()` that returns a vector that the defining matrix can hit from the left, or be coercible into vector space of appropriate dimension.

EXAMPLES:

```
sage: from sage.modules.matrix_morphism import MatrixMorphism, is_MatrixMorphism
sage: V = QQ^3
sage: T = End(V)
sage: M = MatrixSpace(QQ,3)
sage: I = M.identity_matrix()
sage: m = MatrixMorphism(T, I); m
Morphism defined by the matrix
[1 0 0]
[0 1 0]
[0 0 1]
sage: is_MatrixMorphism(m)
True
sage: m.charpoly('x')
x^3 - 3*x^2 + 3*x - 1
sage: m.base_ring()
Rational Field
sage: m.det()
```

```
1
sage: m.fcp('x')
(x - 1)^3
sage: m.matrix()
[1 0 0]
[0 1 0]
[0 0 1]
sage: m.rank()
3
sage: m.trace()
3
```

AUTHOR:

- William Stein: initial versions
- David Joyner (2005-12-17): added examples
- William Stein (2005-01-07): added __reduce__
- Craig Citro (2008-03-18): refactored MatrixMorphism class

class **MatrixMorphism**(*parent, A*)

A morphism defined by a matrix.

matrix()

Return matrix that defines this morphism.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([3*V.0, 2*V.1])
sage: phi.matrix()
[3 0]
[0 2]
```

class **MatrixMorphism_abstract**(*parent*)

base_ring()

Return the base ring of self, that is, the ring over which self is given by a matrix.

EXAMPLES:

```
sage: sage.modules.matrix_morphism.MatrixMorphism((ZZ**2).endomorphism_ring(), Matrix(ZZ,2,[
Integer Ring
```

charpoly(*var='x'*)

Return the characteristic polynomial of this endomorphism.

INPUT: • var – variable

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([V.0+V.1, 2*V.1])
sage: phi.charpoly()
x^2 - 3*x + 2
sage: phi.matrix().charpoly()
x^2 - 3*x + 2
sage: phi.charpoly('T')
T^2 - 3*T + 2
```

decomposition(**args, **kwds*)

Return decomposition of this endomorphism, i.e., sequence of subspaces obtained by finding invariant subspaces of self.

See the documentation for self.matrix().decomposition for more details. All inputs to this function are passed onto the matrix one.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([V.0+V.1, 2*V.1])
sage: phi.decomposition()
[
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1],
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1 -1]
]
```

det()

Return the determinant of this endomorphism.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([V.0+V.1, 2*V.1])
sage: phi.det()
2
```

fcp(*var='x'*)

Return the factorization of the characteristic polynomial.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([V.0+V.1, 2*V.1])
sage: phi.fcp()
(x - 2) * (x - 1)
sage: phi.fcp('T')
(T - 2) * (T - 1)
```

image()

Compute the image of this morphism.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: phi = V.Hom(V)(range(9))
sage: phi.image()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: hom(GF(7)^3, GF(7)^2, 0).image()
Vector space of degree 2 and dimension 0 over Finite Field of size 7
Basis matrix:
[]
```

Compute the image of the identity map on a ZZ-submodule:

```
sage: V = (ZZ^2).span([[1,2],[3,4]])
sage: phi = V.Hom(V)(identity_matrix(ZZ,2))
sage: phi(V.0) == V.0
True
sage: phi(V.1) == V.1
```

```
True
sage: phi.image()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 0]
[0 2]
sage: phi.image() == V
True
```

kernel()

Compute the kernel of this morphism.

EXAMPLES:

```
sage: V = VectorSpace(QQ,3)
sage: id = V.Hom(V)(identity_matrix(QQ,3))
sage: null = V.Hom(V)(0*identity_matrix(QQ,3))
sage: id.kernel()
Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
sage: phi = V.Hom(V)(matrix(QQ,3,range(9)))
sage: phi.kernel()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
sage: hom(CC^2, CC^2, 1).kernel()
Vector space of degree 2 and dimension 0 over Complex Field with 53 bits of precision
Basis matrix:
[]
```

matrix()

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom(V.basis())
sage: phi.matrix()
[1 0]
[0 1]
sage: sage.modules.matrix_morphism.MatrixMorphism_abstract.matrix(phi)
...
NotImplementedError: this method must be overridden in the extension class
```

rank()

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom(V.basis())
sage: phi.rank()
2
sage: V = ZZ^2; phi = V.hom([V.0, V.0])
sage: phi.rank()
1
```

restrict(*sub*)

Restrict this matrix morphism to a subspace sub of the domain.

The codomain and domain of the resulting matrix are both sub.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([3*V.0, 2*V.1])
sage: phi.restrict(V.span([V.0]))
Free module morphism defined by the matrix
```

```
[3]
Domain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
Codomain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
```

restrict_codomain(*sub*)

Restrict this matrix morphism to a subspace sub of the codomain.

The resulting morphism has the same domain as before, but a new codomain.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([4*(V.0+V.1),0])
sage: W = V.span([2*(V.0+V.1)])
sage: phi
Free module morphism defined by the matrix
[4 4]
[0 0]
Domain: Ambient free module of rank 2 over the principal ideal domain ...
Codomain: Ambient free module of rank 2 over the principal ideal domain ...
sage: psi = phi.restrict_codomain(W); psi
Free module morphism defined by the matrix
[2]
[0]
Domain: Ambient free module of rank 2 over the principal ideal domain ...
Codomain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
```

restrict_domain(*sub*)

Restrict this matrix morphism to a subspace sub of the domain. The subspace sub should have a basis() method and elements of the basis should be coercible into domain.

The resulting morphism has the same codomain as before, but a new domain.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([3*V.0, 2*V.1])
sage: phi.restrict_domain(V.span([V.0]))
Free module morphism defined by the matrix
[3 0]
Domain: Free module of degree 2 and rank 1 over Integer Ring
Echelon ...
Codomain: Ambient free module of rank 2 over the principal ideal domain ...
sage: phi.restrict_domain(V.span([V.1]))
Free module morphism defined by the matrix
[0 2]...
```

trace()

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([V.0+V.1, 2*V.1])
sage: phi.trace()
3
```

is_MatrixMorphism(*x*)

Return True if x is a Matrix morphism of free modules.

EXAMPLES:

```
sage: V = ZZ^2; phi = V.hom([3*V.0, 2*V.1])
sage: sage.modules.matrix_morphism.is_MatrixMorphism(phi)
True
```

```
sage: sage.modules.matrix_morphism.is_MatrixMorphism(3)
False
```

CHAPTER

THIRTYFOUR

COMBINATORIAL GEOMETRY

Sage includes support for computing with lattice and reflexive polytopes and Groebner fans. It also has optional support for computing with general polytopes via the polymake program.

34.1 Lattice and reflexive polytopes

This module provides tools for work with lattice and reflexive polytopes. A *convex polytope* is the convex hull of finitely many points in $\mathbf{R}^n$. The dimension n of a polytope is the smallest n such that the polytope can be embedded in $\mathbf{R}^n$.

A *lattice polytope* is a polytope whose vertices all have integer coordinates.

If L is a lattice polytope, the dual polytope of L is

$$\{y \in \mathbf{Z}^n : x \cdot y \geq -1 \text{ all } x \in L\}$$

A *reflexive polytope* is a lattice polytope, such that its polar is also a lattice polytope, i.e. has vertices with integer coordinates.

This Sage module uses Package for Analyzing Lattice Polytopes (PALP), which is a program written in C by Maximilian Kreuzer and Harald Skarke, which is freely available under the GNU licence terms at http://tph16.tuwien.ac.at/~kreuzer/CY/. Moreover, PALP is included standard with Sage.

PALP is described in the paper math.SC/0204356. Its distribution also contains the application nef.x, which was created by Erwin Riegler and computes nef partitions and Hodge data for toric complete intersections.

ACKNOWLEDGMENT: polytope.py module written by William Stein was used as an example of organizing an interface between an external program and Sage. William Stein also helped Andrey Novoseltsev with debugging and tuning of this module.

Robert Bradshaw helped Andrey Novoseltsev to realize plot3d function.

Note: IMPORTANT: PALP requires some parameters to be determined during compilation time, i.e., the maximum dimension of polytopes, the maximum number of points, etc. These limitations may lead to errors during calls to different functions of these module. Currently, a ValueError exception will be raised if the output of poly.x or nef.x is empty or contains the exclamation mark. The error message will contain the exact command that caused an error, the description and vertices of the polytope, and the obtained output.

Data obtained from PALP and some other data is cached and most returned values are immutable. In particular, you cannot change the vertices of the polytope or their order after creation of the polytope.

If you are going to work with large sets of data, take a look at `all_*` functions in this module. They precompute different data for sequences of polynomials with a few runs of external programs. This can significantly affect the time of future computations. You can also use dump/load, but not all data will be stored (currently only faces and the number of their internal and boundary points are stored, in addition to polytope vertices and its polar).

AUTHORS:

- Andrey Novoseltsev (2007-01-11): initial version of this module
- Andrey Novoseltsev (2007-01-15): `all_*` functions
- Andrey Novoseltsev (2008-04-01): second version, including:
  - dual NEF-partitions and necessary convex_hull and minkowski_sum
  - built-in sequences of 2- and 3-dimensional reflexive polytopes
  - plot3d, skeleton_show
- Maximilian Kreuzer and Harald Skarke: authors of PALP (which was also used to obtain the list of 3-dimensional reflexive polytopes)
- Erwin Riegler: the author of nef.x

LatticePolytope(*data, desc=None, compute_vertices=True, copy_vertices=True, n=0*)

Construct a lattice polytope.

LatticePolytope(data, [desc], [compute_vertices], [copy_vertices], [n])

INPUT:

- `data` - matrix whose columns are vertices of the polytope (unless `compute_vertices` is True); a file with matrix data, open for reading; or a filename of such a file. See `read_palp_matrix` for the file format. Points of the given matrix must span the space.
- `desc` - (default: "A lattice polytope") description of the polytope.
- `compute_vertices` - (default: True) if True, the convex hull of the given points will be computed for determining vertices. Otherwise, the given points are vertices.
- `copy_vertices` - (default: True) if False, and compute_vertices is False, and `data` is a matrix of vertices, it will be made immutable.
- `n` - (default: 0) if `data` is a name of a file, that contains data blocks for several polytopes, the n-th block will be used. *NUMERATION STARTS WITH ZERO*.

OUTPUT: a lattice polytope

EXAMPLES: Here we construct a polytope from a matrix whose columns are vertices in 3-dimensional space. In the first case a copy of the given matrix is made during construction, in the second one the matrix is made immutable and used as a matrix of vertices.

```
sage: m = matrix(ZZ, [[1, 0, 0, -1,  0,  0],
...                   [0, 1, 0,  0, -1,  0],
...                   [0, 0, 1,  0,  0, -1]])
...
sage: p = LatticePolytope(m)
sage: p
A lattice polytope: 3-dimensional, 6 vertices.
sage: m.is_mutable()
True
sage: m is p.vertices()
False
sage: p = LatticePolytope(m, compute_vertices=False, copy_vertices=False)
sage: m.is_mutable()
False
sage: m is p.vertices()
True
```

We draw a pretty picture of the polytype in 3-dimensional space:

```
sage: p.plot3d().show()
```

Now we add an extra point, which is in the interiour of the polytope...

```
sage: m = matrix(ZZ, [[1, 0, 0, -1,  0,  0, 0],
...                   [0, 1, 0,  0, -1,  0, 0],
...                   [0, 0, 1,  0,  0, -1, 0]])
...
sage: p = LatticePolytope(m, "A lattice polytope constructed from 7 points")
sage: p
A lattice polytope constructed from 7 points: 3-dimensional, 6 vertices.
```

You can suppress vertex computation for speed but this can lead to mistakes:

```
sage: p = LatticePolytope(m, "A lattice polytope with WRONG vertices",
...                          compute_vertices=False)
...
sage: p
A lattice polytope with WRONG vertices: 3-dimensional, 7 vertices.
```

Points of the given matrix must always span the space, this conditions will be checked unless you specify `compute_vertices=False` option:

```
sage: m = matrix(ZZ, [[1, 0, -1,  0],
...                   [0, 1,  0, -1],
...                   [0, 0,  0,  0]])
...
sage: p = LatticePolytope(m)
...
ValueError: Points must span the space!
Given:
[ 1  0 -1  0]
[ 0  1  0 -1]
[ 0  0  0  0]
```

class **LatticePolytopeClass**(*data, desc, compute_vertices, copy_vertices=True, n=0*)

Class of lattice/reflexive polytopes.

Use `LatticePolytope` for constructing a polytope.

dim()

Return the dimension of this polytope.

EXAMPLES: We create a 3-dimensional octahedron and check its dimension:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.dim()
3
```

distances(*point=None*)

Return the matrix of distances for this polytope or distances for the given point.

The matrix of distances m gives distances m[i,j] between the i-th facet (which is also the i-th vertex of the polar polytope in the reflexive case) and j-th point of this polytope.

If point is specified, integral distances from the point to all facets of this polytope will be computed.

EXAMPLES: The matrix of distances for a 3-dimensional octahedron:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.distances()
[0 0 2 2 2 0 1]
[2 0 2 0 2 0 1]
[0 2 2 2 0 0 1]
[2 2 2 0 0 0 1]
[0 0 0 2 2 2 1]
[2 0 0 0 2 2 1]
[0 2 0 2 0 2 1]
[2 2 0 0 0 2 1]
```

Distances from facets to the point (1,2,3):

```
sage: o.distances([1,2,3])
(1, 3, 5, 7, -5, -3, -1, 1)
```

It is OK to use RATIONAL coordinates:

```
sage: o.distances([1,2,3/2])
(-1/2, 3/2, 7/2, 11/2, -7/2, -3/2, 1/2, 5/2)
sage: o.distances([1,2,sqrt(2)])
...
TypeError: unable to convert sqrt(2) to a rational
```

edges()

Return the sequence of edges of this polytope (i.e. faces of dimension 1).

EXAMPLES: The octahedron has 12 edges:

```
sage: o = lattice_polytope.octahedron(3)
sage: len(o.edges())
12
sage: o.edges()
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2,
```

faces(*dim=None, codim=None*)

Return the sequence of faces of this polytope.

If `dim` or `codim` are specified, returns a sequence of faces of the corresponding dimension or codimension. Otherwise returns the sequence of such sequences for all dimensions.

EXAMPLES: All faces of the 3-dimensional octahedron:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.faces()
[
[[0], [1], [2], [3], [4], [5]],
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2,
[[0, 1, 5], [1, 3, 5], [0, 4, 5], [3, 4, 5], [0, 1, 2], [1, 2, 3], [0, 2, 4], [2, 3, 4]]
]
```

Its faces of dimension one (i.e., edges):

```
sage: o.faces(dim=1)
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2,
```

Its faces of codimension two (also edges):

```
sage: o.faces(codim=2)
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2,
```

It is an error to specify both dimension and codimension at the same time, even if they do agree:

```
sage: o.faces(dim=1, codim=2)
...
ValueError: Both dim and codim are given!
```

facets()

Return the sequence of facets of this polytope (i.e. faces of codimension 1).

EXAMPLES: All facets of the 3-dimensional octahedron:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.facets()
[[0, 1, 5], [1, 3, 5], [0, 4, 5], [3, 4, 5], [0, 1, 2], [1, 2, 3], [0, 2, 4], [2, 3, 4]]
```

Facets are the same as faces of codimension one:

```
sage: o.facets() is o.faces(codim=1)
True
```

index()

Return the index of this polytope in the internal database of 2- or 3-dimensional reflexive polytopes. Databases are stored in the directory of the package.

Note: The first call to this function for each dimension can take a second or so while the dictionary of all polytopes is loaded, but after that it is cached and fast.

EXAMPLES: We check what is the index of the "diamond" in the database:

```
sage: o = lattice_polytope.octahedron(2)
sage: o.index()
3
```

Note that polytopes with the same index are not necessarily the same:

```
sage: o.vertices()
[ 1  0 -1  0]
[ 0  1  0 -1]
sage: lattice_polytope.ReflexivePolytope(2,3).vertices()
[ 1  0  0 -1]
[ 0  1 -1  0]
```

But they are in the same $GL(Z^n)$ orbit and have the same normal form:

```
sage: o.normal_form()
[ 1  0  0 -1]
[ 0  1 -1  0]
sage: lattice_polytope.ReflexivePolytope(2,3).normal_form()
[ 1  0  0 -1]
[ 0  1 -1  0]
```

is_reflexive()

Return True if this polytope is reflexive.

EXAMPLES: The 3-dimensional octahedron is reflexive (and 4318 other 3-polytopes):

```
sage: o = lattice_polytope.octahedron(3)
sage: o.is_reflexive()
True
```

But not all polytopes are reflexive:

```
sage: m = matrix(ZZ, [[1, 0, 0, -1,  0,  0],
...                   [0, 1, 0,  0, -1,  0],
...                   [0, 0, 0,  0,  0, -1]])
...
sage: p = LatticePolytope(m)
```

```
sage: p.is_reflexive()
False
```

mif (*partition*)

Return all vectors $m_{i,f}$, grouped into matrices.

INPUT:

- `partition` - NEF-partition (instance of class NEFPartition)

OUTPUT: A sequence of matrices, one for each facet f of this polytope. Each row of each matrix corresponds to a part of the NEF-partition.

EXAMPLES: We compute $m_{i,f}$ matrices for one of the nef-partitions of the 3-dimensional octahedron:

```
sage: o = lattice_polytope.octahedron(3)
sage: nefp = o.nef_partitions()[0]
sage: o.mif(nefp)
[
[ 0  0  1]
[-1 -1  0],
[ 0  0  1]
[ 1 -1  0],
[ 0  1  1]
[-1  0  0],
[0 1 1]
[1 0 0],
[ 0  0 -1]
[-1 -1  0],
[ 0  0 -1]
[ 1 -1  0],
[ 0  1 -1]
[-1  0  0],
[ 0  1 -1]
[ 1  0  0]
]
```

nef_partitions (*keep_symmetric=False, keep_products=True, keep_projections=True, hodge_numbers=False*)

Return the sequence of NEF-partitions for this polytope.

INPUT:

- `keep_symmetric` - (default: False) if True, "-s" option will be passed to nef.x in order to keep symmetric partitions;
- `keep_products` - (default: True) if True, "-D" option will be passed to nef.x in order to keep product partitions;
- `keep_projections` - (default: True) if True, "-P" option will be passed to nef.x in order to keep projection partitions;
- `hodge_numbers` - (default: False) if False, "-p" option will be passed to nef.x in order to skip Hodge numbers computation, which takes a lot of time.

EXAMPLES: NEF-partitions of the 4-dimensional octahedron:

```
sage: o = lattice_polytope.octahedron(4)
sage: o.nef_partitions()
[
[1, 1, 0, 0, 1, 1, 0, 0] (direct product),
[1, 1, 1, 0, 1, 0, 0, 0],
[1, 1, 1, 0, 1, 1, 0, 0],
[1, 1, 1, 0, 1, 1, 1, 0] (direct product),
[1, 1, 1, 1, 0, 0, 0, 0],
[1, 1, 1, 1, 1, 0, 0, 0],
```

```
[1, 1, 1, 1, 1, 1, 0, 0],
[1, 1, 1, 1, 1, 1, 1, 0] (projection)
]
```

Now we omit projections:

```
sage: o.nef_partitions(keep_projections=False)
[
[1, 1, 0, 0, 1, 1, 0, 0] (direct product),
[1, 1, 1, 0, 1, 0, 0, 0],
[1, 1, 1, 0, 1, 1, 0, 0],
[1, 1, 1, 0, 1, 1, 1, 0] (direct product),
[1, 1, 1, 1, 0, 0, 0, 0],
[1, 1, 1, 1, 1, 0, 0, 0],
[1, 1, 1, 1, 1, 1, 0, 0]
]
```

At present, Hodge numbers cannot be computed for a given NEF-partition:

```
sage: o.nef_partitions()[1].hodge_numbers()
...
NotImplementedError: use nef_partitions(hodge_numbers=True)!
```

But they can be obtained from nef.x for all partitions at once. Partitions will be exactly the same:

```
sage: o.nef_partitions(hodge_numbers=True)
[
[1, 1, 0, 0, 1, 1, 0, 0] (direct product),
[1, 1, 1, 0, 1, 0, 0, 0],
[1, 1, 1, 0, 1, 1, 0, 0],
[1, 1, 1, 0, 1, 1, 1, 0] (direct product),
[1, 1, 1, 1, 0, 0, 0, 0],
[1, 1, 1, 1, 1, 0, 0, 0],
[1, 1, 1, 1, 1, 1, 0, 0],
[1, 1, 1, 1, 1, 1, 1, 0] (projection)
]
```

Now it is possible to get Hodge numbers:

```
sage: o.nef_partitions(hodge_numbers=True)[1].hodge_numbers()
[20]
```

Since NEF-partitions are cached, Hodge numbers are accessible after the first request, even if you do not specify hodge_numbers anymore:

```
sage: o.nef_partitions()[1].hodge_numbers()
[20]
```

We illustrate removal of symmetric partitions on a diamond:

```
sage: o = lattice_polytope.octahedron(2)
sage: o.nef_partitions()
[
[1, 0, 1, 0] (direct product),
[1, 1, 0, 0],
[1, 1, 1, 0] (projection)
]
sage: o.nef_partitions(keep_symmetric=True)
[
[1, 1, 0, 1] (projection),
[1, 0, 1, 1] (projection),
[1, 0, 0, 1],
```

```
[0, 1, 1, 1] (projection),
[0, 1, 0, 1] (direct product),
[0, 0, 1, 1],
[1, 1, 1, 0] (projection)
]
```

NEF-partitions can be computed only for reflexive polytopes:

```
sage: m = matrix(ZZ, [[1, 0, 0, -1,  0,  0],
...                   [0, 1, 0,  0, -1,  0],
...                   [0, 0, 2,  0,  0, -1]])
...
sage: p = LatticePolytope(m)
sage: p.nef_partitions()
...
ValueError: The given polytope is not reflexive!
Polytope: A lattice polytope: 3-dimensional, 6 vertices.
```

nef_x(*keys*)

Run nef.x wilth given `keys` on vertices of this polytope.

INPUT:

- `keys` - a string of options passed to nef.x. The key "-f" is added automatically.

OUTPUT: the output of nef.x as a string.

EXAMPLES: This call is used internally for computing NEF-partitions:

```
sage: o = lattice_polytope.octahedron(3)
sage: s = o.nef_x("-N -V -p")
sage: s                        # output contains random time
M:27 8 N:7 6  codim=2 #part=5
3 6  Vertices of P:
    1    0    0   -1    0    0
    0    1    0    0   -1    0
    0    0    1    0    0   -1
 P:0 V:2 4 5       0sec  0cpu
 P:2 V:3 4 5       0sec  0cpu
 P:3 V:4 5       0sec  0cpu
np=3 d:1 p:1    0sec     0cpu
```

nfacets()

Return the number of facets of this polytope.

EXAMPLES: The number of facets of the 3-dimensional octahedron:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.nfacets()
8
```

normal_form()

Return the normal form of vertices of the polytope. Two lattice polytopes are in the same GL(Z)-orbit if and only if their normal forms are the same.

EXAMPLES: We compute the normal form of the "diamond":

```
sage: o = lattice_polytope.octahedron(2)
sage: o.vertices()
[ 1  0 -1  0]
[ 0  1  0 -1]
sage: o.normal_form()
[ 1  0  0 -1]
[ 0  1 -1  0]
```

The diamond is the 3rd polytope in the internal database...

```
sage: o.index()
3
sage: lattice_polytope.ReflexivePolytope(2,3).vertices()
[ 1  0  0 -1]
[ 0  1 -1  0]
```

npoints()

Return the number of lattice points of this polytope.

EXAMPLES: The number of lattice points of the 3-dimensional octahedron and its polar cube:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.npoints()
7
sage: cube = o.polar()
sage: cube.npoints()
27
```

nvertices()

Return the number of vertices of this polytope.

EXAMPLES: The number of vertices of the 3-dimensional octahedron and its polar cube:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.nvertices()
6
sage: cube = o.polar()
sage: cube.nvertices()
8
```

parent()

Return the set of all lattice polytopes.

EXAMPLES:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.parent()
Set of all Lattice Polytopes
```

plot3d(*show_facets=True, facet_opacity=0.5, facet_color=(0, 1, 0), show_edges=True, edge_thickness=3, edge_color=(0.5, 0.5, 0.5), show_vertices=True, vertex_size=10, vertex_color=(1, 0, 0), show_points=True, point_size=10, point_color=(0, 0, 1), show_vindices=None, vindex_color=(0, 0, 0), show_pindices=None, pindex_color=(0, 0, 0), index_shift=1.1000000000000001*)

Return a 3d-plot of a 3-dimensional polytope.

By default, everything is shown with more or less pretty combination of size and color parameters.

INPUT: Most of the parameters are self-explanatory:

- `show_facets` - (default:True)
- `facet_opacity` - (default:0.5)
- `facet_color` - (default:(0,1,0))
- `show_edges` - (default:True) whether to draw edges as lines
- `edge_thickness` - (default:3)
- `edge_color` - (default:(0.5,0.5,0.5))
- `show_vertices` - (default:True) whether to draw vertices as balls
- `vertex_size` - (default:10)
- `vertex_color` - (default:(1,0,0))
- `show_points` - (default:True) whether to draw other poits as balls
- `point_size` - (default:10)

- point_color - (default:(0,0,1))
- show_vindices - (default:same as show_vertices) whether to show indices of vertices
- vindex_color - (default:(0,0,0)) color for vertex labels
- show_pindices - (default:same as show_points) whether to show indices of other points
- pindex_color - (default:(0,0,0)) color for point labels
- index_shift - (default:1.1)) if 1, labels are placed exactly at the corresponding points. Otherwise the label position is computed as a multiple of the point position vector.

EXAMPLES: The default plot of a cube:

```
sage: c = lattice_polytope.octahedron(3).polar()
sage: c.plot3d()
```

Plot without facets and points, shown without the frame:

```
sage: c.plot3d(show_facets=false,show_points=false).show(frame=False)
```

TESTS:

```
sage: m = matrix([[0,0,0],[0,1,1],[1,0,1],[1,1,0]]).transpose()
sage: p = LatticePolytope(m, compute_vertices=True)
sage: p.plot3d()
```

point(*i*)

Return the i-th point of this polytope, i.e. the i-th column of the matrix returned by points().

EXAMPLES: First few points are actually vertices:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.vertices()
[ 1  0  0 -1  0  0]
[ 0  1  0  0 -1  0]
[ 0  0  1  0  0 -1]
sage: o.point(1)
(0, 1, 0)
```

The only other point in the octahedron is the origin:

```
sage: o.point(6)
(0, 0, 0)
sage: o.points()
[ 1  0  0 -1  0  0  0]
[ 0  1  0  0 -1  0  0]
[ 0  0  1  0  0 -1  0]
```

points()

Return all lattice points of this polytope as columns of a matrix.

EXAMPLES: The lattice points of the 3-dimensional octahedron and its polar cube:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.points()
[ 1  0  0 -1  0  0  0]
[ 0  1  0  0 -1  0  0]
[ 0  0  1  0  0 -1  0]
sage: cube = o.polar()
sage: cube.points()
[-1  1 -1  1 -1  1 -1  1 -1 -1 -1 -1 -1  0  0  0  0  0  0  0  0  1  1  1  1  1]
[-1 -1  1  1 -1 -1  1  1 -1  0  0  0  1 -1 -1 -1  0  0  0  1  1  1 -1  0  0  0  1]
[ 1  1  1  1 -1 -1 -1 -1  0 -1  0  1  0 -1  0  1 -1  0  1 -1  0  1  0 -1  0  1  0]
```

polar()

Return the polar polytope, if this polytope is reflexive.

EXAMPLES: The polar polytope to the 3-dimensional octahedron:

```
sage: o = lattice_polytope.octahedron(3)
sage: cube = o.polar()
sage: cube
A polytope polar to An octahedron: 3-dimensional, 8 vertices.
```

The polar polytope "remembers" the original one:

```
sage: cube.polar()
An octahedron: 3-dimensional, 6 vertices.
sage: cube.polar().polar() is cube
True
```

Only reflexive polytopes have polars:

```
sage: m = matrix(ZZ, [[1, 0, 0, -1,  0,  0],
...                   [0, 1, 0,  0, -1,  0],
...                   [0, 0, 2,  0,  0, -1]])
...
sage: p = LatticePolytope(m)
sage: p.polar()
...
ValueError: The given polytope is not reflexive!
Polytope: A lattice polytope: 3-dimensional, 6 vertices.
```

poly_x(*keys*)

Run poly.x with given `keys` on vertices of this polytope.

INPUT:

- `keys` - a string of options passed to poly.x. The key "f" is added automatically.

OUTPUT: the output of poly.x as a string.

EXAMPLES: This call is used for determining if a polytope is reflexive or not:

```
sage: o = lattice_polytope.octahedron(3)
sage: print o.poly_x("e")
8 3  Vertices of P-dual <-> Equations of P
  -1  -1   1
   1  -1   1
  -1   1   1
   1   1   1
  -1  -1  -1
   1  -1  -1
  -1   1  -1
   1   1  -1
```

Since PALP has limits on different parameters determined during compilation, the following code is likely to fail, unless you change default settings of PALP:

```
sage: BIGO = lattice_polytope.octahedron(7)
sage: BIGO
An octahedron: 7-dimensional, 14 vertices.
sage: BIGO.poly_x("e")      # possibly different output depending on your system
...
ValueError: Error executing "poly.x -fe" for the given polytope!
Polytope: An octahedron: 7-dimensional, 14 vertices.
Vertices:
[ 1  0  0  0  0  0  0 -1  0  0  0  0  0  0]
[ 0  1  0  0  0  0  0  0 -1  0  0  0  0  0]
[ 0  0  1  0  0  0  0  0  0 -1  0  0  0  0]
[ 0  0  0  1  0  0  0  0  0  0 -1  0  0  0]
```

```
[ 0  0  0  0  1  0  0  0  0  0  0 -1  0  0]
[ 0  0  0  0  0  1  0  0  0  0  0  0 -1  0]
[ 0  0  0  0  0  0  1  0  0  0  0  0  0 -1]
Output:
increase POLY_Dmax!
```

show3d()

Show a 3d picture of the polytope with default settings and without axes or frame.

See self.plot3d? for more details.

EXAMPLES:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.show3d()
```

skeleton()

Return the graph of the one-skeleton of this polytope.

EXAMPLES: We construct the one-skeleton graph for the "diamond":

```
sage: o = lattice_polytope.octahedron(2)
sage: g = o.skeleton()
sage: g
Graph on 4 vertices
sage: g.edges()
[(0, 1, None), (0, 3, None), (1, 2, None), (2, 3, None)]
```

skeleton_points(*k=1*)

Return the increasing list of indices of lattice points in k-skeleton of the polytope (k is 1 by default).

EXAMPLES: We compute all skeleton points for the cube:

```
sage: o = lattice_polytope.octahedron(3)
sage: c = o.polar()
sage: c.skeleton_points()
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 15, 19, 21, 22, 23, 25, 26]
```

The default was 1-skeleton:

```
sage: c.skeleton_points(k=1)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 15, 19, 21, 22, 23, 25, 26]
```

0-skeleton just lists all vertices:

```
sage: c.skeleton_points(k=0)
[0, 1, 2, 3, 4, 5, 6, 7]
```

2-skeleton lists all points except for the origin (point #17):

```
sage: c.skeleton_points(k=2)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 25, 2
```

3-skeleton includes all points:

```
sage: c.skeleton_points(k=3)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 2
```

It is OK to compute higher dimensional skeletons - you will get the list of all points:

```
sage: c.skeleton_points(k=100)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 2
```

skeleton_show(*normal=None*)

Show the graph of one-skeleton of this polytope. Works only for 3-dimensional polytopes.

INPUT:

- `normal` - a 3-dimensional vector (can be given as a list), which should be perpendicular to the screen. If not given, will be selected randomly (new each time and it may be far from "nice").

EXAMPLES: Show a pretty picture of the octahedron:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.skeleton_show([1,2,4])
```

vertex(*i*)

Return the i-th vertex of this polytope, i.e. the i-th column of the matrix returned by vertices().

EXAMPLES: Note that numeration starts with zero:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.vertices()
[ 1  0  0 -1  0  0]
[ 0  1  0  0 -1  0]
[ 0  0  1  0  0 -1]
sage: o.vertex(3)
(-1, 0, 0)
```

vertices()

Return vertices of this polytope as columns of a matrix.

EXAMPLES: The lattice points of the 3-dimensional octahedron and its polar cube:

```
sage: o = lattice_polytope.octahedron(3)
sage: o.vertices()
[ 1  0  0 -1  0  0]
[ 0  1  0  0 -1  0]
[ 0  0  1  0  0 -1]
sage: cube = o.polar()
sage: cube.vertices()
[-1  1 -1  1 -1  1 -1  1]
[-1 -1  1  1 -1 -1  1  1]
[ 1  1  1  1 -1 -1 -1 -1]
```

class NEFPartition(*data, polytope=None*)

Construct a NEF-partition from the given list. If the second argument is given, it must be the polytope of the NEF-partition.

A NEF-partition with k parts, $V = V_0 \cap V_1 \cap \cdots \cap V_{k-1}$, is represented by a single list of lenght $n = \#V$, in which the i-th entry is the part number of the i-th vertex of a polytope.

EXAMPLES: All elements of the list will be coerced to integers, so it is OK to use either a list of numbers or a list of strings:

```
sage: lattice_polytope.NEFPartition([1, 1, 0, 0, 0, 1])
[1, 1, 0, 0, 0, 1]
sage: lattice_polytope.NEFPartition(['1', '1', '0', '0', '0', '1'])
[1, 1, 0, 0, 0, 1]
```

dual()

Compute dual nef-partition.

Computation follows the article "Combinatorial aspects of mirror symmetry" by V.Batyrev and B.Nill, initial polytope corresponds to Nabla*, output to Delta*.

EXAMPLES: We compute the dual nef-partition for the first nef-partition of the octahedron:

```
sage: o = lattice_polytope.octahedron(3)
sage: np = o.nef_partitions()[0]
sage: np
[1, 1, 0, 1, 0, 0]
```

```
sage: npd = np.dual()
sage: npd
[1, 0, 1, 0, 0, 1, 0, 1]
sage: npd.polytope().vertices()
[ 1  0  1  0  0 -1  0 -1]
[-1  1  0  0  0 -1  1  0]
[ 0  1  0  1 -1  0 -1  0]
```

hodge_numbers()

Return Hodge numbers corresponding to the given nef-partition.

Note: In order to use this function you must use `nef_partitions(hodge_numbers=True)` for computing nef-partitions.

EXAMPLE: Currently, you need to request hodge_numbers when you compute nef-partitions. (The following weird construction ensures that there are no cached numbers.)

```
sage: o = LatticePolytope(lattice_polytope.octahedron(4).vertices())
sage: np = o.nef_partitions()[0]
sage: np.hodge_numbers()
...
NotImplementedError: use nef_partitions(hodge_numbers=True)!
sage: np = o.nef_partitions(hodge_numbers=True)[0]
sage: np.hodge_numbers()
[4]
```

nparts()

Return the number of parts of this partitions.

EXAMPLES:

```
sage: nefp = lattice_polytope.NEFPartition([1, 1, 0, 0, 0, 1])
sage: nefp.nparts()
2
```

part(*i*)

Return the `i`-th part of the partition.

NUMERATON OF PARTS STARTS WITH ZERO.

EXAMPLES:

```
sage: nefp = lattice_polytope.NEFPartition([1, 1, 0, 0, 0, 1])
sage: nefp.part(0)
[2, 3, 4]
sage: nefp.part(1)
[0, 1, 5]
```

part_of_vertex(*i*)

Return the index of the part containing the `i`-vertex.

EXAMPLES: `nefp.part_of_vertex(i)` is equivalent to `nefp[i]`.

```
sage: nefp = lattice_polytope.NEFPartition([1, 1, 0, 0, 0, 1])
sage: nefp.part_of_vertex(3)
0
sage: nefp.part_of_vertex(5)
1
sage: nefp[3]
0
sage: nefp[5]
1
```

You cannot change a NEF-partition once it is constructed:

```
sage: nefp[3] = 1
...
ValueError: object is immutable; please change a copy instead.
```

polytope()

Return the polytope of this NEF-partition.

EXAMPLE:

```
sage: o = lattice_polytope.octahedron(3)
sage: np = o.nef_partitions()[0]
sage: np.polytope() == o
True
```

ReflexivePolytope(*dim, n*)

Return n-th reflexive polytope from the database of 2- or 3-dimensional reflexive polytopes.

Note:

1. Numeration starts with zero: `0=n=15` for `dim=2` and `0=n=4318` for `dim=3`.
2. During the first call, all reflexive polytopes of requested dimension are loaded and cached for future use, so the first call for 3-dimensional polytopes can take several seconds, but all consecutive calls are fast.
3. Equivalent to `ReflexivePolytopes(dim)[n]` but checks bounds first.

EXAMPLES: The 3rd 2-dimensional polytope is "the diamond:"

```
sage: ReflexivePolytope(2,3)
Reflexive polygon  3: 2-dimensional, 4 vertices.
sage: lattice_polytope.ReflexivePolytope(2,3).vertices()
[ 1  0  0 -1]
[ 0  1 -1  0]
```

There are 16 reflexive polygons and numeration starts with 0:

```
sage: ReflexivePolytope(2,16)
...
ValueError: there are only 16 reflexive polygons!
```

It is not possible to load a 4-dimensional polytope in this way:

```
sage: ReflexivePolytope(4,16)
...
NotImplementedError: only 2- and 3-dimensional reflexive polytopes are available!
```

ReflexivePolytopes(*dim*)

Return the sequence of all 2- or 3-dimensional reflexive polytopes.

Note: During the first call the database is loaded and cached for future use, so repetitive calls will return the same object in memory.

TODO: load a database with precomputed faces. Requires better pickling, otherwise takes forever!

EXAMPLES: There are 16 reflexive polygons:

```
sage: len(ReflexivePolytopes(2))
16
```

It is not possible to load 4-dimensional polytopes in this way:

```
sage: ReflexivePolytopes(4)
...
NotImplementedError: only 2- and 3-dimensional reflexive polytopes are available!
```

class SetOfAllLatticePolytopesClass()

all_cached_data(*polytopes*)

Compute all cached data for all given `polytopes` and their polars.

This functions does it MUCH faster than member functions of `LatticePolytope` during the first run. So it is recommended to use this functions if you work with big sets of data. None of the polytopes in the given sequence should be constructed as the polar polytope to another one.

INPUT: a sequence of lattice polytopes.

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.octahedron(3)
sage: lattice_polytope.all_cached_data([o])
sage: o.faces()
[
[[0], [1], [2], [3], [4], [5]],
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2, 4]]
[[0, 1, 5], [1, 3, 5], [0, 4, 5], [3, 4, 5], [0, 1, 2], [1, 2, 3], [0, 2, 4], [2, 3, 4]]
]
```

However, you cannot use it for polytopes that are constructed as polar polytopes of others:

```
sage: lattice_polytope.all_cached_data([o.polar()])
...
ValueError: Cannot read face structure for a polytope constructed as polar, use _compute_faces!
```

all_faces(*polytopes*)

Compute faces for all given `polytopes`.

This functions does it MUCH faster than member functions of `LatticePolytope` during the first run. So it is recommended to use this functions if you work with big sets of data.

INPUT: a sequence of lattice polytopes.

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.octahedron(3)
sage: lattice_polytope.all_faces([o])
sage: o.faces()
[
[[0], [1], [2], [3], [4], [5]],
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2, 4]]
[[0, 1, 5], [1, 3, 5], [0, 4, 5], [3, 4, 5], [0, 1, 2], [1, 2, 3], [0, 2, 4], [2, 3, 4]]
]
```

However, you cannot use it for polytopes that are constructed as polar polytopes of others:

```
sage: lattice_polytope.all_faces([o.polar()])
...
ValueError: Cannot read face structure for a polytope constructed as polar, use _compute_faces!
```

all_nef_partitions (*polytopes, keep_symmetric=False*)

Compute NEF-partitions for all given `polytopes`.

This functions does it MUCH faster than member functions of `LatticePolytope` during the first run. So it is recommended to use this functions if you work with big sets of data.

Note: member function `is_reflexive` will be called separately for each polytope. It is strictly recommended to call `all_polars` on the sequence of `polytopes` before using this function.

INPUT: a sequence of lattice polytopes.

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.octahedron(3)
sage: lattice_polytope.all_nef_partitions([o])
sage: o.nef_partitions()
[
[1, 1, 0, 1, 0, 0],
[1, 1, 1, 0, 0, 0],
[1, 1, 1, 1, 0, 0]
]
```

You cannot use this function for non-reflexive polytopes:

```
sage: m = matrix(ZZ, [[1, 0, 0, -1,  0,  0],
...                   [0, 1, 0,  0, -1,  0],
...                   [0, 0, 2,  0,  0, -1]])
...
sage: p = LatticePolytope(m)
sage: lattice_polytope.all_nef_partitions([o, p])
...
ValueError: The given polytope is not reflexive!
Polytope: A lattice polytope: 3-dimensional, 6 vertices.
```

all_points (*polytopes*)

Compute lattice points for all given `polytopes`.

This functions does it MUCH faster than member functions of `LatticePolytope` during the first run. So it is recommended to use this functions if you work with big sets of data.

INPUT: a sequence of lattice polytopes.

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.octahedron(3)
sage: lattice_polytope.all_points([o])
sage: o.points()
[ 1  0  0 -1  0  0  0]
[ 0  1  0  0 -1  0  0]
[ 0  0  1  0  0 -1  0]
```

all_polars (*polytopes*)

Compute polar polytopes for all reflexive and equations of facets for all non-reflexive `polytopes`.

This functions does it MUCH faster than member functions of `LatticePolytope` during the first run. So it is recommended to use this functions if you work with big sets of data.

INPUT: a sequence of lattice polytopes.

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.octahedron(3)
sage: lattice_polytope.all_polars([o])
sage: o.polar()
A polytope polar to An octahedron: 3-dimensional, 8 vertices.
```

convex_hull (*points*)

Compute the convex hull of the given points.

Note: `points` might not span the space. Also, it fails for large numbers of vertices in dimensions 4 or greater

INPUT:

- `points` - a list that can be converted into vectors of the same dimension over ZZ.

OUTPUT: list of vertices of the convex hull of the given points (as vectors).

EXAMPLES: Let's compute the convex hull of several points on a line in the plane:

```
sage: lattice_polytope.convex_hull([[1,2],[3,4],[5,6],[7,8]])
[(1, 2), (7, 8)]
```

filter_polytopes (*f, polytopes, subseq=None, print_numbers=False*)

Use the function `f` to filter polytopes in a list.

INPUT:

- `f` - filtering function, it must take one argument, a lattice polytope, and return True or False.
- `polytopes` - list of polytopes.
- `subseq` - (default: None) list of integers. If it is specified, only polytopes with these numbers will be considered.
- `print_numbers` - (default: False) if True, the number of the current polytope will be printed on the screen before calling `f`.

OUTPUT: a list of integers – numbers of polytopes in the given list, that satisfy the given condition (i.e. function `f` returns True) and are elements of subseq, if it is given.

EXAMPLES: Consider a sequence of octahedrons:

```
sage: polytopes = Sequence([lattice_polytope.octahedron(n) for n in range(2, 7)], cr=True)
sage: polytopes
[
An octahedron: 2-dimensional, 4 vertices.,
An octahedron: 3-dimensional, 6 vertices.,
An octahedron: 4-dimensional, 8 vertices.,
An octahedron: 5-dimensional, 10 vertices.,
An octahedron: 6-dimensional, 12 vertices.
]
```

This filters octahedrons of dimension at least 4:

```
sage: lattice_polytope.filter_polytopes(lambda p: p.dim() >= 4, polytopes)
[2, 3, 4]
```

For long tests you can see the current progress:

```
sage: lattice_polytope.filter_polytopes(lambda p: p.nvertices() >= 10, polytopes, print_numbers=
0
1
2
```

```
3
4
[3, 4]
```

Here we consider only some of the polytopes:

```
sage: lattice_polytope.filter_polytopes(lambda p: p.nvertices() >= 10, polytopes, [2, 3, 4], pri
2
3
4
[3, 4]
```

minkowski_sum(*points1, points2*)

Compute the Minkowski sum of two convex polytopes.

Note: Polytopes might not be of maximal dimension.

INPUT:

- `points1, points2` - lists of objects that can be converted into vectors of the same dimension, treated as vertices of two polytopes.

OUTPUT: list of vertices of the Minkowski sum, given as vectors.

EXAMPLES: Let's compute the Minkovski sum of two line segments:

```
sage: lattice_polytope.minkowski_sum([[1,0],[-1,0]],[[0,1],[0,-1]])
[(1, 1), (1, -1), (-1, 1), (-1, -1)]
```

octahedron(*dim*)

Return an octahedron of the given dimension.

EXAMPLES: Here are 3- and 4-dimensional octahedrons:

```
sage: o = lattice_polytope.octahedron(3)
sage: o
An octahedron: 3-dimensional, 6 vertices.
sage: o.vertices()
[ 1  0  0 -1  0  0]
[ 0  1  0  0 -1  0]
[ 0  0  1  0  0 -1]
sage: o = lattice_polytope.octahedron(4)
sage: o
An octahedron: 4-dimensional, 8 vertices.
sage: o.vertices()
[ 1  0  0  0 -1  0  0  0]
[ 0  1  0  0  0 -1  0  0]
[ 0  0  1  0  0  0 -1  0]
[ 0  0  0  1  0  0  0 -1]
```

There exists only one octahedron of each dimension:

```
sage: o is lattice_polytope.octahedron(4)
True
```

positive_integer_relations(*points*)

Return relations between given points.

INPUT:

- `points` - lattice points given as columns of a matrix

OUTPUT: matrix of relations between given points with non-negative integer coefficients

EXAMPLES: This is a 3-dimensional reflexive polytope:

```
sage: m = matrix(ZZ,[[1, 0, -1, 0, -1],
...                  [0, 1, -1, 0,  0],
...                  [0, 0,  0, 1, -1]])
...
sage: p = LatticePolytope(m)
sage: p.points()
[ 1  0 -1  0 -1  0]
[ 0  1 -1  0  0  0]
[ 0  0  0  1 -1  0]
```

We can compute linear relations between its points in the following way:

```
sage: p.points().transpose().kernel().echelonized_basis_matrix()
[ 1  0  0  1  1  0]
[ 0  1  1 -1 -1  0]
[ 0  0  0  0  0  1]
```

However, the above relations may contain negative and rational numbers. This function transforms them in such a way, that all coefficients are non-negative integers:

```
sage: lattice_polytope.positive_integer_relations(p.points())
[1 0 0 1 1 0]
[1 1 1 0 0 0]
[0 0 0 0 0 1]
```

projective_space(*dim*)

Return a simplex of the given dimension, corresponding to P_{dim}.

EXAMPLES: We construct 3- and 4-dimensional simplexes:

```
sage: p = lattice_polytope.projective_space(3)
sage: p
A simplex: 3-dimensional, 4 vertices.
sage: p.vertices()
[ 1  0  0 -1]
[ 0  1  0 -1]
[ 0  0  1 -1]
sage: p = lattice_polytope.projective_space(4)
sage: p
A simplex: 4-dimensional, 5 vertices.
sage: p.vertices()
[ 1  0  0  0 -1]
[ 0  1  0  0 -1]
[ 0  0  1  0 -1]
[ 0  0  0  1 -1]
```

read_all_polytopes(*file_name, desc=None*)

Read all polytopes from the given file.

INPUT:

- `file_name` - the name of a file with VERTICES of polytopes
- `desc` - a string, that will be used for creating polytope descriptions. By default it will be set to 'A lattice polytope #%d from "filename"' + and will be used as `desc % n` where `n` is the number of the polytope in the file (*STARTING WITH ZERO*).

OUTPUT: a sequence of polytopes

EXAMPLES: We use poly.x to compute polar polytopes of 2- and 3-octahedrons and read them:

```
sage: o2 = lattice_polytope.octahedron(2)
sage: o3 = lattice_polytope.octahedron(3)
sage: result_name = lattice_polytope._palp("poly.x -fe", [o2, o3])
sage: f = open(result_name)
sage: f.readlines()
['4 2  Vertices of P-dual <-> Equations of P\n', '  -1   1\n', '   1   1\n', '  -1  -1\n', '   1
sage: f.close()
sage: lattice_polytope.read_all_polytopes(result_name, desc="Polytope from file %d")
[
Polytope from file 0: 2-dimensional, 4 vertices.,
Polytope from file 1: 3-dimensional, 8 vertices.
]
sage: os.remove(result_name)
```

read_palp_matrix(*data*)

Read and return an integer matrix from a string or an opened file.

First input line must start with two integers m and n, the number of rows and columns of the matrix. The rest of the first line is ignored. The next m lines must contain n numbers each.

If m>n, returns the transposed matrix. If the string is empty or EOF is reached, returns the empty matrix, constructed by `matrix()`.

EXAMPLES:

```
sage: lattice_polytope.read_palp_matrix("2 3 comment \n 1 2 3 \n 4 5 6")
[1 2 3]
[4 5 6]
sage: lattice_polytope.read_palp_matrix("3 2 Will be transposed \n 1 2 \n 3 4 \n 5 6")
[1 3 5]
[2 4 6]
```

sage_matrix_to_maxima(*m*)

Convert a Sage matrix to the string representation of Maxima.

EXAMPLE:

```
sage: m = matrix(ZZ,2)
sage: lattice_polytope.sage_matrix_to_maxima(m)
matrix([0,0],[0,0])
```

skip_palp_matrix(*data, n=1*)

Skip matrix data in a file.

INPUT:

- `data` - opened file with blocks of matrix data in the following format: A block consisting of m+1 lines has the number m as the first element of its first line.
- `n` - (default: 1) integer, specifies how many blocks should be skipped

If EOF is reached during the process, raises ValueError exception.

EXAMPLE: We create a file with vertices of the square and the cube, but read only the second set:

```
sage: o2 = lattice_polytope.octahedron(2)
sage: o3 = lattice_polytope.octahedron(3)
sage: result_name = lattice_polytope._palp("poly.x -fe", [o2, o3])
sage: f = open(result_name)
```

```
sage: f.readlines()
['4 2  Vertices of P-dual <-> Equations of P\n', '  -1   1\n', '   1   1\n', '  -1  -1\n', '   1
sage: f.close()
sage: f = open(result_name)
sage: lattice_polytope.skip_palp_matrix(f)
sage: lattice_polytope.read_palp_matrix(f)
[-1  1 -1  1 -1  1 -1  1]
[-1 -1  1  1 -1 -1  1  1]
[ 1  1  1  1 -1 -1 -1 -1]
sage: f.close()
sage: os.remove(result_name)
```

write_palp_matrix(*m, ofile=None, comment='', format=None*)

Write a matrix into a file.

INPUT:

- `m` - a matrix over integers.
- `ofile` - a file opened for writing (default: stdout)
- `comment` - a string (default: empty) see output description
- `format` - a format string used to print matrix entries. By default, "

OUTPUT: First line: number_of_rows number_of_columns comment Next number_of_rows lines: rows of the matrix.

EXAMPLES: This functions is used for writing polytope vertices in PALP format:

```
sage: o = lattice_polytope.octahedron(3)
sage: lattice_polytope.write_palp_matrix(o.vertices(), comment="3D Octahedron")
3 6 3D Octahedron
 1  0  0 -1  0  0
 0  1  0  0 -1  0
 0  0  1  0  0 -1
sage: lattice_polytope.write_palp_matrix(o.vertices(), format="%4d")
3 6
   1    0    0   -1    0    0
   0    1    0    0   -1    0
   0    0    1    0    0   -1
```

34.2 Groebner Fans

Sage provides much of the functionality of gfan, which is a software package whose main function is to enumerate all reduced Groebner bases of a polynomial ideal. The reduced Groebner bases yield the maximal cones in the Groebner fan of the ideal. Several subcomputations can be issued and additional tools are included. Among these the highlights are:

- Commands for computing tropical varieties.
- Interactive walks in the Groebner fan of an ideal.
- Commands for graphical renderings of Groebner fans and monomial ideals.

AUTHORS:

- Anders Nedergaard Jensen: Wrote the gfan C++ program, which implements algorithms many of which were invented by Jensen, Komei Fukuda, and Rekha Thomas. All the underlying hard work of the Groebner fans functionality of Sage depends on this C++ program.
- William Stein (2006-04-20): Wrote first version of the Sage code for working with Groebner fans.
- Tristram Bogart: the design of the Sage interface to gfan is joint work with Tristram Bogart, who also supplied numerous examples.
- Marshall Hampton (2008-03-25): Rewrote various functions to use gfan-0.3. This is still a work in progress, comments are appreciated on sage-devel@googlegroups.com (or personally at hamptonio@gmail.com).

EXAMPLES:

```
sage: x,y = QQ['x,y'].gens()
sage: i = ideal(x^2 - y^2 + 1)
sage: g = i.groebner_fan()
sage: g.reduced_groebner_bases()
[[x^2 - y^2 + 1], [-x^2 + y^2 - 1]]
```

TESTS:

```
sage: x,y = QQ['x,y'].gens()
sage: i = ideal(x^2 - y^2 + 1)
sage: g = i.groebner_fan()
sage: g == loads(dumps(g))
True
```

REFERENCES:

- Anders N. Jensen; Gfan, a software system for Groebner fans; available at http://www.math.tu-berlin.de/~jensen/software/gfan/gfan.html

class GroebnerFan(*I, is_groebner_basis=False, symmetry=None, verbose=False*)

buchberger()

Computes and returns a lexicographic reduced Groebner basis for the ideal.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([x - z^3, y^2 - x + x^2 - z^3*x]).groebner_fan()
sage: G.buchberger()
[-z^3 + y^2, -z^3 + x]
```

characteristic()

Return the characteristic of the base ring.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: i1 = ideal(x*z + 6*y*z - z^2, x*y + 6*x*z + y*z - z^2, y^2 + x*z + y*z)
sage: gf = i1.groebner_fan()
sage: gf.characteristic()
0
```

dimension_of_homogeneity_space()

Return the dimension of the homogeneity space.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.dimension_of_homogeneity_space()
0
```

gfan(*cmd=”, I=None, format=True*)

Returns the gfan output as a string given an input cmd; the default is to produce the list of reduced Groebner bases in gfan format.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: gf = R.ideal([x^3-y,y^3-x-1]).groebner_fan()
sage: gf.gfan()
'Q[x,y]\n{{\ny^9-1-y+3*y^3-3*y^6,\nx+1-y^3}\n,\n{\ny^3-1-x,\nx^3-y}\n,\n{\ny-x^3,\nx^9-1-x}\
```

homogeneity_space()

Return the homogeneity space of a the list of polynomials that define this Groebner fan.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: H = G.homogeneity_space()
```

ideal()

Return the ideal the was used to define this Groebner fan.

EXAMPLES:

```
sage: R.<x1,x2> = PolynomialRing(QQ,2)
sage: gf = R.ideal([x1^3-x2,x2^3-2*x1-2]).groebner_fan()
sage: gf.ideal()
Ideal (x1^3 - x2, x2^3 - 2*x1 - 2) of Multivariate Polynomial Ring in x1, x2 over Rational F
```

interactive(**args, **kwds*)

See the documentation for self[0].interactive(). This does not work with the notebook.

EXAMPLES:

```
sage: print "This is not easily doc-testable; please write a good one!"
This is not easily doc-testable; please write a good one!
```

maximal_total_degree_of_a_groebner_basis()

Return the maximal total degree of any Groebner basis.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.maximal_total_degree_of_a_groebner_basis()
4
```

minimal_total_degree_of_a_groebner_basis()

Return the minimal total degree of any Groebner basis.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.minimal_total_degree_of_a_groebner_basis()
2
```

number_of_reduced_groebner_bases()

Return the number of reduced Groebner bases.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.number_of_reduced_groebner_bases()
3
```

number_of_variables()

Return the number of variables.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.number_of_variables()
2

sage: R = PolynomialRing(QQ,'x',10)
sage: R.inject_variables(globals())
Defining x0, x1, x2, x3, x4, x5, x6, x7, x8, x9
sage: G = ideal([x0 - x9, sum(R.gens())]).groebner_fan()
sage: G.number_of_variables()
10
```

polyhedralfan()

Returns a polyhedral fan object corresponding to the reduced Groebner bases.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-1]).groebner_fan()
sage: pf = gf.polyhedralfan()
sage: pf.rays()
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
```

reduced_groebner_bases()

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: G = R.ideal([x^2*y - z, y^2*z - x, z^2*x - y]).groebner_fan()
sage: X = G.reduced_groebner_bases()
sage: len(X)
33
sage: X[0]
[z^15 - z, y - z^11, x - z^9]
sage: X[0].ideal()
Ideal (z^15 - z, y - z^11, x - z^9) of Multivariate Polynomial Ring in x, y, z over Rational
sage: X[:5]
[[z^15 - z, y - z^11, x - z^9],
[-y + z^11, y*z^4 - z, y^2 - z^8, x - z^9],
[-y^2 + z^8, y*z^4 - z, y^2*z^3 - y, y^3 - z^5, x - y^2*z],
[-y^3 + z^5, y*z^4 - z, y^2*z^3 - y, y^4 - z^2, x - y^2*z],
[-y^4 + z^2, y^6*z - y, y^9 - z, x - y^2*z]]
sage: R3.<x,y,z> = PolynomialRing(GF(2477),3)
sage: gf = R3.ideal([300*x^3-y,y^2-z,z^2-12]).groebner_fan()
sage: gf.reduced_groebner_bases()
[[z^2 - 12, y^2 - z, x^3 + 933*y],
[-y^2 + z, y^4 - 12, x^3 + 933*y],
[z^2 - 12, -300*x^3 + y, x^6 - 1062*z],
[-828*x^6 + z, -300*x^3 + y, x^12 + 200]]
```

render(*file=None, larger=False, shift=0, rgbcolor=(0, 0, 0), polyfill=<function max_degree at 0x8fdcde8>, scale_colors=True*)

Render a Groebner fan as sage graphics or save as an xfig file.

More precisely, the output is a drawing of the Groebner fan intersected with a triangle. The corners of the triangle are (1,0,0) to the right, (0,1,0) to the left and (0,0,1) at the top. If there are more than three variables in the ring we extend these coordinates with zeros.

INPUT:

- `file` - a filename if you prefer the output saved to a file. This will be in xfig format.
- `shift` - shift the positions of the variables in the drawing. For example, with shift=1, the corners will be b (right), c (left), and d (top). The shifting is done modulo the number of variables in the polynomial ring. The default is 0.
- `larger` - bool (default: False); if True, make the triangle larger so that the shape of of the Groebner region appears. Affects the xfig file but probably not the sage graphics (?)
- `rgbcolor` - This will not affect the saved xfig file, only the sage graphics produced.
- `polyfill` - Whether or not to fill the cones with a color determined by the highest degree in each reduced Groebner basis for that cone.
- `scale_colors` - if True, this will normalize color values to try to maximize the range

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x,z]).groebner_fan()
sage: test_render = G.render()

sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([x^2*y - z, y^2*z - x, z^2*x - y]).groebner_fan()
sage: test_render = G.render(larger=True)
```

TESTS:

Testing the case where the number of generators is < 3. Currently, this should raise a `NotImplementedError` error.

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan().render()
...
NotImplementedError
```

render3d(*verbose=False*)

For a Groebner fan of an ideal in a ring with four variables, this function intersects the fan with the standard simplex perpendicular to (1,1,1,1), creating a 3d polytope, which is then projected into 3 dimensions. The edges of this projected polytope are returned as lines.

EXAMPLES:

```
sage: R4.<w,x,y,z> = PolynomialRing(QQ,4)
sage: gf = R4.ideal([w^2-x,x^2-y,y^2-z,z^2-x]).groebner_fan()
sage: three_d = gf.render3d()
```

TESTS:

Now test the case where the number of generators is not 4. Currently, this should raise a `NotImplementedError` error.

```
sage: P.<a,b,c> = PolynomialRing(QQ, 3, order="lex")
sage: sage.rings.ideal.Katsura(P, 3).groebner_fan().render3d()
...
NotImplementedError
```

ring()

Return the multivariate polynomial ring.

EXAMPLES:

```
sage: R.<x1,x2> = PolynomialRing(QQ,2)
sage: gf = R.ideal([x1^3-x2,x2^3-x1-2]).groebner_fan()
sage: gf.ring()
Multivariate Polynomial Ring in x1, x2 over Rational Field
```

tropical_basis(*check=True, verbose=False*)

Return a tropical basis for the tropical curve associated to this ideal.

INPUT:

- `check` - bool (default: True); if True raises a ValueError exception if this ideal does not define a tropical curve (i.e., the condition that R/I has dimension equal to 1 + the dimension of the homogeneity space is not satisfied).

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3, order='lex')
sage: G = R.ideal([y^3-3*x^2, z^3-x-y-2*y^3+2*x^2]).groebner_fan()
sage: G
Groebner fan of the ideal:
Ideal (-3*x^2 + y^3, 2*x^2 - x - 2*y^3 - y + z^3) of Multivariate Polynomial Ring in x, y, z
sage: G.tropical_basis()
[-3*x^2 + y^3, 2*x^2 - x - 2*y^3 - y + z^3, 3/4*x + y^3 + 3/4*y - 3/4*z^3]
```

tropical_intersection(*ideal_arg=False, *args, **kwds*)

Returns information about the tropical intersection of the polynomials defining the ideal.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: i1 = ideal(x*z + 6*y*z - z^2, x*y + 6*x*z + y*z - z^2, y^2 + x*z + y*z)
sage: gf = i1.groebner_fan()
sage: pf = gf.tropical_intersection()
sage: pf.rays()
[[-1, 0, 0]]
```

weight_vectors()

Returns the weight vectors corresponding to the reduced Groebner bases.

EXAMPLES:

```
sage: r3.<x,y,z> = PolynomialRing(QQ,3)
sage: g = r3.ideal([x^3+y,y^3-z,z^2-x]).groebner_fan()
sage: g.weight_vectors()
[(3, 7, 1), (5, 1, 2), (7, 1, 4), (1, 1, 4), (1, 1, 1), (1, 4, 1), (1, 4, 10)]
sage: r4.<x,y,z,w> = PolynomialRing(QQ,4)
sage: g4 = r4.ideal([x^3+y,y^3-z,z^2-x,z^3 - w]).groebner_fan()
sage: len(g4.weight_vectors())
23
```

class **PolyhedralCone**(*gfan_polyhedral_cone, ring=Rational Field*)

ambient_dim()

Returns the ambient dimension of the Groebner cone.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.ambient_dim()
3
```

dim()
Returns the dimension of the Groebner cone.
EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.dim()
3
```

facets()
Returns the inward facet normals of the Groebner cone.
EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.facets()
[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

lineality_dim()
Returns the lineality dimension of the Groebner cone. This is just the difference between the ambient dimension and the dimension of the cone.
EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.lineality_dim()
0
```

relative_interior_point()
Returns a point in the relative interior of the Groebner cone.
EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.relative_interior_point()
[1, 1, 1]
```

class **PolyhedralFan**(*gfan_polyhedral_fan*)

ambient_dim()
Returns the ambient dimension of the Groebner fan.
EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf.polyhedralfan()
sage: a.ambient_dim()
3
```

dim()
Returns the dimension of the Groebner fan.
EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
```

```
sage: a = gf.polyhedralfan()
sage: a.dim()
3
```

lineality_dim()

Returns the lineality dimension of the Groebner fan. This is just the difference between the ambient dimension and the dimension of the cone.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf.polyhedralfan()
sage: a.lineality_dim()
0
```

rays()

Returns a list of rays of the polyhedral fan.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: i2 = ideal(x*z + 6*y*z - z^2, x*y + 6*x*z + y*z - z^2, y^2 + x*z + y*z)
sage: gf2 = i2.groebner_fan(verbose = False)
sage: pf = gf2.polyhedralfan()
sage: pf.rays()
[[1, 0, 0], [-2, -1, 0], [1, 1, 0], [0, -1, 0], [-1, 1, 0]]
```

class **ReducedGroebnerBasis**(*groebner_fan, gens, gfan_gens*)

groebner_cone(*restrict=False*)

Return defining inequalities for the full-dimensional Groebner cone associated to this marked minimal reduced Groebner basis.

INPUT:

- `restrict` - bool (default: False); if True, add an inequality for each coordinate, so that the cone is restricted to the positive orthant.

OUTPUT: tuple of integer vectors

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: poly_cone = G[1].groebner_cone()
sage: poly_cone.facets()
[[-1, 2], [1, -1]]
sage: [g.groebner_cone().facets() for g in G]
[[[0, 1], [1, -2]], [[-1, 2], [1, -1]], [[-1, 1], [1, 0]]]
sage: G[1].groebner_cone(restrict=True).facets()
[[-1, 2], [1, -1]]
```

ideal()

Return the ideal generated by this basis.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([x - z^3, y^2 - 13*x]).groebner_fan()
sage: G[0].ideal()
Ideal (-13*z^3 + y^2, -z^3 + x) of Multivariate Polynomial Ring in x, y, z over Rational Fie
```

interactive(*latex=False, flippable=False, wall=False, inequalities=False, weight=False*)

Do an interactive walk of the Groebner fan starting at this reduced Groebner basis.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G[0].interactive()      # not tested
Initializing gfan interactive mode
*******************************************
*     Press control-C to return to Sage     *
*******************************************
....
```

max_degree(*list_of_polys*)

Computes the maximum degree of a list of polynomials

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import max_degree
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: p_list = [x^2-y,x*y^10-x]
sage: max_degree(p_list)
11.0
```

prefix_check(*str_list*)

Checks if any strings in a list are prefixes of another string in the list.

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import prefix_check
sage: prefix_check(['z1','z1z1'])
False
sage: prefix_check(['z1','zz1'])
True
```

34.3 Polytopes

This module provides access to polymake, which 'has been developed since 1997 in the Discrete Geometry group at the Institute of Mathematics of Technische Universitat Berlin. Since 2004 the development is shared with Fachbereich Mathematik, Technische Universitat Darmstadt. The system offers access to a wide variety of algorithms and packages within a common framework. polymake is flexible and continuously expanding. The software supplies C++ and perl interfaces which make it highly adaptable to individual needs.'

Note: If you have trouble with this module do:

```
sage:  !polymake --reconfigure   # not tested
```

at the command line.

AUTHORS:

- Ewgenij Gawrilow, Michael Joswig: main authors of polymake
- William Stein: Sage interface

class Polymake()

associahedron(*dimension*)

birkhoff(*n*)

cell24()
EXAMPLES:

```
sage: polymake.cell24()            # optional: needs polymake
The 24-cell
```

convex_hull(*points=, []*)
EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: f = x^3 + y^3 + z^3 + x*y*z
sage: e = f.exponents()
sage: a = [[1] + list(v) for v in e]
sage: a
[[1, 3, 0, 0], [1, 0, 3, 0], [1, 1, 1, 1], [1, 0, 0, 3]]
sage: n = polymake.convex_hull(a)       # optional: needs polymake
sage: n                                 # optional
Convex hull of points [[1, 0, 0, 3], [1, 0, 3, 0], [1, 1, 1, 1], [1, 3, 0, 0]]
sage: n.facets()                        # optional
[(0, 1, 0, 0), (3, -1, -1, 0), (0, 0, 1, 0)]
sage: n.is_simple()                     # optional
True
sage: n.graph()                         # optional
'GRAPH\n{1 2}\n{0 2}\n{0 1}\n\n'
```

cube(*dimension, scale=0*)

from_data(*data*)

rand01(*d, n, seed=None*)

reconfigure()
Reconfigure polymake.

Remember to run polymake.reconfigure() as soon as you have changed the customization file and/or installed missing software!

class **Polytope**(*datafile, desc*)
Create a polytope.

EXAMPLES:

```
sage: P = polymake.convex_hull([[1,0,0,0], [1,0,0,1], [1,0,1,0], [1,0,1,1],  [1,1,0,0], [1,1,0,1
```

Note: If you have trouble with this module do:

```
sage:  !polymake --reconfigure   # not tested
```

at the command line.

cmd(*cmd*)

data()

facets()
EXAMPLES:

```
sage: P = polymake.convex_hull([[1,0,0,0], [1,0,0,1], [1,0,1,0], [1,0,1,1],  [1,1,0,0], [1,1
sage: P.facets()                          # optional
[(0, 0, 0, 1), (0, 1, 0, 0), (0, 0, 1, 0), (1, 0, 0, -1), (1, 0, -1, 0), (1, -1, 0, 0)]
```

graph()

is_simple()
Return True if this polytope is simple.

A polytope is *simple* if the degree of each vertex equals the dimension of the polytope.

EXAMPLES:

```
sage: P = polymake.convex_hull([[1,0,0,0], [1,0,0,1], [1,0,1,0], [1,0,1,1],  [1,1,0,0], [1,1
sage: P.is_simple()                              # optional
True
```

AUTHORS:

- Edwin O'Shea (2006-05-02): Definition of simple.

n_facets()
EXAMPLES:

```
sage: P = polymake.convex_hull([[1,0,0,0], [1,0,0,1], [1,0,1,0], [1,0,1,1],  [1,1,0,0], [1,1
sage: P.n_facets()                               # optional
6
```

vertices()
EXAMPLES:

```
sage: P = polymake.convex_hull([[1,0,0,0], [1,0,0,1], [1,0,1,0], [1,0,1,1],  [1,1,0,0], [1,1
sage: P.vertices()                               # optional
[(1, 0, 0, 0), (1, 0, 0, 1), (1, 0, 1, 0), (1, 0, 1, 1), (1, 1, 0, 0), (1, 1, 0, 1), (1, 1,
```

visual()

write(*filename*)

CHAPTER

THIRTYFIVE

HOMOLOGY OF SIMPLICIAL COMPLEXES

Sage includes support for constructing simplicial complexes and computing their homology.

35.1 Finite simplicial complexes

AUTHORS:

- John H. Palmieri (2009-04)

This module implements the basic structure of finite simplicial complexes. Given a set V of "vertices", a simplicial complex on V is a collection K of subsets of V satisfying the condition that if S is one of the subsets in K, then so is every subset of S. The subsets S are called the 'simplices' of K.

A simplicial complex K can be viewed as a purely combinatorial object, as described above, but it also gives rise to a topological space $|K|$ (its *geometric realization*) as follows: first, the points of V should be in general position in euclidean space. Next, if $\{v\}$ is in K, then the vertex v is in $|K|$. If $\{v, w\}$ is in K, then the line segment from v to w is in $|K|$ If $\{u, v, w\}$ is in K, then the triangle with vertices u, v, and w is in $|K|$. In general, $|K|$ is the union of the convex hulls of simplices of K. Frequently, one abuses notation and uses K to denote both the simplicial complex and the associated topological space.

For any simplicial complex K and any commutative ring R there is an associated chain complex, with differential of degree -1. The n^{th} term is the free R-module with basis given by the n-simplices of K. The differential is determined by its value on any simplex: on the n-simplex with vertices $(v_0, v_1, ..., v_n)$, the differential is the alternating sum with i^{th} summand $(-1)^i$ multiplied by the $(n-1)$-simplex obtained by omitting vertex v_i.

In the implementation here, the vertex set must be finite. To define a simplicial complex, specify its vertex set: this should be a list, tuple, or set, or it can be a non-negative integer n, in which case the vertex set is $(0, ..., n)$. Also specify the facets: the maximal faces.

Note: The elements of the vertex set are not automatically contained in the simplicial complex: each one is only included if and only if it is a vertex of at least one of the specified facets.

EXAMPLES:

```
sage: SimplicialComplex([1, 3, 7], [[1], [3, 7]])
Simplicial complex with vertex set (1, 3, 7) and facets {(3, 7), (1,)}
sage: SimplicialComplex(2)    # the empty simplicial complex
Simplicial complex with vertex set (0, 1, 2) and facets {()}
sage: X = SimplicialComplex(3, [[0,1], [1,2], [2,3], [3,0]])
sage: X
```

```
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 2), (2, 3), (0, 3), (0, 1)}
sage: X.stanley_reisner_ring()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x3 over Integer Ring by the ideal (x1*x3, x0*
sage: X.is_pure()
True
```

Sage can perform a number of operations on simplicial complexes, such as the join and the product, and it can also compute homology:

```
sage: S = SimplicialComplex(3, [[0,1], [1,2], [0,2]]) # circle
sage: T = S.product(S)  # torus
sage: T
Simplicial complex with 16 vertices and 18 facets
sage: T.homology()   # this computes reduced homology
{0: 0, 1: Z x Z, 2: Z}
sage: T.euler_characteristic()
0
```

Sage knows about some basic combinatorial data associated to a simplicial complex:

```
sage: X = SimplicialComplex(3, [[0,1], [1,2], [2,3], [0,3]])
sage: X.f_vector()
[1, 4, 4]
sage: X.face_poset()
Finite poset containing 8 elements
sage: X.stanley_reisner_ring()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x3 over Integer Ring by the ideal (x1*x3, x0*
```

class **Simplex**(*X*)

Define a simplex.

Topologically, a simplex is the convex hull of a collection of vertices in general position. Combinatorially, it is defined just by specifying a set of vertices. It is represented in Sage by the tuple of the vertices.

INPUT:

- X - set of vertices

OUTPUT: simplex with those vertices

X may be a non-negative integer n, in which case the simplicial complex will have $n+1$ vertices $(0, 1, ..., n)$, or it may be anything which may be converted to a tuple, in which case the vertices will be that tuple.

Warning: The vertices should be distinct, and no error checking is done to make sure this is the case.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import Simplex
sage: Simplex(4)
(0, 1, 2, 3, 4)
sage: Simplex([3, 4, 1])
(3, 4, 1)
sage: X = Simplex((3, 'a', 'vertex')); X
(3, 'a', 'vertex')
sage: X == loads(dumps(X))
True
```

dimension()

The dimension of this simplex: the number of vertices minus 1.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import Simplex
sage: Simplex(5).dimension() == 5
True
sage: Simplex(5).face(1).dimension()
4
```

face(*n*)

The nth face of this simplex.

INPUT:

- n - an integer between 0 and the dimension of this simplex

OUTPUT: the simplex obtained by removing the nth vertex from this simplex

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import Simplex
sage: S = Simplex(4)
sage: S.face(0)
(1, 2, 3, 4)
sage: S.face(3)
(0, 1, 2, 4)
```

faces()

The list of faces (of codimension 1) of this simplex.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import Simplex
sage: S = Simplex(4)
sage: S.faces()
[(1, 2, 3, 4), (0, 2, 3, 4), (0, 1, 3, 4), (0, 1, 2, 4), (0, 1, 2, 3)]
sage: len(Simplex(10).faces())
11
```

is_empty()

Return True iff this simplex is the empty simplex.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import Simplex
sage: [Simplex(n).is_empty() for n in range(-1,4)]
[True, False, False, False, False]
```

is_face(*other*)

Return True iff this simplex is a face of other.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import Simplex
sage: Simplex(3).is_face(Simplex(5))
True
sage: Simplex(5).is_face(Simplex(2))
False
sage: Simplex(['a', 'b', 'c']).is_face(Simplex(8))
False
```

join(*right, rename_vertices=True*)

The join of this simplex with another one.

The join of two simplices $[v_0, ..., v_k]$ and $[w_0, ..., w_n]$ is the simplex $[v_0, ..., v_k, w_0, ..., w_n]$.

INPUT:

- `right` - the other simplex (the right-hand factor)
- `rename_vertices` – boolean (optional, default True). If this is True, the vertices in the join will be renamed by this formula: vertex "v" in the left-hand factor –> vertex "Lv" in the join, vertex "w" in the right-hand factor –> vertex "Rw" in the join. If this is false, this tries to construct the join without renaming the vertices; this may cause problems if the two factors have any vertices with names in common.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import Simplex
sage: Simplex(2).join(Simplex(3))
('L0', 'L1', 'L2', 'R0', 'R1', 'R2', 'R3')
sage: Simplex(['a', 'b']).join(Simplex(['x', 'y', 'z']))
('La', 'Lb', 'Rx', 'Ry', 'Rz')
sage: Simplex(['a', 'b']).join(Simplex(['x', 'y', 'z']), rename_vertices=False)
('a', 'b', 'x', 'y', 'z')
```

product (*other, rename_vertices=False*)

The product of this simplex with another one, as a list of simplices.

INPUT:

- `other` - the other simplex
- **`rename_vertices` – boolean (optional, default True). If** this is False, then the vertices in the product are the set of ordered pairs (v, w) where v is a vertex in the left-hand factor (`self`) and w is a vertex in the right-hand factor (`other`). If this is True, then the vertices are renamed as "LvRw" (e.g., the vertex (1,2) would become "L1R2"). This is useful if you want to define the Stanley-Reisner ring of the complex: vertex names like (0,1) are not suitable for that, while vertex names like "L0R1" are.

Algorithm: see Hatcher, p. 277-278 (who in turn refers to Eilenberg-Steenrod, p. 68): given `Simplex(m)` and `Simplex(n)`, then `Simplex(m)` x `Simplex(n)` can be triangulated as follows: for each path f from $(0, 0)$ to (m, n) along the integer grid in the plane, going up or right at each lattice point, associate an $(m + n)$-simplex with vertices v_0, v_1, ..., where v_k is the k^{th} vertex in the path f.

Note that there are $m+n$ choose n such paths. Note also that each vertex in the product is a pair of vertices (v, w) where v is a vertex in the left-hand factor and w is a vertex in the right-hand factor.

Note: This produces a list of simplices – not a `Simplex`, not a `SimplicialComplex`.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import Simplex
sage: len(Simplex(2).product(Simplex(2)))
6
sage: Simplex(1).product(Simplex(1))
[((0, 0), (0, 1), (1, 1)), ((0, 0), (1, 0), (1, 1))]
```

REFERENCES:

- A. Hatcher, "Algebraic Topology", Cambridge University Press (2002).

set ()

The frozenset attached to this simplex.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import Simplex
sage: Simplex(3).set()
frozenset([0, 1, 2, 3])
```

tuple ()

The tuple attached to this simplex.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import Simplex
sage: Simplex(3).tuple()
(0, 1, 2, 3)
```

Although simplices are printed as if they were tuples, they are not the same type:

```
sage: type(Simplex(3).tuple())
<type 'tuple'>
sage: type(Simplex(3))
<class 'sage.homology.simplicial_complex.Simplex'>
```

class SimplicialComplex(*vertex_set=, [], maximal_faces=, [], **kwds*)

Define a simplicial complex.

INPUT:

- `vertex_set` - set of vertices
- `maximal_faces` - set of maximal faces
- `vertex_check` - boolean (optional, default True)
- `maximality_check` - boolean (optional, default True)
- `sort_facets` - boolean (optional, default True)
- `name_check` - boolean (optional, default False)

OUTPUT: a simplicial complex

`vertex_set` may be a non-negative integer n (in which case the simplicial complex will have $n+1$ vertices $\{0, 1, ..., n\}$), or it may be anything which may be converted to a tuple. Call the elements of this 'vertices'.

`maximal_faces` should be a list or tuple or set (indeed, anything which may be converted to a set) whose elements are lists (or tuples, etc.) of vertices.

If `vertex_check` is True, check to see that each given maximal face is a subset of the vertex set. Raise an error for any bad face.

If `maximality_check` is True, check that each maximal face is, in fact, maximal. In this case, when producing the internal representation of the simplicial complex, omit those that are not. It is highly recommended that this be True; various methods for this class may fail if faces which are claimed to be maximal are in fact not.

If `sort_facets` is True, sort the vertices in each facet. If the vertices in different facets are not ordered compatibly (e.g., if you have facets (1, 3, 5) and (5, 3, 8)), then homology calculations may have unpredictable results.

If `name_check` is True, check the names of the vertices to see if they can be easily converted to generators of a polynomial ring – use this if you plan to use the Stanley-Reisner ring for the simplicial complex.

Note: The elements of `vertex_set` are not automatically in the simplicial complex: each one is only included if it is a vertex of at least one of the specified facets.

EXAMPLES:

```
sage: SimplicialComplex(4, [[1,2], [1,4]])
Simplicial complex with vertex set (0, 1, 2, 3, 4) and facets {(1, 2), (1, 4)}
sage: SimplicialComplex(3, [[0,2], [0,3], [0]])
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2), (0, 3)}
sage: SimplicialComplex(3, [[0,2], [0,3], [0]], maximality_check=False)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2), (0, 3), (0,)}
sage: S = SimplicialComplex(['a', 'b', 'c'], (('a', 'b'), ('a', 'c'), ('b', 'c')))
sage: S
Simplicial complex with vertex set ('a', 'b', 'c') and facets {('b', 'c'), ('a', 'c'), ('a', 'b'
sage: S == loads(dumps(S))
True
```

add_face(*face*)

Add a face to this simplicial complex

INPUT:

- face - a subset of the vertex set

This changes the simplicial complex, adding a new face and all of its subfaces.

EXAMPLES:

```
sage: X = SimplicialComplex(2, [[0,1], [0,2]])
sage: X.add_face([0,1,2,]); X
Simplicial complex with vertex set (0, 1, 2) and facets {(0, 1, 2)}
sage: Y = SimplicialComplex(3); Y
Simplicial complex with vertex set (0, 1, 2, 3) and facets {()}
sage: Y.add_face([0,1])
sage: Y.add_face([1,2,3])
sage: Y
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 2, 3), (0, 1)}
```

If you add a face which is already present, there is no effect:

```
sage: Y.add_face([1,3]); Y
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 2, 3), (0, 1)}
```

alexander_dual()

The Alexander dual of this simplicial complex: according to the Macaulay2 documentation, this is the simplicial complex whose faces are the complements of its nonfaces.

Thus find the minimal nonfaces and take their complements to find the facets in the Alexander dual.

EXAMPLES:

```
sage: Y = SimplicialComplex(4); Y
Simplicial complex with vertex set (0, 1, 2, 3, 4) and facets {()}
sage: Y.alexander_dual()
Simplicial complex with vertex set (0, 1, 2, 3, 4) and 5 facets
sage: X = SimplicialComplex(3, [[0,1], [1,2], [2,3], [3,0]])
sage: X.alexander_dual()
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 3), (0, 2)}
```

barycentric_subdivision()

The barycentric subdivision of this simplicial complex.

See http://en.wikipedia.org/wiki/Barycentric_subdivision for a definition.

EXAMPLES:

```
sage: triangle = SimplicialComplex(2, [[0,1], [1,2], [0, 2]])
sage: hexagon = triangle.barycentric_subdivision()
sage: hexagon
Simplicial complex with 6 vertices and 6 facets
sage: hexagon.homology(1) == triangle.homology(1)
True
```

Barycentric subdivisions can get quite large, since each n-dimensional facet in the original complex produces $(n+1)!$ facets in the subdivision:

```
sage: S4 = simplicial_complexes.Sphere(4)
sage: S4
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and 6 facets
sage: S4.barycentric_subdivision()
Simplicial complex with 62 vertices and 720 facets
```

betti(*dim=None, subcomplex=None*)

The Betti numbers of this simplicial complex as a dictionary (or a single Betti number, if only one dimension is given).

INPUT:

- `dim` - integer or list of integers or None (optional, default None). If None, then return every Betti number, as a dictionary with keys the non-negative integers. If `dim` is an integer or list, return the Betti number for each given dimension. (Actually, if `dim` is a list, return the Betti numbers, as a dictionary, in the range from `min(dim)` to `max(dim)`. If `dim` is a number, return the Betti number in that dimension.)
- `subcomplex` - a subcomplex of this simplicial complex (optional, default None). Compute the Betti numbers of the homology relative to this subcomplex.

EXAMPLES: Build the two-sphere as a three-fold join of a two-point space with itself:

```
sage: S = SimplicialComplex(1, [[0], [1]])
sage: (S*S*S).betti()
{0: 0, 1: 0, 2: 1}
sage: (S*S*S).betti([1,2])
{1: 0, 2: 1}
sage: (S*S*S).betti(2)
1
```

category()

Return the category to which this chain complex belongs: the category of all simplicial complexes.

EXAMPLES:

```
sage: SimplicialComplex(5, [[0,1], [1,2,3,4,5]]).category()
Category of simplicial complexes
```

chain_complex(*dimensions=None, base_ring=Integer Ring, subcomplex=None, augmented=False, cochain=False, verbose=False, check_diffs=False*)

The chain complex associated to this simplicial complex.

INPUT:

- `dimensions` - if None, compute the chain complex in all dimensions. If a list or tuple of integers, compute the chain complex in those dimensions, setting the chain groups in all other dimensions to zero.
- `base_ring` - commutative ring (optional, default ZZ)
- `subcomplex` - a subcomplex of this simplicial complex (optional, default empty). Compute the chain complex relative to this subcomplex.
- `augmented` - boolean (optional, default False). If True, return the augmented chain complex (that is, include a class in dimension -1 corresponding to the empty cell). This is ignored if `dimensions` is specified.
- `cochain` - boolean (optional, default False). If True, return the cochain complex (that is, the dual of the chain complex).
- `verbose` - boolean (optional, default False). If True, print some messages as the chain complex is computed.
- `check_diffs` - boolean (optional, default False). If True, make sure that the chain complex is actually a chain complex: the differentials are composable and their product is zero.

Note: If subcomplex is nonempty, then the argument `augmented` has no effect: the chain complex relative to a nonempty subcomplex is zero in dimension -1.

EXAMPLES:

```
sage: circle = SimplicialComplex(2, [[0,1], [1,2], [0, 2]])
sage: circle.chain_complex()
Chain complex with at most 2 nonzero terms over Integer Ring.
sage: circle.chain_complex()._latex_()
'\Bold{Z}^{3} \xrightarrow{d_{1}} \Bold{Z}^{3}'
sage: circle.chain_complex(base_ring=QQ, augmented=True)
Chain complex with at most 3 nonzero terms over Rational Field.
```

cohomology(*dim=None, base_ring=Integer Ring, subcomplex=None, enlarge=True, algorithm='auto', verbose=False*)

The reduced cohomology of this simplicial complex.

INPUT:

- `dim` - integer or list of integers or None (optional, default None). If None, then return the cohomology in every dimension. If `dim` is an integer or list, return the cohomology in the given dimensions. (Actually, if `dim` is a list, return the cohomology in the range from `min(dim)` to `max(dim)`.)
- `base_ring` - commutative ring (optional, default ZZ). Must be ZZ or a field.
- `subcomplex` - a subcomplex of this simplicial complex (optional, default empty). Compute cohomology relative to this subcomplex.
- `enlarge` - boolean (optional, default True). If True, find a new subcomplex homotopy equivalent to, and probably larger than, the given one.
- `algorithm` - string (optional, default 'auto'). This only has an effect if working over the integers. If 'dhsw', then preprocess each boundary matrix using the Dumas, Heckenbach, Saunders, and Welker elimination algorithm. If 'pari', then compute elementary divisors using Pari. If 'linbox', then use LinBox. If 'auto', then use 'dhsw' for large matrices and Pari for small ones.
- `verbose` - boolean (optional, default False). If True, print some messages as the cohomology is computed.

EXAMPLES:

```
sage: circle = SimplicialComplex(2, [[0,1], [1,2], [0, 2]])
sage: circle.cohomology()
{0: 0, 1: Z}
sage: P2 = SimplicialComplex(5, [[0,1,2], [0,2,3], [0,1,5], [0,4,5], [0,3,4], [1,2,4], [1,3,
sage: P2.cohomology()
{0: 0, 1: 0, 2: C2}
sage: P2.cohomology(base_ring=GF(2))
{0: Vector space of dimension 0 over Finite Field of size 2,
1: Vector space of dimension 1 over Finite Field of size 2,
2: Vector space of dimension 1 over Finite Field of size 2}
sage: P2.cohomology(base_ring=GF(3))
{0: Vector space of dimension 0 over Finite Field of size 3,
1: Vector space of dimension 0 over Finite Field of size 3,
2: Vector space of dimension 0 over Finite Field of size 3}
```

Relative cohomology:

```
sage: T = SimplicialComplex(1, [[0,1]])
sage: U = SimplicialComplex(1, [[0], [1]])
sage: T.cohomology(subcomplex=U)
{0: 0, 1: Z}
```

cone()

The cone on this simplicial complex.

The cone is the simplicial complex formed by adding a new vertex C and simplices of the form $[C, v_0, ..., v_k]$ for every simplex $[v_0, ..., v_k]$ in the original simplicial complex. That is, the cone is the join of the original complex with a one-point simplicial complex.

EXAMPLES:

```
sage: S = SimplicialComplex(1, [[0], [1]])
sage: S.cone()
Simplicial complex with vertex set ('L0', 'L1', 'R0') and facets {('L0', 'R0'), ('L1', 'R0')
```

dimension()

The dimension of this simplicial complex: the maximum dimension of its faces.

EXAMPLES:

```
sage: U = SimplicialComplex(5, [[1,2], [1, 3, 4]])
sage: U.dimension()
2
sage: X = SimplicialComplex(3, [[0,1], [0,2], [1,2]])
sage: X.dimension()
1
```

euler_characteristic()

The Euler characteristic of this simplicial complex: the alternating sum over $n \geq 0$ of the number of n-simplices.

EXAMPLES:

```
sage: Y = SimplicialComplex(5, [[1,2], [1,4]])
sage: Y.euler_characteristic()
1
sage: X = SimplicialComplex(3, [[0,1], [0,2], [1,2]])
sage: X.euler_characteristic()
0
```

f_vector()

The f-vector of this simplicial complex: a list whose n^{th} item is the number of $(n-1)$-faces. Note that, like all lists in Sage, this is indexed starting at 0: the 0th element in this list is the number of -1 faces.

EXAMPLES:

```
sage: S = Set(range(1,5))
sage: Z = SimplicialComplex(S, S.subsets()); Z
Simplicial complex with vertex set (1, 2, 3, 4) and facets {(1, 2, 3, 4)}
sage: Z.f_vector()
[1, 4, 6, 4, 1]
sage: Y = SimplicialComplex(5, [[1,2], [1,4]])
sage: Y.f_vector()[2]
2
```

face_poset()

The face poset of this simplicial complex, the poset of nonempty faces, ordered by inclusion.

EXAMPLES:

```
sage: P = SimplicialComplex(3, [[0, 1], [1,2], [2,3]]).face_poset(); P
Finite poset containing 7 elements
sage: P.list()
[(3,), (2,), (2, 3), (1,), (0,), (0, 1), (1, 2)]
```

faces(*subcomplex=None*)

The faces of this simplicial complex, in the form of a dictionary of sets keyed by dimension. If the optional argument `subcomplex` is present, then return only the faces which are *not* in the subcomplex.

INPUT:

- `subcomplex` - a subcomplex of this simplicial complex (optional, default None). Return faces which are not in this subcomplex.

EXAMPLES:

```
sage: Y = SimplicialComplex(5, [[1,2], [1,4]])
sage: Y.faces()
{0: set([(4,), (2,), (1,)]), 1: set([(1, 2), (1, 4)]), -1: set([()])}
sage: L = SimplicialComplex(5, [[1,2]])
sage: Y.faces(subcomplex=L)
{0: set([(4,)]), 1: set([(1, 4)]), -1: set([])}
```

facets()

The maximal faces (a.k.a. facets) of this simplicial complex.

This just returns the set of facets used in defining the simplicial complex, so if the simplicial complex was defined with no maximality checking, none is done here, either.

EXAMPLES:

```
sage: Y = SimplicialComplex(5, [[0,2], [1,4]])
sage: Y.maximal_faces()
{(1, 4), (0, 2)}
```

`facets` is a synonym for `maximal_faces`:

```
sage: S = SimplicialComplex(2, [[0,1], [0,1,2]])
sage: S.facets()
{(0, 1, 2)}
```

graph()

The 1-skeleton of this simplicial complex, as a graph.

EXAMPLES:

```
sage: S = SimplicialComplex(3, [[0,1,2,3]])
sage: G = S.graph(); G
Graph on 4 vertices
sage: G.edges()
[(0, 1, None), (0, 2, None), (0, 3, None), (1, 2, None), (1, 3, None), (2, 3, None)]
```

homology(*dim=None, base_ring=Integer Ring, subcomplex=None, cohomology=False, enlarge=True, algorithm='auto', verbose=False*)

The reduced homology of this simplicial complex.

INPUT:

- `dim` - integer or list of integers or None (optional, default None). If None, then return the homology in every dimension. If `dim` is an integer or list, return the homology in the given dimensions. (Actually, if `dim` is a list, return the homology in the range from `min(dim)` to `max(dim)`.)
- `base_ring` - commutative ring (optional, default ZZ). Must be ZZ or a field.
- `subcomplex` - a subcomplex of this simplicial complex (optional, default None). Compute homology relative to this subcomplex.
- `cohomology` - boolean (optional, default False). If True, compute cohomology rather than homology.
- `enlarge` - boolean (optional, default True). If True, find a new subcomplex homotopy equivalent to, and probably larger than, the given one.
- `algorithm` - string (optional, default 'auto'). This only has an effect if working over the integers. If 'dhsw', then preprocess each boundary matrix using the Dumas, Heckenbach, Saunders, and Welker elimination algorithm. If 'pari', then compute elementary divisors using Pari. If 'linbox', then use LinBox. If 'auto', then use 'dhsw' for large matrices and Pari for small ones.
- `verbose` - boolean (optional, default False). If True, print some messages as the homology is computed.

Algorithm: if `subcomplex` is None, replace it with a facet – a contractible subcomplex of the original complex. Then no matter what `subcomplex` is, replace it with a subcomplex L which is homotopy equivalent and as large as possible. Compute the homology of the original complex relative to L: if L is large, then the relative chain complex will be small enough to speed up computations considerably.

EXAMPLES:

```
sage: circle = SimplicialComplex(2, [[0,1], [1,2], [0, 2]])
sage: circle.homology()
{0: 0, 1: Z}
sage: sphere = SimplicialComplex(3, [[0,1,2,3]])
sage: sphere.remove_face([0,1,2,3])
sage: sphere
```

```
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (0, 1, 2), (1, 2, 3),
sage: sphere.homology()
{0: 0, 1: 0, 2: Z}
```

Another way to get a two-sphere: take a two-point space and take its three-fold join with itself:

```
sage: S = SimplicialComplex(1, [[0], [1]])
sage: (S*S*S).homology(dim=2, cohomology=True)
Z
```

Relative homology:

```
sage: T = SimplicialComplex(2, [[0,1,2]])
sage: U = SimplicialComplex(2, [[0,1], [1,2], [0,2]])
sage: T.homology(subcomplex=U)
{0: 0, 1: 0, 2: Z}
```

is_pure()

True iff this simplicial complex is pure: a simplicial complex is pure iff all of its maximal faces have the same dimension.

Warning: This may give the wrong answer if the simplicial complex was constructed with `maximality_check` set to False.

EXAMPLES:

```
sage: U = SimplicialComplex(5, [[1,2], [1, 3, 4]])
sage: U.is_pure()
False
sage: X = SimplicialComplex(3, [[0,1], [0,2], [1,2]])
sage: X.is_pure()
True
```

join(*right, rename_vertices=True*)

The join of this simplicial complex with another one.

The join of two simplicial complexes S and T is the simplicial complex $S * T$ with simplices of the form $[v_0, ..., v_k, w_0, ..., w_n]$ for all simplices $[v_0, ..., v_k]$ in S and $[w_0, ..., w_n]$ in T.

INPUT:

- `right` - the other simplicial complex (the right-hand factor)
- `rename_vertices` – boolean (optional, default True). If this is True, the vertices in the join will be renamed by the formula: vertex "v" in the left-hand factor –> vertex "Lv" in the join, vertex "w" in the right-hand factor –> vertex "Rw" in the join. If this is false, this tries to construct the join without renaming the vertices; this will cause problems if the two factors have any vertices with names in common.

EXAMPLES:

```
sage: S = SimplicialComplex(1, [[0], [1]])
sage: T = SimplicialComplex([2, 3], [[2], [3]])
sage: S.join(T)
Simplicial complex with vertex set ('L0', 'L1', 'R2', 'R3') and 4 facets
sage: S.join(T, rename_vertices=False)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(1, 3), (1, 2), (0, 2), (0, 3)}
```

The notation '*' may be used, as well:

```
sage: S * S
Simplicial complex with vertex set ('L0', 'L1', 'R0', 'R1') and 4 facets
sage: S * S * S * S * S * S * S * S * S
Simplicial complex with 16 vertices and 256 facets
```

link(*simplex*)

The link of a simplex in this simplicial complex.

The link of a simplex F is the simplicial complex formed by all simplices G which are disjoint from F but for which $F \cup G$ is a simplex.

INPUT:

- `simplex` - a simplex in this simplicial complex.

EXAMPLES:

```
sage: X = SimplicialComplex(4, [[0,1,2], [1,2,3]])
sage: from sage.homology.simplicial_complex import Simplex
sage: X.link(Simplex([0]))
Simplicial complex with vertex set (0, 1, 2, 3, 4) and facets {(1, 2)}
sage: X.link([1,2])
Simplicial complex with vertex set (0, 1, 2, 3, 4) and facets {(3,), (0,)}
sage: Y = SimplicialComplex(3, [[0,1,2,3]])
sage: Y.link([1])
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3)}
```

maximal_faces()

The maximal faces (a.k.a. facets) of this simplicial complex.

This just returns the set of facets used in defining the simplicial complex, so if the simplicial complex was defined with no maximality checking, none is done here, either.

EXAMPLES:

```
sage: Y = SimplicialComplex(5, [[0,2], [1,4]])
sage: Y.maximal_faces()
{(1, 4), (0, 2)}
```

`facets` is a synonym for `maximal_faces`:

```
sage: S = SimplicialComplex(2, [[0,1], [0,1,2]])
sage: S.facets()
{(0, 1, 2)}
```

minimal_nonfaces()

Set consisting of the minimal subsets of the vertex set of this simplicial complex which do not form faces.

Algorithm: first take the complement (within the vertex set) of each facet, obtaining a set $(f_1, f_2, ...)$ of simplices. Now form the set of all simplices of the form $(v_1, v_2, ...)$ where vertex v_i is in face f_i. This set will contain the minimal nonfaces and may contain some non-minimal nonfaces also, so loop through the set to find the minimal ones. (The last two steps are taken care of by the `_transpose_simplices` routine.)

This is used in computing the Stanley-Reisner ring and the Alexander dual.

EXAMPLES:

```
sage: X = SimplicialComplex(4)
sage: X.minimal_nonfaces()
{(4,), (2,), (3,), (0,), (1,)}
sage: X.add_face([1,2])
sage: X.add_face([1,3])
sage: X.minimal_nonfaces()
{(4,), (2, 3), (0,)}
sage: Y = SimplicialComplex(3, [[0,1], [1,2], [2,3], [3,0]])
sage: Y.minimal_nonfaces()
{(1, 3), (0, 2)}
```

n_faces(*n, subcomplex=None*)

The set of faces of dimension n of this simplicial complex. If the optional argument `subcomplex` is present, then return the `n`-dimensional faces which are *not* in the subcomplex.

INPUT:

- n - non-negative integer
- subcomplex - a subcomplex of this simplicial complex (optional, default None). Return n-dimensional faces which are not in this subcomplex.

EXAMPLES:

```
sage: S = Set(range(1,5))
sage: Z = SimplicialComplex(S, S.subsets())
sage: Z
Simplicial complex with vertex set (1, 2, 3, 4) and facets {(1, 2, 3, 4)}
sage: Z.n_faces(2)
set([(1, 3, 4), (1, 2, 3), (2, 3, 4), (1, 2, 4)])
sage: K = SimplicialComplex(S, [[1,2,3], [2,3,4]])
sage: Z.n_faces(2, subcomplex=K)
set([(1, 3, 4), (1, 2, 4)])
```

n_skeleton(*n*)

The n-skeleton of this simplicial complex: the simplicial complex obtained by discarding all of the simplices in dimensions larger than n.

INPUT:

- n - non-negative integer

EXAMPLES:

```
sage: X = SimplicialComplex(3, [[0,1], [1,2,3], [0,2,3]])
sage: X.n_skeleton(1)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(2, 3), (0, 2), (1, 3), (1, 2),
sage: X.n_skeleton(2)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (1, 2, 3), (0, 1)}
```

product(*right, rename_vertices=True*)

The product of this simplicial complex with another one.

INPUT:

- right - the other simplicial complex (the right-hand factor)
- **rename_vertices – boolean (optional, default True). If** this is False, then the vertices in the product are the set of ordered pairs (v, w) where v is a vertex in self and w is a vertex in right. If this is True, then the vertices are renamed as "LvRw" (e.g., the vertex (1,2) would become "L1R2"). This is useful if you want to define the Stanley-Reisner ring of the complex: vertex names like (0,1) are not suitable for that, while vertex names like "L0R1" are.

The vertices in the product will be the set of ordered pairs (v, w) where v is a vertex in self and w is a vertex in right.

Warning: If X and Y are simplicial complexes, then X*Y returns their join, not their product.

EXAMPLES:

```
sage: S = SimplicialComplex(3, [[0,1], [1,2], [0,2]]) # circle
sage: K = SimplicialComplex(1, [[0,1]])   # edge
sage: S.product(K).vertices()  # cylinder
('L0R0', 'L0R1', 'L1R0', 'L1R1', 'L2R0', 'L2R1', 'L3R0', 'L3R1')
sage: S.product(K, rename_vertices=False).vertices()
((0, 0), (0, 1), (1, 0), (1, 1), (2, 0), (2, 1), (3, 0), (3, 1))
sage: T = S.product(S)  # torus
sage: T
Simplicial complex with 16 vertices and 18 facets
sage: T.homology()
{0: 0, 1: Z x Z, 2: Z}
```

These can get large pretty quickly:

```
sage: T = simplicial_complexes.Torus(); T
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5, 6) and 14 facets
sage: K = simplicial_complexes.KleinBottle(); K
Simplicial complex with 9 vertices and 18 facets
sage: T.product(K)      # long time: 5 or 6 seconds
Simplicial complex with 63 vertices and 1512 facets
```

remove_face (*face*)

Remove a face from this simplicial complex

INPUT:

- `face` - a face of the simplicial complex

This changes the simplicial complex, removing the given face any face which contains it.

Algorithm: take the Alexander dual, add the complement of `face`, and then take the Alexander dual again.

EXAMPLES:

```
sage: S = range(1,5)
sage: Z = SimplicialComplex(S, [S]); Z
Simplicial complex with vertex set (1, 2, 3, 4) and facets {(1, 2, 3, 4)}
sage: Z.remove_face([1,2])
sage: Z
Simplicial complex with vertex set (1, 2, 3, 4) and facets {(1, 3, 4), (2, 3, 4)}
```

stanley_reisner_ring (*base_ring=Integer Ring*)

The Stanley-Reisner ring of this simplicial complex.

INPUT:

- `base_ring` - a commutative ring (optional, default ZZ)

OUTPUT:

- a quotient of a polynomial algebra with coefficients in `base_ring`, with one generator for each vertex in the simplicial complex, by the ideal generated by the products of those vertices which do not form faces in it.

Thus the ideal is generated by the products corresponding to the minimal nonfaces of the simplicial complex.

> **Warning:** This may be quite slow!
> Also, this may behave badly if the vertices have the 'wrong' names. To avoid this, define the simplicial complex at the start with the flag `name_check` set to True.
> More precisely, this is a quotient of a polynomial ring with one generator for each vertex. If the name of a vertex is a non-negative integer, then the corresponding polynomial generator is named 'x' followed by that integer (e.g., 'x2', 'x3', 'x5', ...). Otherwise, the polynomial generators are given the same names as the vertices. Thus if the vertex set is (2, 'x2'), there will be problems.

EXAMPLES:

```
sage: X = SimplicialComplex(3, [[0,1], [1,2], [2,3], [0,3]])
sage: X.stanley_reisner_ring()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x3 over Integer Ring by the ideal (x
sage: Y = SimplicialComplex(4); Y
Simplicial complex with vertex set (0, 1, 2, 3, 4) and facets {()}
sage: Y.stanley_reisner_ring()
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x3, x4 over Integer Ring by the idea
sage: Y.add_face([0,1,2,3,4])
sage: Y.stanley_reisner_ring(base_ring=QQ)
Quotient of Multivariate Polynomial Ring in x0, x1, x2, x3, x4 over Rational Field by the id
```

suspension(*n=1*)

The suspension of this simplicial complex.

INPUT:

- n - positive integer (optional, default 1): suspend this many times.

The suspension is the simplicial complex formed by adding two new vertices S_0 and S_1 and simplices of the form $[S_0, v_0, ..., v_k]$ and $[S_1, v_0, ..., v_k]$ for every simplex $[v_0, ..., v_k]$ in the original simplicial complex. That is, the cone is the join of the original complex with a two-point simplicial complex.

EXAMPLES:

```
sage: S = SimplicialComplex(1, [[0], [1]])
sage: S.suspension()
Simplicial complex with vertex set ('L0', 'L1', 'R0', 'R1') and 4 facets
sage: S3 = S.suspension(3)  # the 3-sphere
sage: S3.homology()
{0: 0, 1: 0, 2: 0, 3: Z}
```

vertices()

The vertex set of this simplicial complex.

EXAMPLES:

```
sage: S = SimplicialComplex(15, [[0,1], [1,2]])
sage: S
Simplicial complex with 16 vertices and facets {(1, 2), (0, 1)}
sage: S.vertices()
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15)
```

Note that this actually returns a simplex:

```
sage: type(S.vertices())
<class 'sage.homology.simplicial_complex.Simplex'>
```

lattice_paths(*t1, t2*)

Given lists (or tuples or ...) t1 and t2, think of them as labelings for vertices: t1 labeling points on the x-axis, t2 labeling points on the y-axis, both increasing. Return the list of rectilinear paths along the grid defined by these points in the plane, starting from (t1[0], t2[0]), ending at (t1[last], t2[last]), and at each grid point, going either right or up. See the examples.

INPUT:

- t1, t2 - tuples, lists, other iterables.

OUTPUT: list of lists of vertices making up the paths as described above

This is used when triangulating the product of simplices.

EXAMPLES:

```
sage: from sage.homology.simplicial_complex import lattice_paths
sage: lattice_paths([0,1,2], [0,1,2])
[[(0, 0), (0, 1), (0, 2), (1, 2), (2, 2)],
 [(0, 0), (0, 1), (1, 1), (1, 2), (2, 2)],
 [(0, 0), (1, 0), (1, 1), (1, 2), (2, 2)],
 [(0, 0), (0, 1), (1, 1), (2, 1), (2, 2)],
 [(0, 0), (1, 0), (1, 1), (2, 1), (2, 2)],
 [(0, 0), (1, 0), (2, 0), (2, 1), (2, 2)]]
sage: lattice_paths(('a', 'b', 'c'), (0, 3, 5))
[[('a', 0), ('a', 3), ('a', 5), ('b', 5), ('c', 5)],
 [('a', 0), ('a', 3), ('b', 3), ('b', 5), ('c', 5)],
 [('a', 0), ('b', 0), ('b', 3), ('b', 5), ('c', 5)],
 [('a', 0), ('a', 3), ('b', 3), ('c', 3), ('c', 5)],
```

```
[('a', 0), ('b', 0), ('b', 3), ('c', 3), ('c', 5)],
[('a', 0), ('b', 0), ('c', 0), ('c', 3), ('c', 5)]]
```

35.2 Chain complexes

AUTHORS:

- John H. Palmieri (2009-04)

This module implements chain complexes of free R-modules, for any commutative ring R (although the interesting things, like homology, only work if R is the integers or a field).

Fix a ring R. A chain complex over R is a collection of R-modules $\{C_n\}$ indexed by the integers, with R-module maps $d_n : C_n \to C_{n+1}$ such that $d_{n+1} \circ d_n = 0$ for all n. The maps d_n are called *differentials*.

One can vary this somewhat: the differentials may decrease degree by one instead of increasing it: sometimes a chain complex is defined with $d_n : C_n \to C_{n-1}$ for each n. Indeed, the differentials may change dimension by any fixed integer.

Also, the modules may be indexed over an abelian group other than the integers, e.g., $\mathbf{Z}^m$ for some integer $m \geq 1$, in which case the differentials may change the grading by any element of that grading group.

In this implementation, the ring R must be commutative and the modules C_n must be free R-modules. As noted above, homology calculations will only work if the ring R is either $\mathbf{Z}$ or a field. The modules may be indexed by any free abelian group. The differentials may increase degree by 1 or decrease it, or indeed change it by any fixed amount: this is controlled by the `degree` parameter used in defining the chain complex.

class `ChainComplex` (*data=None, base_ring=None, grading_group=Integer Ring, degree=1, check_products=True*)

Define a chain complex.

INPUT:

- `data` - the data defining the chain complex; see below for more details.
- `base_ring` - a commutative ring (optional), the ring over which the chain complex is defined. If this is not specified, it is determined by the data defining the chain complex.
- `grading_group` - a free abelian group (optional, default ZZ), the group over which the chain complex is indexed.
- `degree` - element of grading_group (optional, default 1), the degree of the differential.
- `check_products` - boolean (optional, default True). If True, check that each consecutive pair of differentials are composable and have composite equal to zero.

OUTPUT: a chain complex

Warning: Right now, homology calculations will only work if the base ring is either ZZ or a field, so please take this into account when defining a chain complex.

Use data to define the chain complex. This may be in any of the following forms.

1. a dictionary with integers (or more generally, elements of grading_group) for keys, and with data[n] a matrix representing (via left multiplication) the differential coming from degree n. (Note that the shape of the matrix then determines the rank of the free modules C_n and C_{n+d}.)
2. a list or tuple of the form $[C_0, d_0, C_1, d_1, C_2, d_2, ...]$, where each C_i is a free module and each d_i is a matrix, as above. This only makes sense if `grading_group` is $\mathbf{Z}$ and `degree` is 1.

3. a list or tuple of the form $[r_0, d_0, r_1, d_1, r_2, d_2, ...]$, where r_i is the rank of the free module C_i and each d_i is a matrix, as above. This only makes sense if `grading_group` is **Z** and `degree` is 1.
4. a list or tuple of the form $[d_0, d_1, d_2, ...]$ where each d_i is a matrix, as above. This only makes sense if `grading_group` is **Z** and `degree` is 1.

Note: In fact, the free modules C_i in case 2 and the ranks r_i in case 3 are ignored: only the matrices are kept, and from their shapes, the ranks of the modules are determined. (Indeed, if `data` is a list or tuple, then any element which is not a matrix is discarded; thus the list may have any number of different things in it, and all of the non-matrices will be ignored.) No error checking is done to make sure, for instance, that the given modules have the appropriate ranks for the given matrices. However, as long as `check_products` is True, the code checks to see if the matrices are composable and that each appropriate composite is zero.

If the base ring is not specified, then the matrices are examined to determine a ring over which they are all naturally defined, and this becomes the base ring for the complex. If no such ring can be found, an error is raised. If the base ring is specified, then the matrices are converted automatically to this ring when defining the chain complex. If some matrix cannot be converted, then an error is raised.

EXAMPLES:

```
sage: ChainComplex()
Chain complex with at most 0 nonzero terms over Integer Ring.
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C
Chain complex with at most 2 nonzero terms over Integer Ring.
sage: D = ChainComplex([matrix(ZZ, 2, 2, [0, 1, 0, 0]), matrix(ZZ, 2, 2, [0, 1, 0, 0])], base_ri
Chain complex with at most 3 nonzero terms over Finite Field of size 2.
sage: D == loads(dumps(D))
True
```

Note that when a chain complex is defined in Sage, new differentials may be created: every nonzero module in the chain complex must have a differential coming from it, even if that differential is zero:

```
sage: IZ = ChainComplex({0: identity_matrix(ZZ, 1)})
sage: IZ.differential()  # the differentials in the chain complex
{0: [1], 1: []}
sage: IZ.differential(1).parent()
Full MatrixSpace of 0 by 1 dense matrices over Integer Ring
```

Defining the base ring implicitly:

```
sage: ChainComplex([matrix(QQ, 3, 1), matrix(ZZ, 4, 3)])
Chain complex with at most 2 nonzero terms over Rational Field.
sage: ChainComplex([matrix(GF(125, 'a'), 3, 1), matrix(ZZ, 4, 3)])
Chain complex with at most 2 nonzero terms over Finite Field in a of size 5^3.
```

If the matrices are defined over incompatible rings, an error results:

```
sage: ChainComplex([matrix(GF(125, 'a'), 3, 1), matrix(QQ, 4, 3)])
...
TypeError: unable to find a common ring for all elements
```

If the base ring is given explicitly but is not compatible with the matrices, an error results:

```
sage: ChainComplex([matrix(GF(125, 'a'), 3, 1)], base_ring=QQ)
...
TypeError: Unable to coerce 0 (<type 'sage.rings.finite_field_givaro.FiniteField_givaroElement'>
```

base_ring()

The base ring for this simplicial complex.

EXAMPLES:

```
sage: ChainComplex().base_ring()
Integer Ring
```

betti(*dim=None, base_ring=None*)

The Betti number of the homology of the chain complex in this dimension.

That is, write the homology in this dimension as a direct sum of a free module and a torsion module; the Betti number is the rank of the free summand.

INPUT:

- dim - an element of the grading group for the chain complex or None (optional, default None). If None, then return every Betti number, as a dictionary indexed by degree. If an element of the grading group, then return the Betti number in that dimension.
- base_ring - a commutative ring (optional, default is the base ring for the chain complex). Compute homology with these coefficients. Must be either the integers or a field.

OUTPUT: the Betti number in dimension dim - the rank of the free part of the homology module in this dimension.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C.betti(0)
2
sage: [C.betti(n) for n in range(5)]
[2, 1, 0, 0, 0]
sage: C.betti()
{0: 2, 1: 1}
```

category()

Return the category to which this chain complex belongs: the category of all chain complexes over the base ring.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])}, base_ring=GF(7))
sage: C.category()
Category of chain complexes over Finite Field of size 7
```

differential(*dim=None*)

The differentials which make up the chain complex.

INPUT:

- dim - element of the grading group (optional, default None). If this is None, return a dictionary of all of the differentials. If this is a single element, return the differential starting in that dimension.

OUTPUT: either a dictionary of all of the differentials or a single differential (i.e., a matrix)

EXAMPLES:

```
sage: D = ChainComplex({0: matrix(ZZ, 2, 2, [1,0,0,2])})
sage: D.differential()
{0: [1 0]
[0 2], 1: []}
sage: D.differential(0)
[1 0]
[0 2]
sage: C = ChainComplex({0: identity_matrix(ZZ, 40)})
sage: C.differential()
{0: 40 x 40 dense matrix over Integer Ring, 1: []}
```

free_module()

The free module underlying this chain complex.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0]), 1: matrix(ZZ, 0, 2)})
sage: C.free_module()
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

This defines the forgetful functor from the category of chain complexes to the category of free modules:

```
sage: FreeModules(ZZ)(C)
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

homology(*dim=None, base_ring=None, verbose=False, algorithm='auto'*)

The homology of the chain complex in the given dimension.

INPUT:

- `dim` - an element of the grading group for the chain complex (optional, default None): the degree in which to compute homology. If this is None, return the homology in every dimension in which the chain complex is possibly nonzero.
- `base_ring` - a commutative ring (optional, default is the base ring for the chain complex). Must be either the integers **Z** or a field.
- `verbose` - boolean (optional, default False). If True, print some messages as the homology is computed.
- `algorithm` - string (optional, default 'auto'). This only has an effect if working over the integers. If 'dhsw', then preprocess each boundary matrix using the Dumas, Heckenbach, Saunders, and Welker elimination algorithm, and then compute elementary divisors using Pari. If 'pari', then just compute elementary divisors using Pari. If 'linbox', then use LinBox if available (likely it's not). If 'auto', then use 'dhsw' for large matrices and 'pari' for small ones.

OUTPUT: if dim is specified, the homology in dimension dim. Otherwise, the homology in every dimension as a dictionary indexed by dimension.

Warning: This only works if the base ring is the integers or a field. Other values will return an error.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C.homology()
{0: Z x Z, 1: Z x C3}
sage: C.homology(dim=1, base_ring = GF(3))
Vector space of dimension 2 over Finite Field of size 3
sage: D = ChainComplex({0: identity_matrix(ZZ, 4), 4: identity_matrix(ZZ, 30)})
sage: D.homology()
{0: 0, 1: 0, 4: 0, 5: 0}
```

torsion_list(*max_prime, min_prime=2*)

Look for torsion in this chain complex by computing its mod p homology for a range of primes p.

INPUT:

- `max_prime` - prime number: search for torsion mod p for all p strictly less than this number.
- `min_prime` - prime (optional, default 2): search for torsion mod p for primes at least as big as this.

Return a list of pairs (p, `dims`) where p is a prime at which there is torsion and `dims` is a list of dimensions in which this torsion occurs.

The base ring for the chain complex must be the integers; if not, an error is raised.

Algorithm: let C denote the chain complex. Let P equal `max_prime`. Compute the mod P homology of C, and use this as the base-line computation: the assumption is that this is isomorphic to the integral

homology tensored with $\mathbf{F}_P$. Then compute the mod p homology for a range of primes p, and record whenever the answer differs from the base-line answer.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C.homology()
{0: Z x Z, 1: Z x C3}
sage: C.torsion_list(11)
[(3, [1])]
sage: C = ChainComplex([matrix(ZZ, 1, 1, [2]), matrix(ZZ, 1, 1), matrix(1, 1, [3])])
sage: C.homology()
{0: 0, 1: C2, 2: 0, 3: C3}
sage: C.torsion_list(5)
[(2, [1]), (3, [3])]
```

HomologyGroup(*n, invfac=None*)

Abelian group on n generators. This class inherits from `AbelianGroup`; see that for more documentation. The main difference between the classes is in the print representation; also, this class does not accept a `names` argument.

EXAMPLES:

```
sage: from sage.homology.chain_complex import HomologyGroup
sage: G = AbelianGroup(5,[5,5,7,8,9]); G
Multiplicative Abelian Group isomorphic to C5 x C5 x C7 x C8 x C9
sage: H = HomologyGroup(5,[5,5,7,8,9]); H
C5 x C5 x C7 x C8 x C9
sage: AbelianGroup(4)
Multiplicative Abelian Group isomorphic to Z x Z x Z x Z
sage: HomologyGroup(4)
Z x Z x Z x Z
sage: HomologyGroup(100)
Z^100
```

class **HomologyGroup_class**(*n, invfac*)

Abelian group on n generators. This class inherits from `AbelianGroup`; see that for more documentation. The main difference between the classes is in the print representation; also, this class does not accept a `names` argument.

EXAMPLES:

```
sage: from sage.homology.chain_complex import HomologyGroup
sage: G = AbelianGroup(5,[5,5,7,8,9]); G
Multiplicative Abelian Group isomorphic to C5 x C5 x C7 x C8 x C9
sage: H = HomologyGroup(5,[5,5,7,8,9]); H
C5 x C5 x C7 x C8 x C9
sage: G == loads(dumps(G))
True
sage: AbelianGroup(4)
Multiplicative Abelian Group isomorphic to Z x Z x Z x Z
sage: HomologyGroup(4)
Z x Z x Z x Z
sage: HomologyGroup(100)
Z^100
```

dhsw_snf(*mat, verbose=False*)

Preprocess a matrix using the "Elimination algorithm" described by Dumas et al., and then call `elementary_divisors` on the resulting (smaller) matrix.

'dhsw' stands for 'Dumas, Heckenbach, Saunders, Welker,' and 'snf' stands for 'Smith Normal Form.'

INPUT:

- `mat` - an integer matrix, either sparse or dense.

(They use the transpose of the matrix considered here, so they use rows instead of columns.)

Algorithm: go through `mat` one column at a time. For each column, add multiples of previous columns to it until either

- it's zero, in which case it should be deleted.
- its first nonzero entry is 1 or -1, in which case it should be kept.
- its first nonzero entry is something else, in which case it is deferred until the second pass.

Then do a second pass on the deferred columns.

At this point, the columns with 1 or -1 in the first entry contribute to the rank of the matrix, and these can be counted and then deleted (after using the 1 or -1 entry to clear out its row). Suppose that there were N of these.

The resulting matrix should be much smaller; we then feed it to Sage's `elementary_divisors` function, and prepend N 1's to account for the rows deleted in the previous step.

EXAMPLES:

```
sage: from sage.homology.chain_complex import dhsw_snf
sage: mat = matrix(ZZ, 3, 4, range(12))
sage: dhsw_snf(mat)
[1, 4, 0]
sage: mat = random_matrix(ZZ, 20, 20, x=-1, y=2)
sage: mat.elementary_divisors() == dhsw_snf(mat)
True
```

REFERENCES:

- Dumas, Heckenbach, Saunders, Welker, "Computing simplicial homology based on efficient Smith normal form algorithms," in "Algebra, geometry, and software systems" (2003), 177-206.

35.3 Examples of simplicial complexes

AUTHORS:

- John H. Palmieri (2009-04)

This file constructs some examples of simplicial complexes. There are two main types: surfaces and examples related to graph theory.

For surfaces, this file introduces a class, SimplicialSurface, which is based on SimplicialComplex but also includes a `connected_sum` method. It also has routines for defining the 2-sphere, the n-sphere for any n, the torus, the real projective plane, the Klein bottle, and surfaces of arbitrary genus, all as simplicial complexes.

Aside from surfaces, this file also provides some functions for constructing some other simplicial complexes: the simplicial complex of not-i-connected graphs on n vertices, the matching complex on n vertices, and the chessboard complex for an n by i chessboard. These provide examples of large simplicial complexes; for example, not_i_connected_graphs(7,2) has over a million simplices.

All of these examples are accessible by typing "simplicial_complexes.NAME", where "NAME" is the name of the example; you can get a list by typing "simplicial_complexes." and hitting the TAB key:

```
Sphere
Simplex
Torus
ProjectivePlane
KleinBottle
SurfaceOfGenus
MooreSpace
NotIConnectedGraphs
MatchingComplex
ChessboardComplex
RandomComplex
```

See the documentation for `simplicial_complexes` and for each particular type of example for full details.

class SimplicialComplexExamples()

Some examples of simplicial complexes.

Here are the available examples; you can also type "simplicial_complexes." and hit tab to get a list:

```
Sphere
Simplex
Torus
ProjectivePlane
KleinBottle
SurfaceOfGenus
MooreSpace
NotIConnectedGraphs
MatchingComplex
ChessboardComplex
RandomComplex
```

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(2) # the 2-sphere
sage: S.homology()
{0: 0, 1: 0, 2: Z}
sage: simplicial_complexes.SurfaceOfGenus(3)
Simplicial complex with 15 vertices and 38 facets
sage: M4 = simplicial_complexes.MooreSpace(4)
sage: M4.homology()
{0: 0, 1: C4, 2: 0}
sage: simplicial_complexes.MatchingComplex(6).homology()
{0: 0, 1: Z^16, 2: 0}
```

ChessboardComplex(*n, i*)

The chessboard complex for an n by i chessboard.

Fix integers $n, i > 0$ and consider sets V of n vertices and W of i vertices. A 'partial matching' between V and W is a graph formed by edges (v, w) with $v \in V$ and $w \in W$ so that each vertex is in at most one edge. If G is a partial matching, then so is any graph obtained by deleting edges from G. Thus the set of all partial matchings on V and W, viewed as a set of subsets of the $n + i$ choose 2 possible edges, is closed under taking subsets, and thus forms a simplicial complex called the 'chessboard complex'. This function produces that simplicial complex. (It is called the chessboard complex because such graphs also correspond to ways of placing rooks on an n by i chessboard so that none of them are attacking each other.)

INPUT:

- `n, i` - positive integers.

See Dumas et al. for information on computing its homology by computer, and see Wachs for an expository article about the theory.

EXAMPLES:

```
sage: C = simplicial_complexes.ChessboardComplex(5,5)
sage: C.f_vector()
[1, 25, 200, 600, 600, 120]
sage: simplicial_complexes.ChessboardComplex(3,3).homology()
{0: 0, 1: Z x Z x Z x Z, 2: 0}
```

REFERENCES:

- Dumas, Heckenbach, Saunders, Welker, "Computing simplicial homology based on efficient Smith normal form algorithms," in "Algebra, geometry, and software systems" (2003), 177-206.
- Wachs, "Topology of Matching, Chessboard and General Bounded Degree Graph Complexes" (Algebra Universalis Special Issue in Memory of Gian-Carlo Rota, Algebra Universalis, 49 (2003) 345-385)

KleinBottle()

A triangulation of the Klein bottle, formed by taking the connected sum of a projective plane with itself. (This is not a minimal triangulation.)

EXAMPLES:

```
sage: simplicial_complexes.KleinBottle()
Simplicial complex with 9 vertices and 18 facets
```

MatchingComplex(n)

The matching complex of graphs on n vertices.

Fix an integer $n > 0$ and consider a set V of n vertices. A 'partial matching' on V is a graph formed by edges so that each vertex is in at most one edge. If G is a partial matching, then so is any graph obtained by deleting edges from G. Thus the set of all partial matchings on n vertices, viewed as a set of subsets of the n choose 2 possible edges, is closed under taking subsets, and thus forms a simplicial complex called the 'matching complex'. This function produces that simplicial complex.

INPUT:

- n - positive integer.

See Dumas et al. for information on computing its homology by computer, and see Wachs for an expository article about the theory. For example, the homology of these complexes seems to have only mod 3 torsion, and this has been proved for the bottom non-vanishing homology group for the matching complex M_n.

EXAMPLES:

```
sage: M = simplicial_complexes.MatchingComplex(7)
sage: H = M.homology()
sage: H
{0: 0, 1: C3, 2: Z^20}
sage: H[2].ngens()
20
sage: simplicial_complexes.MatchingComplex(8).homology(2)  # long time (a few seconds)
Z^132
```

REFERENCES:

- Dumas, Heckenbach, Saunders, Welker, "Computing simplicial homology based on efficient Smith normal form algorithms," in "Algebra, geometry, and software systems" (2003), 177-206.
- Wachs, "Topology of Matching, Chessboard and General Bounded Degree Graph Complexes" (Algebra Universalis Special Issue in Memory of Gian-Carlo Rota, Algebra Universalis, 49 (2003) 345-385)

MooreSpace(q)

Triangulation of the mod q Moore space.

INPUT:

- q - integer, at least 2

This is a simplicial complex with simplices of dimension 0, 1, and 2, such that its reduced homology is isomorphic to $\mathbf{Z}/q\mathbf{Z}$ in dimension 1, zero otherwise.

If $q = 2$, this is the projective plane. If $q > 2$, then construct it as follows: start with a triangle with vertices 1, 2, 3. We take a $3q$-gon forming a q-fold cover of the triangle, and we form the resulting complex as an identification space of the $3q$-gon. To triangulate this identification space, put q vertices A_0, ..., A_{q-1}, in the interior, each of which is connected to 1, 2, 3 (two facets each: $[1, 2, A_i]$, $[2, 3, A_i]$). Put q more vertices in the interior: B_0, ..., B_{q-1}, with facets $[3, 1, B_i]$, $[3, B_i, A_i]$, $[1, B_i, A_{i+1}]$, $[B_i, A_i, A_{i+1}]$. Then triangulate the interior polygon with vertices A_0, A_1, ..., A_{q-1}.

EXAMPLES:

```
sage: simplicial_complexes.MooreSpace(3).homology()
{0: 0, 1: C3, 2: 0}
sage: simplicial_complexes.MooreSpace(4).suspension().homology()
{0: 0, 1: 0, 2: C4, 3: 0}
sage: simplicial_complexes.MooreSpace(8)
Simplicial complex with 19 vertices and 54 facets
```

NotIConnectedGraphs(*n, i*)

The simplicial complex of all graphs on n vertices which are not i-connected.

Fix an integer $n > 0$ and consider the set of graphs on n vertices. View each graph as its set of edges, so it is a subset of a set of size n choose 2. A graph is i-connected if, for any $j < i$, if any j vertices are removed along with the edges emanating from them, then the graph remains connected. Now fix i: it is clear that if G is not i-connected, then the same is true for any graph obtained from G by deleting edges. Thus the set of all graphs which are not i-connected, viewed as a set of subsets of the n choose 2 possible edges, is closed under taking subsets, and thus forms a simplicial complex. This function produces that simplicial complex.

INPUT:

- `n`, `i` - non-negative integers with i at most n

See Dumas et al. for information on computing its homology by computer, and see Babson et al. for theory. For example, Babson et al. show that when $i = 2$, the reduced homology of this complex is nonzero only in dimension $2n - 5$, where it is free abelian of rank $(n - 2)!$.

EXAMPLES:

```
sage: simplicial_complexes.NotIConnectedGraphs(5,2).f_vector()
[1, 10, 45, 120, 210, 240, 140, 20]
sage: simplicial_complexes.NotIConnectedGraphs(5,2).homology(5).ngens()
6
```

REFERENCES:

- Babson, Bjorner, Linusson, Shareshian, and Welker, "Complexes of not i-connected graphs," Topology 38 (1999), 271-299
- Dumas, Heckenbach, Saunders, Welker, "Computing simplicial homology based on efficient Smith normal form algorithms," in "Algebra, geometry, and software systems" (2003), 177-206.

ProjectivePlane()

A minimal triangulation of the real projective plane.

EXAMPLES:

```
sage: P = simplicial_complexes.ProjectivePlane()
sage: P.cohomology()
{0: 0, 1: 0, 2: C2}
sage: P.cohomology(base_ring=GF(2))
{0: Vector space of dimension 0 over Finite Field of size 2,
1: Vector space of dimension 1 over Finite Field of size 2,
2: Vector space of dimension 1 over Finite Field of size 2}
```

RandomComplex(*n, d, p=0.5*)

A random d-dimensional simplicial complex on n vertices.

INPUT:

- n - number of vertices
- d - dimension of the complex
- p - floating point number between 0 and 1 (optional, default 0.5)

A random d-dimensional simplicial complex on n vertices, as defined for example by Meshulam and Wallach, is constructed as follows: take n vertices and include all of the simplices of dimension strictly less than d, and then for each possible simplex of dimension d, include it with probability p.

EXAMPLES:

```
sage: simplicial_complexes.RandomComplex(6, 2)
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5, 6) and 15 facets
```

If d is too large (if $d > n + 1$, so that there are no d-dimensional simplices), then return the simplicial complex with a single $(n + 1)$-dimensional simplex:

```
sage: simplicial_complexes.RandomComplex(6,12)
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5, 6, 7) and facets {(0, 1, 2, 3, 4, 5, 6
```

REFERENCES:

- Meshulam and Wallach, "Homological connectivity of random k-dimensional complexes", preprint, math.CO/0609773.

Simplex(*n*)

An n-dimensional simplex, as a simplicial complex.

INPUT:

- n - a non-negative integer

OUTPUT: the simplicial complex consisting of the n-simplex on vertices $(0, 1, ..., n)$ and all of its faces.

EXAMPLES:

```
sage: simplicial_complexes.Simplex(3)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 1, 2, 3)}
sage: simplicial_complexes.Simplex(5).euler_characteristic()
1
```

Sphere(*n*)

A minimal triangulation of the n-dimensional sphere.

INPUT:

- n - positive integer

EXAMPLES:

```
sage: simplicial_complexes.Sphere(2)
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (0, 1, 2), (1, 2, 3),
sage: simplicial_complexes.Sphere(5).homology()
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: Z}
sage: [simplicial_complexes.Sphere(n).euler_characteristic() for n in range(6)]
[2, 0, 2, 0, 2, 0]
sage: [simplicial_complexes.Sphere(n).f_vector() for n in range(6)]
[[1, 2],
 [1, 3, 3],
 [1, 4, 6, 4],
 [1, 5, 10, 10, 5],
 [1, 6, 15, 20, 15, 6],
 [1, 7, 21, 35, 35, 21, 7]]
```

SurfaceOfGenus(*g, orientable=True*)

A surface of genus g.

INPUT:

- `g` - a non-negative integer. The desired genus
- `orientable` - boolean (optional, default True). If True, return an orientable surface, and if False, return a non-orientable surface.

In the orientable case, return a sphere if g is zero, and otherwise return a g-fold connected sum of a torus with itself.

In the non-orientable case, raise an error if g is zero. If g is positive, return a g-fold connected sum of a projective plane with itself.

EXAMPLES:

```
sage: simplicial_complexes.SurfaceOfGenus(2)
Simplicial complex with 11 vertices and 26 facets
sage: simplicial_complexes.SurfaceOfGenus(1, orientable=False)
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and 10 facets
```

Torus()

A minimal triangulation of the torus.

EXAMPLES:

```
sage: simplicial_complexes.Torus().homology(1)
Z x Z
```

class **SimplicialSurface**(*complex, faces=, []*)

A simplicial surface is a simplicial complex structure on a surface without boundary. (The only difference between this and a SimplicialComplex is that there is a connected sum operation defined for these surfaces.)

This can either take one argument, a simplicial complex, or two arguments, the usual arguments to define a simplicial complex. It does almost no error checking to see whether the resulting complex actually represents a surface.

INPUT:

- `complex` - either a SimplicialComplex or a vertex set suitable for defining a simplicial complex.
- **faces - (optional, default []). If the argument complex is** a simplicial complex, then the faces argument is ignored. If complex is a vertex set, then faces is treated as the list of maximal faces to define the simplicial complex.

OUTPUT: a simplicial complex representing a surface.

EXAMPLES:

```
sage: from sage.homology.examples import SimplicialSurface
sage: sphere = SimplicialSurface(3, [[1,2,3], [0,2,3], [0,1,3], [0,1,2]])
sage: sphere
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (0, 1, 2), (1, 2, 3), (0,
sage: sphere == loads(dumps(sphere))
True
```

connected_sum(*other*)

The connected sum of this simplicial surface with another one.

The connected sum of two surfaces is defined as follows: remove a 2-simplex from each, producing a boundary for each surface consisting of three edges, and then glue the two surfaces together along this triangular boundary.

INPUT:

- `other` - the other simplicial surface.

OUTPUT: the connected sum (self # other) as a simplicial surface

Algorithm: a facet is chosen from each surface, and removed. The vertices of these two facets are relabeled to (0,1,2). Of the remaining vertices, the ones from the left-hand factor are renamed by prepending an "L", and similarly the remaining vertices in the right-hand factor are renamed by prepending an "R".

EXAMPLES:

```
sage: from sage.homology.examples import SimplicialSurface
sage: S = SimplicialSurface(3, [[1,2,3], [0,2,3], [0,1,3], [0,1,2]])
sage: S     # S is the sphere
Simplicial complex with vertex set (0, 1, 2, 3) and facets {(0, 2, 3), (0, 1, 2), (1, 2, 3),
sage: S.connected_sum(S)
Simplicial complex with vertex set (0, 1, 2, 'L0', 'R0') and 6 facets
sage: P = SimplicialSurface(5, [[0,1,2], [0,2,3], [0,1,5], [0,4,5], [0,3,4], [1,2,4], [1,3,4
sage: P     # P is the projective plane
Simplicial complex with vertex set (0, 1, 2, 3, 4, 5) and 10 facets
sage: P.connected_sum(P)    # the Klein bottle
Simplicial complex with 9 vertices and 18 facets
```

The notation '+' may be used for connected sum, also:

```
sage: P + P    # the Klein bottle
Simplicial complex with 9 vertices and 18 facets
sage: (P + P).homology()
{0: 0, 1: Z x C2, 2: 0}
```

matching(*A, B*)

List of maximal matchings between the sets A and B: a matching is a set of pairs (a,b) in A x B where each a, b appears in at most one pair. A maximal matching is one which is maximal with respect to inclusion of subsets of A x B.

INPUT:

- `A, B` - list, tuple, or indeed anything which can be converted to a set.

EXAMPLES:

```
sage: from sage.homology.examples import matching
sage: matching([1,2], [3,4])
[set([(1, 3), (2, 4)]), set([(2, 3), (1, 4)])]
sage: matching([0,2], [0])
[set([(0, 0)]), set([(2, 0)])]
```

rename_vertex(*n, keep, left=True*)

Rename a vertex: the three vertices from the list 'keep' get relabeled 0, 1, 2, in order. Any other vertex (e.g. 4) gets renamed to by prepending an 'L' or an 'R' (thus to either 'L4' or 'R4'), depending on whether the argument left is True or False.

INPUT:

- `n` - a 'vertex': either an integer or a string
- `keep` - a list of three vertices
- `left` - boolean (optional, default True)

This is used by the `connected_sum` method for simplicial surfaces.

EXAMPLES:

```
sage: from sage.homology.examples import rename_vertex
sage: rename_vertex(6, [5, 6, 7])
1
```

```
sage: rename_vertex(3, [5, 6, 7])
'L3'
sage: rename_vertex(3, [5, 6, 7], left=False)
'R3'
```

CHAPTER

THIRTYSIX

L-FUNCTIONS

Sage includes several standard open source packages for computing with L-functions.

36.1 Rubinstein's L-function Calculator

This a standard part of Sage. This interface provides complete access to Rubinstein's lcalc calculator with extra PARI functionality compiled in.

Note: Each call to `lcalc` runs a complete `lcalc` process. On a typical Linux system, this entails about 0.3 seconds overhead.

AUTHORS:

- Michael Rubinstein (2005): released under GPL the C++ program lcalc
- William Stein (2006-03-05): wrote Sage interface to lcalc

class `LCalc`()

Rubinstein's L-functions Calculator

Type `lcalc.[tab]` for a list of useful commands that are implemented using the command line interface, but return objects that make sense in Sage. For each command the possible inputs for the L-function are:

- `"` - (default) the Riemann zeta function
- `'tau'` - the L function of the Ramanujan delta function
- elliptic curve E - where E is an elliptic curve over $\mathbb{Q}$; defines $L(E, s)$

You can also use the complete command-line interface of Rubinstein's L-functions calculations program via this class. Type `lcalc.help()` for a list of commands and how to call them.

`analytic_rank`(*L=''*)

Return the analytic rank of the L-function at the central critical point.

INPUT:

- `L` - defines L-function (default: Riemann zeta function)

OUTPUT: integer

Note: Of course this is not provably correct in general, since it is an open problem to compute analytic ranks provably correctly in general.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: lcalc.analytic_rank(E)
1
```

help()

twist_values(*s, dmin, dmax, L=''*)

Return values of $L(s, \chi_k)$ for each quadratic character χ_k whose discriminant d satisfies $d_{\min} \leq d \leq d_{\max}$.

INPUT:

- `s` - complex numbers
- `dmin` - integer
- `dmax` - integer
- `L` - defines L-function (default: Riemann zeta function)

OUTPUT:

- `list` - list of pairs (d, L(s,chi_d))

EXAMPLES:

```
sage: lcalc.twist_values(0.5, -10, 10)
[(-8, 1.10042141), (-7, 1.14658567), (-4, 0.667691457), (-3, 0.480867558), (5, 0.231750947),
```

twist_zeros(*n, dmin, dmax, L=''*)

Return first n real parts of nontrivial zeros for each quadratic character χ_k whose discriminant d satisfies $d_{\min} \leq d \leq d_{\max}$.

INPUT:

- `n` - integer
- `dmin` - integer
- `dmax` - integer
- `L` - defines L-function (default: Riemann zeta function)

OUTPUT:

- `dict` - keys are the discriminants d, and values are list of corresponding zeros.

EXAMPLES:

```
sage: lcalc.twist_zeros(3, -3, 6)
{-3: [8.03973716, 11.2492062, 15.7046192], 5: [6.64845335, 9.83144443, 11.9588456]}
```

value(*s, L=''*)

Return $L(s)$ for s a complex number.

INPUT:

- `s` - complex number
- `L` - defines L-function (default: Riemann zeta function)

EXAMPLES:

```
sage: I = CC.0
sage: lcalc.value(0.5 + 100*I)
2.69261989 - 0.0203860296*I
```

Note, Sage can also compute zeta at complex numbers (using the PARI C library):

```
sage: (0.5 + 100*I).zeta()
2.69261988568132 - 0.0203860296025982*I
```

values_along_line(*s0, s1, number_samples, L=''*)

Return values of $L(s)$ at `number_samples` equally-spaced sample points along the line from s_0 to s_1 in the complex plane.

INPUT:

- `s0, s1` - complex numbers
- `number_samples` - integer

- L - defines L-function (default: Riemann zeta function)

OUTPUT:

- list - list of pairs (s, zeta(s)), where the s are equally spaced sampled points on the line from s0 to s1.

EXAMPLES:

```
sage: I = CC.0
sage: lcalc.values_along_line(0.5, 0.5+20*I, 5)
[(0.500000000, -1.46035451), (0.500000000 + 4.00000000*I, 0.606783764 + 0.0911121400*I), (0.
```

Sometimes warnings are printed (by lcalc) when this command is run:

```
sage: E = EllipticCurve('389a')
sage: E.lseries().values_along_line(0.5, 3, 5)
[(0, 0.209951303),
 (0.500000000, -...e-16),
 (1.00000000, 0.133768433),
 (1.50000000, 0.360092864),
 (2.00000000, 0.552975867)]
```

zeros (*n, L=''*)

Return the imaginary parts of the first n nontrivial zeros of the L-function in the upper half plane, as 32-bit reals.

INPUT:

- n - integer
- L - defines L-function (default: Riemann zeta function)

This function also checks the Riemann Hypothesis and makes sure no zeros are missed. This means it looks for several dozen zeros to make sure none have been missed before outputting any zeros at all, so takes longer than `self.zeros_of_zeta_in_interval(...)`.

EXAMPLES:

```
sage: lcalc.zeros(4)                          # long
[14.1347251, 21.0220396, 25.0108576, 30.4248761]
sage: lcalc.zeros(5, L='--tau')               # long
[9.22237940, 13.9075499, 17.4427770, 19.6565131, 22.3361036]
sage: lcalc.zeros(3, EllipticCurve('37a'))    # long
[0.000000000, 5.00317001, 6.87039122]
```

zeros_in_interval (*x, y, stepsize, L=''*)

Return the imaginary parts of (most of) the nontrivial zeros of the L-function on the line $\Re(s) = 1/2$ with positive imaginary part between x and y, along with a technical quantity for each.

INPUT:

- x, y, stepsize - positive floating point numbers
- L - defines L-function (default: Riemann zeta function)

OUTPUT: list of pairs (zero, S(T)).

Rubinstein writes: The first column outputs the imaginary part of the zero, the second column a quantity related to $S(T)$ (it increases roughly by 2 whenever a sign change, i.e. pair of zeros, is missed). Higher up the critical strip you should use a smaller stepsize so as not to miss zeros.

EXAMPLES:

```
sage: lcalc.zeros_in_interval(10, 30, 0.1)
[(14.1347251, 0.184672916), (21.0220396, -0.0677893290), (25.0108576, -0.0555872781)]
```

36.2 Watkins Symmetric Power L-function Calculator

SYMPOW is a package to compute special values of symmetric power elliptic curve L-functions. It can compute up to about 64 digits of precision. This interface provides complete access to sympow, which is a standard part of Sage (and includes the extra data files).

Note: Each call to `sympow` runs a complete `sympow` process. This incurs about 0.2 seconds overhead.

AUTHORS:

- Mark Watkins (2005-2006): wrote and released sympow
- William Stein (2006-03-05): wrote Sage interface

ACKNOWLEDGEMENT (from sympow readme):

- The quad-double package was modified from David Bailey's package: http://crd.lbl.gov/~dhbailey/mpdist/
- The `squfof` implementation was modified from Allan Steel's version of Arjen Lenstra's original LIP-based code.
- The `ec_ap` code was originally written for the kernel of MAGMA, but was modified to use small integers when possible.
- SYMPOW was originally developed using PARI, but due to licensing difficulties, this was eliminated. SYMPOW also does not use the standard math libraries unless Configure is run with the -lm option. SYMPOW still uses GP to compute the meshes of inverse Mellin transforms (this is done when a new symmetric power is added to datafiles).

class **Sympow**()

Watkins Symmetric Power L-function Calculator

Type `sympow.[tab]` for a list of useful commands that are implemented using the command line interface, but return objects that make sense in Sage.

You can also use the complete command-line interface of sympow via this class. Type `sympow.help()` for a list of commands and how to call them.

L(*E, n, prec*)

Return $L(\mathrm{Sym}^{(n)}(E, \text{edge}))$ to prec digits of precision, where edge is the *right* edge. Here n must be even.

INPUT:

- `E` - elliptic curve
- `n` - even integer
- `prec` - integer

OUTPUT:

- `string` - real number to prec digits of precision as a string.

Note: Before using this function for the first time for a given n, you may have to type `sympow('-new_data n')`, where `n` is replaced by your value of n.

If you would like to see the extensive output sympow prints when running this function, just type `set_verbose(2)`.

EXAMPLES:

```
sage: a = sympow.L(EllipticCurve('11a'), 2, 16); a   # optional
'1.057599244590958E+00'
sage: RR(a)                    # optional -- requires precomputations
1.05759924459096
```

Lderivs(*E, n, prec, d*)

Return 0^{th} to d^{th} derivatives of $L(\mathrm{Sym}^{(n)}(E,s)$ to prec digits of precision, where s is the right edge if n is even and the center if n is odd.

INPUT:

- `E` - elliptic curve
- `n` - integer (even or odd)
- `prec` - integer
- `d` - integer

OUTPUT: a string, exactly as output by sympow

Note: To use this function you may have to run a few commands like `sympow('-new_data 1d2')`, each which takes a few minutes. If this function fails it will indicate what commands have to be run.

EXAMPLES:

```
sage: print sympow.Lderivs(EllipticCurve('11a'), 1, 16, 2)  # not tested
...
 1n0: 2.538418608559107E-01
 1w0: 2.538418608559108E-01
 1n1: 1.032321840884568E-01
 1w1: 1.059251499158892E-01
 1n2: 3.238743180659171E-02
 1w2: 3.414818600982502E-02
```

analytic_rank(*E*)

Return the analytic rank and leading L-value of the elliptic curve E.

INPUT:

- `E` - elliptic curve over Q

OUTPUT:

- `integer` - analytic rank
- `string` - leading coefficient (as string)

Note: The analytic rank is *not* computed provably correctly in general.

Note: In computing the analytic rank we consider $L^{(r)}(E,1)$ to be 0 if $L^{(r)}(E,1)/\Omega_E > 0.0001$.

EXAMPLES: We compute the analytic ranks of the lowest known conductor curves of the first few ranks:

```
sage: sympow.analytic_rank(EllipticCurve('11a'))
(0, '2.53842e-01')
sage: sympow.analytic_rank(EllipticCurve('37a'))
(1, '3.06000e-01')
sage: sympow.analytic_rank(EllipticCurve('389a'))
(2, '7.59317e-01')
sage: sympow.analytic_rank(EllipticCurve('5077a'))
(3, '1.73185e+00')
sage: sympow.analytic_rank(EllipticCurve([1, -1, 0, -79, 289]))
(4, '8.94385e+00')
sage: sympow.analytic_rank(EllipticCurve([0, 0, 1, -79, 342]))  # long
(5, '3.02857e+01')
sage: sympow.analytic_rank(EllipticCurve([1, 1, 0, -2582, 48720]))  # long
(6, '3.20781e+02')
sage: sympow.analytic_rank(EllipticCurve([0, 0, 0, -10012, 346900]))  # long
(7, '1.32517e+03')
```

help()

modular_degree(*E*)

Return the modular degree of the elliptic curve E, assuming the Stevens conjecture.

INPUT:

- `E` - elliptic curve over Q

OUTPUT:

- `integer` - modular degree

EXAMPLES: We compute the modular degrees of the lowest known conductor curves of the first few ranks:

```
sage: sympow.modular_degree(EllipticCurve('11a'))
1
sage: sympow.modular_degree(EllipticCurve('37a'))
2
sage: sympow.modular_degree(EllipticCurve('389a'))
40
sage: sympow.modular_degree(EllipticCurve('5077a'))
1984
sage: sympow.modular_degree(EllipticCurve([1, -1, 0, -79, 289]))
334976
```

new_data(*n*)

Pre-compute data files needed for computation of n-th symmetric powers.

36.3 Dokchitser's L-functions Calculator

AUTHORS:

- Tim Dokchitser (2002): original PARI code and algorithm (and the documentation below is based on Dokchitser's docs).
- William Stein (2006-03-08): Sage interface

TODO:

- add more examples from data/extcode/pari/dokchitser that illustrate use with Eisenstein series, number fields, etc.
- plug this code into number fields and modular forms code (elliptic curves are done).

class Dokchitser(*conductor, gammaV, weight, eps, poles=, [], residues='automatic', prec=53, init=None*)

Dokchitser's L-functions Calculator

Create a Dokchitser L-series with

Dokchitser(conductor, gammaV, weight, eps, poles, residues, init, prec)

where

- `conductor` - integer, the conductor
- `gammaV` - list of Gamma-factor parameters, e.g. [0] for Riemann zeta, [0,1] for ell.curves, (see examples).
- `weight` - positive real number, usually an integer e.g. 1 for Riemann zeta, 2 for H^1 of curves/$\mathbf{Q}$
- `eps` - complex number; sign in functional equation
- `poles` - (default: []) list of points where $L^*(s)$ has (simple) poles; only poles with $Re(s) > weight/2$ should be included
- `residues` - vector of residues of $L^*(s)$ in those poles or set residues='automatic' (default value)
- `prec` - integer (default: 53) number of *bits* of precision

RIEMANN ZETA FUNCTION:

We compute with the Riemann Zeta function.

```
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1], residues=[-1], init='1
sage: L
Dokchitser L-series of conductor 1 and weight 1
sage: L(1)
...
ArithmeticError:   ###   user error: L*(s) has a pole at s=1.000000000000000000
sage: L(2)
1.64493406684823
sage: L(2, 1.1)
1.64493406684823
sage: L.derivative(2)
-0.937548254315844
sage: h = RR('0.0000000000001')
sage: (zeta(2+h) - zeta(2))/h
-0.937028232783632
sage: L.taylor_series(2, k=5)
1.64493406684823 - 0.937548254315844*z + 0.994640117149451*z^2 - 1.00002430047384*z^3 + 1.000061
```

RANK 1 ELLIPTIC CURVE:

We compute with the L-series of a rank 1 curve.

```
sage: E = EllipticCurve('37a')
sage: L = E.lseries().dokchitser(); L
Dokchitser L-function associated to Elliptic Curve defined by y^2 + y = x^3 - x over Rational Fi
sage: L(1)
0
sage: L.derivative(1)
0.305999773834052
sage: L.derivative(1,2)
0.373095594536324
sage: L.num_coeffs()
48
sage: L.taylor_series(1,4)
0.305999773834052*z + 0.186547797268162*z^2 - 0.136791463097188*z^3 + O(z^4)
sage: L.check_functional_equation()
6.11218974800000e-18                          # 32-bit
6.04442711160669e-18                          # 64-bit
```

RANK 2 ELLIPTIC CURVE:

We compute the leading coefficient and Taylor expansion of the L-series of a rank 2 curve.

```
sage: E = EllipticCurve('389a')
sage: L = E.lseries().dokchitser()
sage: L.num_coeffs ()
156
sage: L.derivative(1,E.rank())
1.51863300057685
sage: L.taylor_series(1,4)
-1.28158145691931e-23 + (7.26268290635587e-24)*z + 0.759316500288427*z^2 - 0.430302337583362*z^3
-2.69129566562797e-23 + (1.52514901968783e-23)*z + 0.759316500288427*z^2 - 0.430302337583362*z^3
```

RAMANUJAN DELTA L-FUNCTION:

The coefficients are given by Ramanujan's tau function:

```
sage: L = Dokchitser(conductor=1, gammaV=[0,1], weight=12, eps=1)
sage: pari_precode = 'tau(n)=(5*sigma(n,3)+7*sigma(n,5))*n/12 - 35*sum(k=1,n-1,(6*k-4*(n-k))*sig
sage: L.init_coeffs('tau(k)', pari_precode=pari_precode)
```

We redefine the default bound on the coefficients: Deligne's estimate on tau(n) is better than the default coefgrow(n)='(4n)^{11/2}' (by a factor 1024), so re-defining coefgrow() improves efficiency (slightly faster).

```
sage: L.num_coeffs()
12
sage: L.set_coeff_growth('2*n^(11/2)')
sage: L.num_coeffs()
11
```

Now we're ready to evaluate, etc.

```
sage: L(1)
0.0374412812685155
sage: L(1, 1.1)
0.0374412812685155
sage: L.taylor_series(1,3)
0.0374412812685155 + 0.0709221123619322*z + 0.0380744761270520*z^2 + O(z^3)
```

check_functional_equation(*T=1.2*)

Verifies how well numerically the functional equation is satisfied, and also determines the residues if `self.poles != []` and residues='automatic'.

More specifically: for $T > 1$ (default 1.2), `self.check_functional_equation(T)` should ideally return 0 (to the current precision).

- if what this function returns does not look like 0 at all, probably the functional equation is wrong (i.e. some of the parameters gammaV, conductor etc., or the coefficients are wrong),
- if checkfeq(T) is to be used, more coefficients have to be generated (approximately T times more), e.g. call cflength(1.3), initLdata("a(k)",1.3), checkfeq(1.3)
- T=1 always (!) returns 0, so T has to be away from 1
- default value $T = 1.2$ seems to give a reasonable balance
- if you don't have to verify the functional equation or the L-values, call num_coeffs(1) and initLdata("a(k)",1), you need slightly less coefficients.

EXAMPLES:

```
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1], residues=[-1], ini
sage: L.check_functional_equation ()
-2.71050543200000e-20                          # 32-bit
-2.71050543121376e-20                          # 64-bit
```

If we choose the sign in functional equation for the ζ function incorrectly, the functional equation doesn't check out.

```
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=-11, poles=[1], residues=[-1], i
sage: L.check_functional_equation ()
-9.73967861488124
```

derivative(*s, k=1*)

Return the k-th derivative of the L-series at s.

Warning: If k is greater than the order of vanishing of L at s you may get nonsense.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: L = E.lseries().dokchitser()
sage: L.derivative(1,E.rank())
1.51863300057685
```

gp()

Return the gp interpreter that is used to implement this Dokchitser L-function.

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: L = E.lseries().dokchitser()
sage: L(2)
0.546048036215014
sage: L.gp()
GP/PARI interpreter
```

init_coeffs(*v, cutoff=1, w=None, pari_precode='', max_imaginary_part=0, max_asymp_coeffs=40*)

Set the coefficients a_n of the L-series. If $L(s)$ is not equal to its dual, pass the coefficients of the dual as the second optional argument.

INPUT:

- `v` - list of complex numbers or string (pari function of k)
- `cutoff` - real number = 1 (default: 1)
- `w` - list of complex numbers or string (pari function of k)
- `pari_precode` - some code to execute in pari before calling initLdata
- `max_imaginary_part` - (default: 0): redefine if you want to compute L(s) for s having large imaginary part,
- `max_asymp_coeffs` - (default: 40): at most this many terms are generated in asymptotic series for phi(t) and G(s,t).

EXAMPLES:

```
sage: L = Dokchitser(conductor=1, gammaV=[0,1], weight=12, eps=1)
sage: pari_precode = 'tau(n)=(5*sigma(n,3)+7*sigma(n,5))*n/12 - 35*sum(k=1,n-1,(6*k-4*(n-k))
sage: L.init_coeffs('tau(k)', pari_precode=pari_precode)
```

num_coeffs(*T=1*)

Return number of coefficients a_n that are needed in order to perform most relevant L-function computations to the desired precision.

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: L = E.lseries().dokchitser()
sage: L.num_coeffs()
26
sage: E = EllipticCurve('5077a')
sage: L = E.lseries().dokchitser()
sage: L.num_coeffs()
568
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1], residues=[-1], ini
sage: L.num_coeffs()
4
```

set_coeff_growth(*coefgrow*)

You might have to redefine the coefficient growth function if the a_n of the L-series are not given by the following PARI function:

```
coefgrow(n) = if(length(Lpoles),
                 1.5*n^(vecmax(real(Lpoles))-1),
                 sqrt(4*n)^(weight-1));
```

INPUT:

- `coefgrow` - string that evaluates to a PARI function of n that defines a coefgrow function.

EXAMPLE:

```
sage: L = Dokchitser(conductor=1, gammaV=[0,1], weight=12, eps=1)
sage: pari_precode = 'tau(n)=(5*sigma(n,3)+7*sigma(n,5))*n/12 - 35*sum(k=1,n-1,(6*k-4*(n-k))
sage: L.init_coeffs('tau(k)', pari_precode=pari_precode)
sage: L.set_coeff_growth('2*n^(11/2)')
sage: L(1)
0.0374412812685155
```

taylor_series(*a=0, k=6, var='z'*)

Return the first k terms of the Taylor series expansion of the L-series about a.

This is returned as a series in `var`, where you should view `var` as equal to $s - a$. Thus this function returns the formal power series whose coefficients are $L^{(n)}(a)/n!$.

INPUT:

- `a` - complex number (default: 0); point about which to expand
- `k` - integer (default: 6), series is $O(\text{“}var\text{“}^k)$
- `var` - string (default: 'z'), variable of power series

EXAMPLES:

```
sage: L = Dokchitser(conductor=1, gammaV=[0], weight=1, eps=1, poles=[1], residues=[-1], ini
sage: L.taylor_series(2, 3)
1.64493406684823 - 0.937548254315844*z + 0.994640117149451*z^2 + O(z^3)
sage: E = EllipticCurve('37a')
sage: L = E.lseries().dokchitser()
sage: L.taylor_series(1)
0.305999773834052*z + 0.186547797268162*z^2 - 0.136791463097188*z^3 + 0.0161066468496401*z^4
```

We compute a Taylor series where each coefficient is to high precision.

```
sage: E = EllipticCurve('389a')
sage: L = E.lseries().dokchitser(200)
sage: L.taylor_series(1,3)
6.2239725530250970363983975962696997888173850098274602272589e-73 + (-3.527106203544994604921
6.2239725530250970363983942830358807065824196704980264311105e-73 + (-3.527106203544994604921
```

reduce_load_dokchitser(*D*)

CHAPTER

THIRTYSEVEN

SCHEMES

37.1 Scheme implementation overview

Various parts of schemes were implemented by David Kohel, David Joyner, and William Stein.

AUTHORS:

- David Kohel (2006-01-03): initial version
- William Stein (2006-01-05)
- William Stein (2006-01-20)

- **Scheme:** A scheme whose datatype might be not be defined in terms of algebraic equations: e.g. the Jacobian of a curve may be represented by means of a Scheme.
- **AlgebraicScheme:** A scheme defined by means of polynomial equations, which may be reducible or defined over a ring other than a field. In particular, the defining ideal need not be a radical ideal, and an algebraic scheme may be defined over Spec(R).
- **AmbientSpaces:** Most effective models of algebraic scheme will be defined, not by generic gluings, but by embeddings in some fixed ambient space.
- **AffineSpace:** Affine spaces, and their affine subschemes form the most important universal objects from which algebraic schemes are built. The affine spaces form universal objects in the sense that a morphism is uniquely determined by the images of its coordinate functions and any such images determine a well-defined morphism.

  By default affine spaces will embed in some ordinary projective space, unless it is created as an affine patch of another object.
- **ProjectiveSpace:** The projective spaces are the most natural ambient spaces for most projective objects. They are locally universal objects.
- **ProjectiveSpace_ordinary (not implemented)** The ordinary projective spaces have the standard weights $[1, .., 1]$ on their coefficients.
- **ProjectiveSpace_weighted (not implemented):** A special subtype for non-standard weights.
- **ToricSpace (not implemented):** This defines a projective toric variety, which defines a space equipped with a toral action and certain defining data. These generalise projective spaces, but it is not envisioned that the latter should inherit from the `ToricSpace` type.
- **AlgebraicScheme_subscheme_affine:** An algebraic scheme defined by means of an embedding in a fixed ambient affine space.

- **AlgebraicScheme_subscheme_projective:** An algebraic scheme defined by means of an embedding in a fixed ambient projective space.
- **QuasiAffineScheme (not yet implemented):** An open subset $U = X \setminus Z$ of a closed subset X of affine space; note that this is mathematically a quasi-projective scheme, but its ambient space is an affine space and its points are represented by affine rather than projective points.

  Note: AlgebraicScheme_quasi is implemented, as a base class for this.
- **QuasiProjectiveScheme (not yet implemented):** An open subset of a closed subset of projective space; this datatype stores the defining polynomial, polynomials, or ideal defining the projective closure X plus the closed subscheme Z of X whose complement $U = X \setminus Z$ is the quasi-projective scheme.

  Note: the quasi-affine and quasi-projective datatype lets one create schemes like the multiplicative group scheme $\mathbb{G}_m = \mathbb{A}^1 \setminus \{(0)\}$ and the non-affine scheme $\mathbb{A}^2 \setminus \{(0,0)\}$. The latter is not affine and is not of the form $\mathrm{Spec}(R)$.

37.1.1 TODO List

- **PointSets and points over a ring:** For algebraic schemes X/S and T/S over S, one can form the point set $X(T)$ of morphisms from $T \to X$ over S.

  ```
  sage: PP.<X,Y,Z> = ProjectiveSpace(2, QQ)
  sage: PP
  Projective Space of dimension 2 over Rational Field
  ```

  The last line is an abuse of language – returning the generators of the coordinate ring by `gens()`.

  A projective space object in the category of schemes is a locally free object – the images of the generator functions *locally* determine a point. Over a field, one can choose one of the standard affine patches by the condition that a coordinate function $X_i \neq 0$

  ```
  sage: PP(QQ)
  Set of Rational Points of Projective Space of dimension 2 over Rational Field
  sage: PP(QQ)([-2,3,5])
  (-2/5 : 3/5 : 1)
  ```

  Over a ring, this is not true, e.g. even over an integral domain which is not a PID, there may be no *single* affine patch which covers a point.

  ```
  sage: R.<x> = ZZ[]
  sage: S.<t> = R.quo(x^2+5)
  sage: P.<X,Y,Z> = ProjectiveSpace(2, S)
  sage: P(S)
  Set of Rational Points of Projective Space of dimension 2 over
  Univariate Quotient Polynomial Ring in t over Integer Ring with
  modulus x^2 + 5
  ```

  In order to represent the projective point $(2 : 1+t) = (1-t : 3)$ we note that the first representative is not well-defined at the prime $pp = (2, 1+t)$ and the second element is not well-defined at the prime $qq = (1-t, 3)$, but that $pp + qq = (1)$, so globally the pair of coordinate representatives is well-defined.

  ```
  sage: P( [2, 1+t] )
  ...
  NotImplementedError
  ```

  In fact, we need a test `R.ideal([2,1+t]) == R.ideal([1])` in order to make this meaningful.

37.2 Schemes

AUTHORS:

- William Stein, David Kohel, Kiran Kedlaya (2008): added zeta_series

class AffineScheme(*X*)

An abstract affine scheme.

hom(*x*, *Y=None*)

Return the scheme morphism from self to Y defined by x.

If Y is not given, try to determine from context.

EXAMPLES: We construct the inclusion from $\mathrm{Spec}(\mathbf{Q})$ into $\mathrm{Spec}(\mathbf{Z})$ induced by the inclusion from $\mathbf{Z}$ into $\mathbf{Q}$.

```
sage: X = Spec(QQ)
sage: X.hom(ZZ.hom(QQ))
Affine Scheme morphism:
  From: Spectrum of Rational Field
  To:   Spectrum of Integer Ring
  Defn: Ring Coercion morphism:
          From: Integer Ring
          To:   Rational Field
```

class Scheme(*X*)

base_extend(*Y*)

Y is either a scheme in the same category as self or a ring.

EXAMPLES:

```
sage: X = Spec(QQ)
sage: X.base_scheme()
Spectrum of Integer Ring
sage: X.base_extend(QQ)
...
NotImplementedError
```

base_morphism()

Return the structure morphism from the scheme self to its base scheme.

EXAMPLES:

```
sage: A = AffineSpace(4, QQ)
sage: A.base_morphism()
Scheme morphism:
  From: Affine Space of dimension 4 over Rational Field
  To:   Spectrum of Rational Field
  Defn: Structure map

sage: X = Spec(QQ)
sage: X.base_morphism()
Scheme morphism:
  From: Spectrum of Rational Field
  To:   Spectrum of Integer Ring
  Defn: Structure map
```

base_ring()

Return the base ring of the scheme self.

EXAMPLES:

```
sage: A = AffineSpace(4, QQ)
sage: A.base_ring()
Rational Field

sage: X = Spec(QQ)
sage: X.base_ring()
Integer Ring
```

base_scheme()

Return the base scheme of the scheme self.

EXAMPLES:

```
sage: A = AffineSpace(4, QQ)
sage: A.base_scheme()
Spectrum of Rational Field

sage: X = Spec(QQ)
sage: X.base_scheme()
Spectrum of Integer Ring
```

category()

Return the category to which this scheme belongs. This is the category of all schemes over the base scheme of self.

EXAMPLES:

```
sage: ProjectiveSpace(4, QQ).category()
Category of schemes over Spectrum of Rational Field
```

coordinate_ring()

Return the coordinate ring of this scheme, if defined. Otherwise raise a ValueError.

EXAMPLES:

```
sage: R.<x, y> = QQ[]
sage: I = (x^2 - y^2)*R
sage: X = Spec(R.quotient(I))
sage: X.coordinate_ring()
Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the ideal (x^2 - y^2
```

count_points(n)

Count points over $\mathbf{F}_q, \ldots, \mathbf{F}_{q^n}$ on a scheme over a finite field $\mathbf{F}_q$.

Note: This is currently only implemented for schemes over prime order finite fields.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3))
sage: C = HyperellipticCurve(x^3+x^2+1)
sage: C.count_points(4)
[6, 12, 18, 96]
sage: C.base_extend(GF(9,'a')).count_points(2)
...
NotImplementedError: Point counting only implemented for schemes over prime fields
```

dimension()

Return the absolute dimension of this scheme.

EXAMPLES:

```
sage: R.<x, y> = QQ[]
sage: I = (x^2 - y^2)*R
sage: X = Spec(R.quotient(I))
```

```
sage: X.dimension_absolute()
...
NotImplementedError
sage: X.dimension()
...
NotImplementedError
```

dimension_absolute()

Return the absolute dimension of this scheme.

EXAMPLES:

```
sage: R.<x, y> = QQ[]
sage: I = (x^2 - y^2)*R
sage: X = Spec(R.quotient(I))
sage: X.dimension_absolute()
...
NotImplementedError
sage: X.dimension()
...
NotImplementedError
```

dimension_relative()

Return the relative dimension of this scheme over its base.

EXAMPLES:

```
sage: R.<x, y> = QQ[]
sage: I = (x^2 - y^2)*R
sage: X = Spec(R.quotient(I))
sage: X.dimension_relative()
...
NotImplementedError
```

hom(*x, Y=None*)

Return the scheme morphism from self to Y defined by x. If x is a scheme, try to determine a natural map to x.

If Y is not given, try to determine Y from context.

EXAMPLES:

```
sage: P = ProjectiveSpace(ZZ, 3)
sage: P.hom(Spec(ZZ))
Scheme morphism:
  From: Projective Space of dimension 3 over Integer Ring
  To:   Spectrum of Integer Ring
  Defn: Structure map
```

identity_morphism()

Return the identity morphism of the scheme self.

EXAMPLES:

```
sage: X = Spec(QQ)
sage: X.identity_morphism()
Scheme endomorphism of Spectrum of Rational Field
  Defn: Identity map
```

point(*v, check=True*)

point_homset(*S=None*)

Return the set of S-valued points of this scheme.

EXAMPLES:

```
sage: P = ProjectiveSpace(ZZ, 3)
sage: P.point_homset(ZZ)
Set of Rational Points of Projective Space of dimension 3 over Integer Ring
sage: P.point_homset(QQ)
Set of Rational Points of Projective Space of dimension 3 over Rational Field
sage: P.point_homset(GF(11))
Set of Rational Points of Projective Space of dimension 3 over Finite Field of size 11
```

point_set(*S=None*)

Return the set of S-valued points of this scheme.

EXAMPLES:

```
sage: P = ProjectiveSpace(ZZ, 3)
sage: P.point_homset(ZZ)
Set of Rational Points of Projective Space of dimension 3 over Integer Ring
sage: P.point_homset(QQ)
Set of Rational Points of Projective Space of dimension 3 over Rational Field
sage: P.point_homset(GF(11))
Set of Rational Points of Projective Space of dimension 3 over Finite Field of size 11
```

structure_morphism()

Return the structure morphism from the scheme self to its base scheme.

EXAMPLES:

```
sage: A = AffineSpace(4, QQ)
sage: A.base_morphism()
Scheme morphism:
  From: Affine Space of dimension 4 over Rational Field
  To:   Spectrum of Rational Field
  Defn: Structure map

sage: X = Spec(QQ)
sage: X.base_morphism()
Scheme morphism:
  From: Spectrum of Rational Field
  To:   Spectrum of Integer Ring
  Defn: Structure map
```

union(*X*)

Return the disjoint union of the schemes self and X.

EXAMPLES:

```
sage: S = Spec(QQ)
sage: X = AffineSpace(1, QQ)
sage: S.union(X)
...
NotImplementedError
```

zeta_series(*n, t*)

Compute a power series approximation to the zeta function of a scheme over a finite field.

INPUT:

- `n` - the number of terms of the power series to compute
- `t` - the variable which the series should be returned

OUTPUT: A power series approximating the zeta function of self

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3))
sage: C = HyperellipticCurve(x^3+x^2+1)
sage: R.<t> = PowerSeriesRing(Integers())
sage: C.zeta_series(4,t)
1 + 6*t + 24*t^2 + 78*t^3 + 240*t^4 + O(t^5)
sage: (1+2*t+3*t^2)/(1-t)/(1-3*t) + O(t^5)
1 + 6*t + 24*t^2 + 78*t^3 + 240*t^4 + O(t^5)
```

Note that this function depends on count_points, which is only defined for prime order fields:

```
sage: C.base_extend(GF(9,'a')).zeta_series(4,t)
...
NotImplementedError: Point counting only implemented for schemes over prime fields
```

is_AffineScheme(*x*)

Return True if x is an affine scheme.

EXAMPLES:

```
sage: from sage.schemes.generic.scheme import is_AffineScheme
sage: is_AffineScheme(5)
False
sage: E = Spec(QQ)
sage: is_AffineScheme(E)
True
```

is_Scheme(*x*)

Return True if x is a scheme.

EXAMPLES:

```
sage: from sage.schemes.generic.scheme import is_Scheme
sage: is_Scheme(5)
False
sage: X = Spec(QQ)
sage: is_Scheme(X)
True
```

37.3 Spec of a ring

class Spec(*R, S=None, check=True*)

The spectrum of a commutative ring, as a scheme.

Note: Calling `Spec(R)` twice produces two distinct (but equal) schemes, which is important for gluing to construct more general schemes.

EXAMPLES:

```
sage: Spec(QQ)
Spectrum of Rational Field
sage: Spec(PolynomialRing(QQ, 'x'))
Spectrum of Univariate Polynomial Ring in x over Rational Field
sage: Spec(PolynomialRing(QQ, 'x', 3))
Spectrum of Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
sage: X = Spec(PolynomialRing(GF(49,'a'), 3, 'x')); X
Spectrum of Multivariate Polynomial Ring in x0, x1, x2 over Finite Field in a of size 7^2
sage: loads(X.dumps()) == X
True
```

```
sage: A = Spec(ZZ); B = Spec(ZZ)
sage: A is B
False
sage: A == B
True
```

A TypeError is raised if the input is not a CommutativeRing.

```
sage: Spec(5)
...
TypeError: R (=5) must be a commutative ring
sage: Spec(FreeAlgebra(QQ,2, 'x'))
...
TypeError: R (=Free Algebra on 2 generators (x0, x1) over Rational Field) must be a commutative
```

EXAMPLES:

```
sage: X = Spec(ZZ)
sage: X
Spectrum of Integer Ring
sage: X.base_scheme()
Spectrum of Integer Ring
sage: X.base_ring()
Integer Ring
sage: X.dimension()
1
```

coordinate_ring()

Return the underlying ring of this scheme.

EXAMPLES:

```
sage: Spec(QQ).coordinate_ring()
Rational Field
sage: Spec(PolynomialRing(QQ, 3, 'x')).coordinate_ring()
Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
```

dimension()

Return the absolute dimension of this scheme.

EXAMPLES:

```
sage: S = Spec(ZZ)
sage: S.dimension_absolute()
1
sage: S.dimension()
1
```

dimension_absolute()

Return the absolute dimension of this scheme.

EXAMPLES:

```
sage: S = Spec(ZZ)
sage: S.dimension_absolute()
1
sage: S.dimension()
1
```

dimension_relative()

Return the relative dimension of this scheme over its base.

EXAMPLES:

```
sage: S = Spec(ZZ)
sage: S.dimension_relative()
0
```

is_noetherian()
Return True if this scheme is Noetherian.
EXAMPLES:

```
sage: Spec(ZZ).is_noetherian()
True
```

is_Spec(*X*)
EXAMPLES:

```
sage: from sage.schemes.generic.spec import is_Spec
sage: is_Spec(QQ^3)
False
sage: X = Spec(QQ); X
Spectrum of Rational Field
sage: is_Spec(X)
True
```

37.4 Scheme obtained by glueing two other schemes

class GluedScheme(*f, g, check=True*)
INPUT:

- f - open immersion from a scheme U to a scheme X
- g - open immersion from U to a scheme Y

OUTPUT: The scheme obtained by gluing X and Y along the open set U.

Note: Checking that f and g are open immersions is not implemented.

gluing_maps()

37.5 Points on schemes

class SchemePoint(*S*)
Base class for points on a scheme, either topological or defined by a morphism.

scheme()
Return the scheme on which self is a point.
EXAMPLES:

```
sage: from sage.schemes.generic.point import SchemePoint
sage: S = Spec(ZZ)
sage: P = SchemePoint(S)
sage: P.scheme()
Spectrum of Integer Ring
```

class SchemeRationalPoint(*f*)

morphism()

class SchemeTopologicalPoint(*S*)

class SchemeTopologicalPoint_affine_open(*u, x*)

affine_open()
Return the affine open subset U.

embedding_of_affine_open()
Return the embedding from the affine open subset U into this scheme.

point_on_affine()
Return the scheme point on the affine open U.

class SchemeTopologicalPoint_prime_ideal(*S, P, check=False*)

prime_ideal()
Return the prime ideal that defines this scheme point.
EXAMPLES:

```
sage: from sage.schemes.generic.point import SchemeTopologicalPoint_prime_ideal
sage: P2.<x, y, z> = ProjectiveSpace(2, QQ)
sage: pt = SchemeTopologicalPoint_prime_ideal(P2, y*z-x^2)
sage: pt.prime_ideal()
Principal ideal (-x^2 + y*z) of Multivariate Polynomial Ring in x, y, z over Rational Field
```

is_SchemeRationalPoint(*x*)

is_SchemeTopologicalPoint(*x*)

37.6 Ambient Spaces

class AmbientSpace(*n, R=Integer Ring*)
Base class for ambient spaces over a ring.
INPUT:

- n - dimension
- R - ring

ambient_space()
Return the ambient space of the scheme self, in this case self itself.
EXAMPLES:

```
sage: P = ProjectiveSpace(4, ZZ)
sage: P.ambient_space() is P
True

sage: A = AffineSpace(2, GF(3))
sage: A.ambient_space()
Affine Space of dimension 2 over Finite Field of size 3
```

base_extend(*S, check=True*)
Return the base change of self to the ring S, via the natural map from the base ring of self to S.
A ValueError is raised if there is no natural map between the two rings.
EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: PQ = P.base_extend(QQ); PQ
Projective Space of dimension 2 over Rational Field
sage: PQ.base_extend(GF(5))
...
ValueError: No natural map from the base ring (=Rational Field) to S (=Finite Field of size
```

defining_polynomials()

Return the defining polynomials of the scheme self. Since self is an ambient space, this is an empty list.

EXAMPLES:

```
sage: ProjectiveSpace(2, QQ).defining_polynomials()
()
sage: AffineSpace(0, ZZ).defining_polynomials()
()
```

dimension()

Return the absolute dimension of this scheme.

EXAMPLES:

```
sage: A2Q = AffineSpace(2, QQ)
sage: A2Q.dimension_absolute()
2
sage: A2Q.dimension()
2
sage: A2Z = AffineSpace(2, ZZ)
sage: A2Z.dimension_absolute()
3
sage: A2Z.dimension()
3
```

dimension_absolute()

Return the absolute dimension of this scheme.

EXAMPLES:

```
sage: A2Q = AffineSpace(2, QQ)
sage: A2Q.dimension_absolute()
2
sage: A2Q.dimension()
2
sage: A2Z = AffineSpace(2, ZZ)
sage: A2Z.dimension_absolute()
3
sage: A2Z.dimension()
3
```

dimension_relative()

Return the relative dimension of this scheme over its base.

EXAMPLES:

```
sage: A2Q = AffineSpace(2, QQ)
sage: A2Q.dimension_relative()
2
sage: A2Z = AffineSpace(2, ZZ)
sage: A2Z.dimension_relative()
2
```

gen(*n=0*)

Return the n-th generator of the coordinate ring of the scheme self.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: P.gen(1)
y
```

gens()

Return the generators of the coordinate ring of the scheme self.

EXAMPLES:

```
sage: AffineSpace(0, QQ).gens()
()

sage: P.<x, y, z> = ProjectiveSpace(2, GF(5))
sage: P.gens()
(x, y, z)
```

ngens()

Return the number of generators of the coordinate ring of the scheme self.

EXAMPLES:

```
sage: AffineSpace(0, QQ).ngens()
0

sage: ProjectiveSpace(50, ZZ).ngens()
51
```

is_AmbientSpace(x)

Return True if x is an ambient space.

EXAMPLES:

```
sage: from sage.schemes.generic.ambient_space import is_AmbientSpace
sage: is_AmbientSpace(ProjectiveSpace(3, ZZ))
True
sage: is_AmbientSpace(AffineSpace(2, QQ))
True
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: is_AmbientSpace(P.subscheme([x+y+z]))
False
```

37.7 Affine n space over a ring.

AffineSpace(*n, R=None, names='x'*)

Return affine space of dimension n over the ring R.

EXAMPLES: The dimension and ring can be given in either order.

```
sage: AffineSpace(3, QQ, 'x')
Affine Space of dimension 3 over Rational Field
sage: AffineSpace(5, QQ, 'x')
Affine Space of dimension 5 over Rational Field
sage: A = AffineSpace(2, QQ, names='XY'); A
Affine Space of dimension 2 over Rational Field
sage: A.coordinate_ring()
Multivariate Polynomial Ring in X, Y over Rational Field
```

Use the divide operator for base extension.

```
sage: AffineSpace(5, names='x')/GF(17)
Affine Space of dimension 5 over Finite Field of size 17
```

The default base ring is **Z**.

```
sage: AffineSpace(5, names='x')
Affine Space of dimension 5 over Integer Ring
```

There is also an affine space associated to each polynomial ring.

```
sage: R = GF(7)['x,y,z']
sage: A = AffineSpace(R); A
Affine Space of dimension 3 over Finite Field of size 7
sage: A.coordinate_ring() is R
True
```

class **AffineSpace_generic**(*n, R, names*)

Affine space of dimension n over the ring R.

EXAMPLES:

```
sage: X.<x,y,z> = AffineSpace(3, QQ)
sage: X.base_scheme()
Spectrum of Rational Field
sage: X.base_ring()
Rational Field
sage: X.structure_morphism ()
Scheme morphism:
  From: Affine Space of dimension 3 over Rational Field
  To:   Spectrum of Rational Field
  Defn: Structure map
```

Loading and saving:

```
sage: loads(X.dumps()) == X
True
```

We create several other examples of affine spaces.

```
sage: AffineSpace(5, PolynomialRing(QQ, 'z'), 'Z')
Affine Space of dimension 5 over Univariate Polynomial Ring in z over Rational Field
```

```
sage: AffineSpace(RealField(), 3, 'Z')
Affine Space of dimension 3 over Real Field with 53 bits of precision
```

```
sage: AffineSpace(Qp(7), 2, 'x')
Affine Space of dimension 2 over 7-adic Field with capped relative precision 20
```

Even 0-dimensional affine spaces are supported.

```
sage: AffineSpace(0)
Affine Space of dimension 0 over Integer Ring
```

coordinate_ring()

Return the coordinate ring of this scheme, if defined.

EXAMPLES:

```
sage: R = AffineSpace(2, GF(9,'alpha'), 'z').coordinate_ring(); R
Multivariate Polynomial Ring in z0, z1 over Finite Field in alpha of size 3^2
sage: AffineSpace(3, R, 'x').coordinate_ring()
Multivariate Polynomial Ring in x0, x1, x2 over Multivariate Polynomial Ring in z0, z1 over
```

ngens()

Return the number of generators of self, i.e. the number of variables in the coordinate ring of self.

EXAMPLES:

```
sage: AffineSpace(3, QQ).ngens()
3
sage: AffineSpace(7, ZZ).ngens()
7
```

projective_embedding(*i=None, PP=None*)

Returns a morphism from this space into an ambient projective space of the same dimension.

INPUT:

- `i` - integer (default: dimension of self = last coordinate) determines which projective embedding to compute. The embedding is that which has a 1 in the i-th coordinate, numbered from 0.
- `PP` - (default: None) ambient projective space, i.e., codomain of morphism; this is constructed if it is not given.

EXAMPLES:

```
sage: AA = AffineSpace(2, QQ, 'x')
sage: pi = AA.projective_embedding(0); pi
Scheme morphism:
  From: Affine Space of dimension 2 over Rational Field
  To:   Projective Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x0, x1) to
        (1 : x0 : x1)
sage: z = AA(3,4)
sage: pi(z)
(1/4 : 3/4 : 1)
sage: pi(AA(0,2))
(1/2 : 0 : 1)
sage: pi = AA.projective_embedding(1); pi
Scheme morphism:
  From: Affine Space of dimension 2 over Rational Field
  To:   Projective Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x0, x1) to
        (x0 : 1 : x1)
sage: pi(z)
(3/4 : 1/4 : 1)
sage: pi = AA.projective_embedding(2)
sage: pi(z)
(3 : 4 : 1)
```

rational_points(*F=None*)

Return the list of F-rational points on the affine space self, where F is a given finite field, or the base ring of self.

EXAMPLES:

```
sage: A = AffineSpace(1, GF(3))
sage: A.rational_points()
[(0), (1), (2)]
sage: A.rational_points(GF(3^2, 'b'))
[(0), (2*b), (b + 1), (b + 2), (2), (b), (2*b + 2), (2*b + 1), (1)]
```

```
sage: AffineSpace(2, ZZ).rational_points(GF(2))
[(0, 0), (1, 0), (0, 1), (1, 1)]
```

TESTS:

```
sage: AffineSpace(2, QQ).rational_points()
...
TypeError: Base ring (= Rational Field) must be a finite field.
sage: AffineSpace(1, GF(3)).rational_points(ZZ)
...
TypeError: Second argument (= Integer Ring) must be a finite field.
```

subscheme(*X*)

Return the closed subscheme defined by X.

INPUT:

•X - a list or tuple of equations

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: X = A.subscheme([x, y^2, x*y^2]); X
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x
  y^2
  x*y^2

sage: X.defining_polynomials ()
(x, y^2, x*y^2)
sage: I = X.defining_ideal(); I
Ideal (x, y^2, x*y^2) of Multivariate Polynomial Ring in x, y over Rational Field
sage: I.groebner_basis()
[y^2, x]
sage: X.dimension()
0
sage: X.base_ring()
Rational Field
sage: X.base_scheme()
Spectrum of Rational Field
sage: X.structure_morphism()
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x
  y^2
  x*y^2
  To:   Spectrum of Rational Field
  Defn: Structure map
sage: X.dimension()
0
```

is_AffineSpace(*x*)

Returns True if x is an affine space, i.e., an ambient space $\mathbb{A}^n_R$, where R is a ring and $n \geq 0$ is an integer.

EXAMPLES:

```
sage: from sage.schemes.generic.affine_space import is_AffineSpace
sage: is_AffineSpace(AffineSpace(5, names='x'))
True
sage: is_AffineSpace(AffineSpace(5, GF(9,'alpha'), names='x'))
True
```

```
sage: is_AffineSpace(Spec(ZZ))
False
```

37.8 Projective n space over a ring.

EXAMPLES: We construct projective space over various rings of various dimensions.

The simplest projective space:

```
sage: ProjectiveSpace(0)
Projective Space of dimension 0 over Integer Ring
```

A slightly bigger projective space over $\mathbf{Q}$:

```
sage: X = ProjectiveSpace(1000, QQ); X
Projective Space of dimension 1000 over Rational Field
sage: X.dimension()
1000
```

We can use "over" notation to create projective spaces over various base rings.

```
sage: X = ProjectiveSpace(5)/QQ; X
Projective Space of dimension 5 over Rational Field
sage: X/CC
Projective Space of dimension 5 over Complex Field with 53 bits of precision
```

The third argument specifies the printing names of the generators of the homogenous coordinate ring. Using objgens() you can obtain both the space and the generators as ready to use variables.

```
sage: P2, (x,y,z) = ProjectiveSpace(2, QQ, 'xyz').objgens()
sage: P2
Projective Space of dimension 2 over Rational Field
sage: x.parent()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

For example, we use x, y, z to define the intersection of two lines.

```
sage: V = P2.subscheme([x+y+z, x+y-z]); V
Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
 x + y + z
 x + y - z
sage: V.dimension()
0
```

ProjectiveSpace(*n, R=None, names='x'*)

Return projective space of dimension n over the ring R.

EXAMPLES: The dimension and ring can be given in either order.

```
sage: ProjectiveSpace(3, QQ)
Projective Space of dimension 3 over Rational Field
sage: ProjectiveSpace(5, QQ)
Projective Space of dimension 5 over Rational Field
sage: P = ProjectiveSpace(2, QQ, names='XYZ'); P
```

```
Projective Space of dimension 2 over Rational Field
sage: P.coordinate_ring()
Multivariate Polynomial Ring in X, Y, Z over Rational Field
```

The divide operator does base extension.

```
sage: ProjectiveSpace(5)/GF(17)
Projective Space of dimension 5 over Finite Field of size 17
```

The default base ring is $\mathbf{Z}$.

```
sage: ProjectiveSpace(5)
Projective Space of dimension 5 over Integer Ring
```

There is also an projective space associated each polynomial ring.

```
sage: R = GF(7)['x,y,z']
sage: P = ProjectiveSpace(R); P
Projective Space of dimension 2 over Finite Field of size 7
sage: P.coordinate_ring()
Multivariate Polynomial Ring in x, y, z over Finite Field of size 7
sage: P.coordinate_ring() is R
True
```

Projective spaces are not cached, i.e., there can be several with the same base ring and dimension (to facilitate glueing constructions).

class **ProjectiveSpace_field**(*n, R=Integer Ring, names=None*)

class **ProjectiveSpace_finite_field**(*n, R=Integer Ring, names=None*)

rational_points(*F=None*)

Return the list of F-rational points on the affine space self, where F is a given finite field, or the base ring of self.

EXAMPLES:

```
sage: P = ProjectiveSpace(1, GF(3))
sage: P.rational_points()
[(0 : 1), (1 : 1), (2 : 1), (1 : 0)]
sage: P.rational_points(GF(3^2, 'b'))
[(0 : 1), (2*b : 1), (b + 1 : 1), (b + 2 : 1), (2 : 1), (b : 1), (2*b + 2 : 1), (2*b + 1 : 1
```

class **ProjectiveSpace_rational_field**(*n, R=Integer Ring, names=None*)

rational_points(*bound=0*)

Returns the projective points $(x_0 : \cdots : x_n)$ over $\mathbf{Q}$ with $|x_i| \leq$ bound.

INPUT:

- `bound` - integer

EXAMPLES:

```
sage: PP = ProjectiveSpace(0,QQ)
sage: PP.rational_points(1)
[(1)]
sage: PP = ProjectiveSpace(1,QQ)
sage: PP.rational_points(2)
[(-2 : 1), (-1 : 1), (0 : 1), (1 : 1), (2 : 1), (-1/2 : 1), (1/2 : 1), (1 : 0)]
sage: PP = ProjectiveSpace(2,QQ)
```

```
sage: PP.rational_points(2)
[(-2 : -2 : 1), (-1 : -2 : 1), (0 : -2 : 1), (1 : -2 : 1), (2 : -2 : 1),
(-2 : -1 : 1), (-1 : -1 : 1), (0 : -1 : 1), (1 : -1 : 1), (2 : -1 : 1),
(-2 : 0 : 1), (-1 : 0 : 1), (0 : 0 : 1), (1 : 0 : 1), (2 : 0 : 1), (-2 :
1 : 1), (-1 : 1 : 1), (0 : 1 : 1), (1 : 1 : 1), (2 : 1 : 1), (-2 : 2 :
1), (-1 : 2 : 1), (0 : 2 : 1), (1 : 2 : 1), (2 : 2 : 1), (-1/2 : -1 :
1), (1/2 : -1 : 1), (-1 : -1/2 : 1), (-1/2 : -1/2 : 1), (0 : -1/2 : 1),
(1/2 : -1/2 : 1), (1 : -1/2 : 1), (-1/2 : 0 : 1), (1/2 : 0 : 1), (-1 :
1/2 : 1), (-1/2 : 1/2 : 1), (0 : 1/2 : 1), (1/2 : 1/2 : 1), (1 : 1/2 :
1), (-1/2 : 1 : 1), (1/2 : 1 : 1), (-2 : 1 : 0), (-1 : 1 : 0), (0 : 1 :
0), (1 : 1 : 0), (2 : 1 : 0), (-1/2 : 1 : 0), (1/2 : 1 : 0), (1 : 0 :
0)]
```

Note: The very simple algorithm works as follows: every point $(x_0 : \cdots : x_n)$ in projective space has a unique largest index i for which x_i is not zero. The algorithm then iterates downward on this index. We normalize by choosing x_i positive. Then, the points $x_0, \ldots, x_{i-1}$ are the points of affine i-space that are relatively prime to x_i. We access these by using the Tuples method.

AUTHORS:

- Benjamin Antieau (2008-01-12)

class **ProjectiveSpace_ring**(*n, R=Integer Ring, names=None*)

Projective space of dimension n over the ring R.

EXAMPLES:

```
sage: X.<x,y,z,w> = ProjectiveSpace(3, QQ)
sage: X.base_scheme()
Spectrum of Rational Field
sage: X.base_ring()
Rational Field
sage: X.structure_morphism ()
Scheme morphism:
  From: Projective Space of dimension 3 over Rational Field
  To:   Spectrum of Rational Field
  Defn: Structure map
sage: X.coordinate_ring()
Multivariate Polynomial Ring in x, y, z, w over Rational Field
```

Loading and saving:

```
sage: loads(X.dumps()) == X
True
```

affine_patch(*i*)

Return the i^{th} affine patch of this projective space. This is an ambient affine space $\mathbb{A}^n_R$, where R is the base ring of self, whose "projective embedding" map is 1 in the i^{th} factor.

INPUT:

- `i` - integer between 0 and dimension of self, inclusive.

OUTPUT: an ambient affine space with fixed projective_embedding map.

EXAMPLES:

```
sage: PP = ProjectiveSpace(5) / QQ
sage: AA = PP.affine_patch(2)
sage: AA
Affine Space of dimension 5 over Rational Field
sage: AA.projective_embedding()
Scheme morphism:
```

```
    From: Affine Space of dimension 5 over Rational Field
    To:   Projective Space of dimension 5 over Rational Field
    Defn: Defined on coordinates by sending (x0, x1, x2, x3, x4) to
          (x0 : x1 : 1 : x2 : x3 : x4)
sage: AA.projective_embedding(0)
Scheme morphism:
    From: Affine Space of dimension 5 over Rational Field
    To:   Projective Space of dimension 5 over Rational Field
    Defn: Defined on coordinates by sending (x0, x1, x2, x3, x4) to
          (1 : x0 : x1 : x2 : x3 : x4)
```

coordinate_ring()

Return the coordinate ring of this scheme.

EXAMPLES:

```
sage: ProjectiveSpace(3, GF(19^2,'alpha'), 'abcd').coordinate_ring()
Multivariate Polynomial Ring in a, b, c, d over Finite Field in alpha of size 19^2

sage: ProjectiveSpace(3).coordinate_ring()
Multivariate Polynomial Ring in x0, x1, x2, x3 over Integer Ring

sage: ProjectiveSpace(2, QQ, ['alpha', 'beta', 'gamma']).coordinate_ring()
Multivariate Polynomial Ring in alpha, beta, gamma over Rational Field
```

ngens()

Return the number of generators of self, i.e. the number of variables in the coordinate ring of self.

EXAMPLES:

```
sage: ProjectiveSpace(3, QQ).ngens()
4
sage: ProjectiveSpace(7, ZZ).ngens()
8
```

subscheme(*X*)

Return the closed subscheme defined by X.

INPUT:

- X - a list or tuple of equations

EXAMPLES:

```
sage: A.<x,y,z> = ProjectiveSpace(2, QQ)
sage: X = A.subscheme([x*z^2, y^2*z, x*y^2]); X
Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
  x*z^2
  y^2*z
  x*y^2
sage: X.defining_polynomials ()
(x*z^2, y^2*z, x*y^2)
sage: I = X.defining_ideal(); I
Ideal (x*z^2, y^2*z, x*y^2) of Multivariate Polynomial Ring in x, y, z over Rational Field
sage: I.groebner_basis()
[x*y^2, y^2*z,  x*z^2]
sage: X.dimension()
0
sage: X.base_ring()
Rational Field
sage: X.base_scheme()
Spectrum of Rational Field
sage: X.structure_morphism()
```

```
Scheme morphism:
  From: Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
  x*z^2
  y^2*z
  x*y^2
  To:   Spectrum of Rational Field
  Defn: Structure map
```

is_ProjectiveSpace(*x*)

Return True if x is a projective space, i.e., an ambient space $\mathbb{P}^n_R$, where R is a ring and $n \geq 0$ is an integer.

EXAMPLES:

```
sage: from sage.schemes.generic.projective_space import is_ProjectiveSpace
sage: is_ProjectiveSpace(ProjectiveSpace(5, names='x'))
True
sage: is_ProjectiveSpace(ProjectiveSpace(5, GF(9,'alpha'), names='x'))
True
sage: is_ProjectiveSpace(Spec(ZZ))
False
```

37.9 Algebraic schemes

An algebraic scheme must be defined by sets of equations in affine or projective spaces, perhaps by means of gluing relations.

class AlgebraicScheme(*A*)

An algebraic scheme presented as a subscheme in an ambient space.

ambient_space()

Return the ambient space of this algebraic scheme.

EXAMPLES:

```
sage: A.<x, y> = AffineSpace(2, GF(5))
sage: S = A.subscheme([])
sage: S.ambient_space()
Affine Space of dimension 2 over Finite Field of size 5

sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([x-y, x-z])
sage: S.ambient_space() is P
True
```

coordinate_ring()

Return the coordinate ring of this algebraic scheme. The result is cached.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([x-y, x-z])
sage: S.coordinate_ring()
Quotient of Multivariate Polynomial Ring in x, y, z over Integer Ring by the ideal (x - y, x
```

ngens()

Return the number of generators of the ambient space of this algebraic scheme.

EXAMPLES:

```
sage: A.<x, y> = AffineSpace(2, GF(5))
sage: S = A.subscheme([])
sage: S.ngens()
2

sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([x-y, x-z])
sage: P.ngens()
3
```

class AlgebraicScheme_quasi(*X, Y*)

The quasi-affine or quasi-projective scheme $X - Y$, where X and Y are both closed subschemes of a common ambient affine or projective space.

X()

Return the scheme X such that self is represented as $X - Y$.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([])
sage: T = P.subscheme([x-y])
sage: U = T.complement(S)
sage: U.X() is S
True
```

Y()

Return the scheme Y such that self is represented as $X - Y$.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([])
sage: T = P.subscheme([x-y])
sage: U = T.complement(S)
sage: U.Y() is T
True
```

rational_points(*F=None, bound=0*)

Return the set of rational points on this algebraic scheme over the field F.

EXAMPLES:

```
sage: A.<x, y> = AffineSpace(2, GF(7))
sage: S = A.subscheme([x^2-y])
sage: T = A.subscheme([x-y])
sage: U = T.complement(S)
sage: U.rational_points()
[(2, 4), (3, 2), (4, 2), (5, 4), (6, 1)]
sage: U.rational_points(GF(7^2, 'b'))
[(2, 4), (3, 2), (4, 2), (5, 4), (6, 1), (b, b + 4), (b + 1, 3*b + 5), (b + 2, 5*b + 1),
(b + 3, 6), (b + 4, 2*b + 6), (b + 5, 4*b + 1), (b + 6, 6*b + 5), (2*b, 4*b + 2),
(2*b + 1, b + 3), (2*b + 2, 5*b + 6), (2*b + 3, 2*b + 4), (2*b + 4, 6*b + 4),
(2*b + 5, 3*b + 6), (2*b + 6, 3), (3*b, 2*b + 1), (3*b + 1, b + 2), (3*b + 2, 5),
(3*b + 3, 6*b + 3), (3*b + 4, 5*b + 3), (3*b + 5, 4*b + 5), (3*b + 6, 3*b + 2),
(4*b, 2*b + 1), (4*b + 1, 3*b + 2), (4*b + 2, 4*b + 5), (4*b + 3, 5*b + 3),
(4*b + 4, 6*b + 3), (4*b + 5, 5), (4*b + 6, b + 2), (5*b, 4*b + 2), (5*b + 1, 3),
(5*b + 2, 3*b + 6), (5*b + 3, 6*b + 4), (5*b + 4, 2*b + 4), (5*b + 5, 5*b + 6),
(5*b + 6, b + 3), (6*b, b + 4), (6*b + 1, 6*b + 5), (6*b + 2, 4*b + 1), (6*b + 3, 2*b + 6),
(6*b + 4, 6), (6*b + 5, 5*b + 1), (6*b + 6, 3*b + 5)]
```

class AlgebraicScheme_subscheme(*A, polys*)

An algebraic scheme presented as a closed subscheme is defined by explicit polynomial equations. This is as opposed to a general scheme, which could, e.g., be the Neron model of some object, and for which we do not want to give explicit equations.

INPUT:

- `A` - ambient space (affine or projective n-space over a ring)
- `polys` - ideal or tuple of defining polynomials

base_extend(*R*)

Return the base change to the ring R of this scheme.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, GF(11))
sage: S = P.subscheme([x^2-y*z])
sage: S.base_extend(GF(11^2, 'b'))
Closed subscheme of Projective Space of dimension 2 over Finite Field in b of size 11^2 defi
  x^2 - y*z
sage: S.base_extend(ZZ)
...
ValueError: No natural map from the base ring (=Finite Field of size 11) to S (=Integer Ring
```

complement(*other=None*)

Return the scheme-theoretic complement other - self, where self and other are both closed algebraic subschemes of the same ambient space.

If other is unspecified, it is taken to be the ambient space of self.

EXAMPLES:

```
sage: A.<x, y, z> = AffineSpace(3, ZZ)
sage: X = A.subscheme([x+y-z])
sage: Y = A.subscheme([x-y+z])
sage: Y.complement(X)
Quasi-affine scheme X - Y, where:
  X: Closed subscheme of Affine Space of dimension 3 over Integer Ring defined by:
  x + y - z
  Y: Closed subscheme of Affine Space of dimension 3 over Integer Ring defined by:
  x - y + z
sage: Y.complement()
Quasi-affine scheme X - Y, where:
  X: Closed subscheme of Affine Space of dimension 3 over Integer Ring defined by:
  (no equations)
  Y: Closed subscheme of Affine Space of dimension 3 over Integer Ring defined by:
  x - y + z

sage: P.<x, y, z> = ProjectiveSpace(2, QQ)
sage: X = P.subscheme([x^2+y^2+z^2])
sage: Y = P.subscheme([x*y+y*z+z*x])
sage: Y.complement(X)
Quasi-projective scheme X - Y, where:
  X: Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
  x^2 + y^2 + z^2
  Y: Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
  x*y + x*z + y*z
sage: Y.complement(P)
Quasi-projective scheme X - Y, where:
  X: Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
  (no equations)
  Y: Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
  x*y + x*z + y*z
```

defining_ideal()

Return the ideal that defines this scheme as a subscheme of its ambient space.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([x^2-y*z, x^3+z^3])
sage: S.defining_ideal()
Ideal (x^2 - y*z, x^3 + z^3) of Multivariate Polynomial Ring in x, y, z over Integer Ring
```

defining_polynomials()

Return the polynomials that define this scheme as a subscheme of its ambient space.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([x^2-y*z, x^3+z^3])
sage: S.defining_polynomials()
(x^2 - y*z, x^3 + z^3)
```

intersection(*other*)

Return the scheme-theoretic intersection of self and other in their common ambient space.

EXAMPLES:

```
sage: A.<x, y> = AffineSpace(2, ZZ)
sage: X = A.subscheme([x^2-y])
sage: Y = A.subscheme([y])
sage: X.intersection(Y)
Closed subscheme of Affine Space of dimension 2 over Integer Ring defined by:
  x^2 - y
  y
```

irreducible_components()

Return the irreducible components of this algebraic scheme, as subschemes of the same ambient space.

OUTPUT: an immutable sequence of irreducible subschemes of the ambient space of this scheme

The components are cached.

EXAMPLES:

We define what is clearly a union of four hypersurfaces in $\P^4_{\mathbf{Q}}$ then find the irreducible components.

```
sage: PP.<x,y,z,w,v> = ProjectiveSpace(4,QQ)
sage: V = PP.subscheme( (x^2 - y^2 - z^2)*(w^5 -  2*v^2*z^3)* w * (v^3 - x^2*z) )
sage: V.irreducible_components()
[
Closed subscheme of Projective Space of dimension 4 over Rational Field defined by:
w,
Closed subscheme of Projective Space of dimension 4 over Rational Field defined by:
x^2 - y^2 - z^2,
Closed subscheme of Projective Space of dimension 4 over Rational Field defined by:
x^2*z - v^3,
Closed subscheme of Projective Space of dimension 4 over Rational Field defined by:
w^5 - 2*z^3*v^2
]
```

rational_points(*F=None, bound=0*)

EXAMPLES:

One can enumerate points up to a given bound on a projective scheme over the rationals.

```
sage: E = EllipticCurve('37a')
sage: E.rational_points(bound=8)
[(0 : 0 : 1),
```

```
 (1 : 0 : 1),
 (-1 : 0 : 1),
 (0 : -1 : 1),
 (1 : -1 : 1),
 (-1 : -1 : 1),
 (2 : 2 : 1),
 (2 : -3 : 1),
 (1/4 : -3/8 : 1),
 (1/4 : -5/8 : 1),
 (0 : 1 : 0)]
```

For a small finite field, the complete set of points can be enumerated.

```
sage: Etilde = E.base_extend(GF(3))
sage: Etilde.rational_points()
[(0 : 0 : 1), (1 : 0 : 1), (2 : 0 : 1), (0 : 2 : 1), (1 : 2 : 1), (2 : 2 : 1), (0 : 1 : 0)]
```

The class of hyperelliptic curves does not (yet) support desingularization of the places at infinity into two points.

```
sage: FF = FiniteField(7)
sage: P.<x> = PolynomialRing(FiniteField(7))
sage: C = HyperellipticCurve(x^8+x+1)
sage: C.rational_points()
[(2 : 0 : 1), (4 : 0 : 1), (0 : 1 : 1), (6 : 1 : 1), (0 : 6 : 1), (6 : 6 : 1), (0 : 1 : 0)]
```

TODO:

1. The above algorithms enumerate all projective points and test whether they lie on the scheme; Implement a more naive sieve at least for covers of the projective line.
2. Implement Stoll's model in weighted projective space to resolve singularities and find two points (1 : 1 : 0) and (-1 : 1 : 0) at infinity.

reduce()

Return the corresponding reduced algebraic space associated to this scheme.

EXAMPLES: First we construct the union of a doubled and triplled line in the affine plane over $\mathbf{Q}$.

```
sage: A.<x,y> = AffineSpace(2, QQ)
sage: X = A.subscheme([(x-1)^2*(x-y)^3]); X
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^5 - 3*x^4*y + 3*x^3*y^2 - x^2*y^3 - 2*x^4 + 6*x^3*y - 6*x^2*y^2 + 2*x*y^3 + x^3 - 3*x^2*
sage: X.dimension()
1
```

Then we compute the corresponding reduced scheme.

```
sage: Y = X.reduce(); Y
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^2 - x*y - x + y
```

Finally, we verify that the reduced scheme Y is the union of those two lines.

```
sage: L1 = A.subscheme([x-1]); L1
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x - 1
sage: L2 = A.subscheme([x-y]); L2
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x - y
sage: W = L1.union(L2); W             # taken in ambient space
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^2 - x*y - x + y
```

```
sage: Y == W
True
```

union(*other*)

Return the scheme-theoretic union of self and other in their common ambient space.

EXAMPLES: We construct the union of a line and a tripled-point on the line.

```
sage: A.<x,y> = AffineSpace(2, QQ)
sage: I = ideal([x,y])^3
sage: P = A.subscheme(I)
sage: L = A.subscheme([y-1])
sage: S = L.union(P); S
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
y^4 - y^3
x*y^3 - x*y^2
x^2*y^2 - x^2*y
x^3*y - x^3
sage: S.dimension()
1
sage: S.reduce()
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
y^2 - y
x*y - x
```

We can also use the notation "+" for the union:

```
sage: A.subscheme([x]) + A.subscheme([y^2 - (x^3+1)])
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
-x^4 + x*y^2 - x
```

Saving and loading:

```
sage: loads(S.dumps()) == S
True
```

class **AlgebraicScheme_subscheme_affine**(*A*, *polys*)

dimension()

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(2, QQ)
sage: A.subscheme([]).dimension()
2
sage: A.subscheme([x]).dimension()
1
sage: A.subscheme([x^5]).dimension()
1
sage: A.subscheme([x^2 + y^2 - 1]).dimension()
1
sage: A.subscheme([x*(x-1), y*(y-1)]).dimension()
0
```

Something less obvious

```
sage: A.<x,y,z,w> = AffineSpace(4, QQ)
sage: X = A.subscheme([x^2, x^2*y^2 + z^2, z^2 - w^2, 10*x^2 + w^2 - z^2])
sage: X
Closed subscheme of Affine Space of dimension 4 over Rational Field defined by:
x^2
x^2*y^2 + z^2
```

```
z^2 - w^2
10*x^2 - z^2 + w^2
sage: X.dimension()
1
```

projective_embedding(*i=None, X=None*)

Returns a morphism from this affine scheme into an ambient projective space of the same dimension.

INPUT:

- `i` - integer (default: dimension of self = last coordinate) determines which projective embedding to compute. The embedding is that which has a 1 in the i-th coordinate, numbered from 0.
- `X` - (default: None) projective scheme, i.e., codomain of morphism; this is constructed if it is not given.

EXAMPLES:

```
sage: A.<x, y, z> = AffineSpace(3, ZZ)
sage: S = A.subscheme([x*y-z])
sage: S.projective_embedding()
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 3 over Integer Ring defined by:
  x*y - z
  To:   Closed subscheme of Projective Space of dimension 3 over Integer Ring defined by:
  x0*x1 - x2*x3
  Defn: Defined on coordinates by sending (x, y, z) to
        (x : y : z : 1)
```

class **AlgebraicScheme_subscheme_projective**(*A, polys*)

affine_patch(*i*)

Return the i^{th} affine patch of this projective scheme. This is the intersection with this i^{th} affine patch of its ambient space.

INPUT:

- `i` - integer between 0 and dimension of self, inclusive.

OUTPUT: an affine scheme with fixed projective_embedding map.

EXAMPLES:

```
sage: PP = ProjectiveSpace(2, QQ, names='X,Y,Z')
sage: X,Y,Z = PP.gens()
sage: C = PP.subscheme(X^3*Y + Y^3*Z + Z^3*X)
sage: U = C.affine_patch(0)
sage: U
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
x0^3*x1 + x1^3 + x0
sage: U.projective_embedding()
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x0^3*x1 + x1^3 + x0
  To:   Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
  X^3*Y + Y^3*Z + X*Z^3
  Defn: Defined on coordinates by sending (x0, x1) to
        (1 : x0 : x1)
```

dimension()

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(2, QQ)
sage: A.subscheme([]).dimension()
2
sage: A.subscheme([x]).dimension()
1
sage: A.subscheme([x^5]).dimension()
1
sage: A.subscheme([x^2 + y^2 - 1]).dimension()
1
sage: A.subscheme([x*(x-1), y*(y-1)]).dimension()
0
```

Something less obvious

```
sage: A.<x,y,z,w> = AffineSpace(4, QQ)
sage: X = A.subscheme([x^2, x^2*y^2 + z^2, z^2 - w^2, 10*x^2 + w^2 - z^2])
sage: X
Closed subscheme of Affine Space of dimension 4 over Rational Field defined by:
x^2
x^2*y^2 + z^2
z^2 - w^2
10*x^2 - z^2 + w^2
sage: X.dimension()
1
```

is_AlgebraicScheme(x)

Return True if x is an algebraic scheme, i.e., a subscheme of an ambient space over a ring defined by polynomial equations.

EXAMPLES: Affine space is itself not an algebraic scheme, though the closed subscheme defined by no equations is.

```
sage: from sage.schemes.generic.algebraic_scheme import is_AlgebraicScheme
sage: is_AlgebraicScheme(AffineSpace(10, QQ))
False
sage: V = AffineSpace(10, QQ).subscheme([]); V
Closed subscheme of Affine Space of dimension 10 over
Rational Field defined by:
  (no equations)
sage: is_AlgebraicScheme(V)
True
```

We create a more complicated closed subscheme.

```
sage: A, x = AffineSpace(10, QQ).objgens()
sage: X = A.subscheme([sum(x)]); X
Closed subscheme of Affine Space of dimension 10 over Rational Field defined by:
x0 + x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9
sage: is_AlgebraicScheme(X)
True


sage: is_AlgebraicScheme(QQ)
False
sage: S = Spec(QQ)
sage: is_AlgebraicScheme(S)
False
```

37.10 Hypersurfaces in affine and projective space

AUTHORS:

- William Stein <wstein@gmail.com> (2005-12-08)
- David Kohel <kohel@maths.usyd.edu.au> (2005-12-08)
- Alex Ghitza <aghitza@alum.mit.edu> (2009-04-17)

class **AffineHypersurface** (*poly, ambient=None*)

The affine hypersurface defined by the given polynomial.

EXAMPLES:

```
sage: A.<x, y, z> = AffineSpace(ZZ, 3)
sage: AffineHypersurface(x*y-z^3, A)
Affine hypersurface defined by -z^3 + x*y in Affine Space of dimension 3 over Integer Ring
```

```
sage: A.<x, y, z> = QQ[]
sage: AffineHypersurface(x*y-z^3)
Affine hypersurface defined by -z^3 + x*y in Affine Space of dimension 3 over Rational Field
```

defining_polynomial ()

Return the polynomial equation that cuts out this affine hypersurface.

EXAMPLES:

```
sage: R.<x, y, z> = ZZ[]
sage: H = AffineHypersurface(x*z+y^2)
sage: H.defining_polynomial()
y^2 + x*z
```

class **ProjectiveHypersurface** (*poly, ambient=None*)

The projective hypersurface defined by the given polynomial.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(ZZ, 2)
sage: ProjectiveHypersurface(x-y, P)
Projective hypersurface defined by x - y in Projective Space of dimension 2 over Integer Ring
```

```
sage: R.<x, y, z> = QQ[]
sage: ProjectiveHypersurface(x-y)
Projective hypersurface defined by x - y in Projective Space of dimension 2 over Rational Field
```

defining_polynomial ()

Return the polynomial equation that cuts out this projective hypersurface.

EXAMPLES:

```
sage: R.<x, y, z> = ZZ[]
sage: H = ProjectiveHypersurface(x*z+y^2)
sage: H.defining_polynomial()
y^2 + x*z
```

is_Hypersurface (*self*)

Return True if self is a hypersurface, i.e. an object of the type ProjectiveHypersurface or AffineHypersurface.

EXAMPLES:

```
sage: from sage.schemes.generic.hypersurface import is_Hypersurface
sage: R.<x, y, z> = ZZ[]
sage: H = ProjectiveHypersurface(x*z+y^2)
sage: is_Hypersurface(H)
True

sage: H = AffineHypersurface(x*z+y^2)
sage: is_Hypersurface(H)
True

sage: H = ProjectiveSpace(QQ, 5)
sage: is_Hypersurface(H)
False
```

37.11 Set of homomorphisms between two schemes

SchemeHomset(*R, S, cat=None, check=True*)

class **SchemeHomsetModule_abelian_variety_coordinates_field**(*X, S, cat=None, check=True*)

base_extend(*R*)

class **SchemeHomset_affine_coordinates**(*X, S*)
Set of points on X defined over the base ring of X, and given by explicit tuples.

points(*B=0*)

class **SchemeHomset_coordinates**(*X, S*)
Set of points on X defined over the base ring of X, and given by explicit tuples.

value_ring()
Returns S for a homset X(T) where T = Spec(S).

class **SchemeHomset_generic**(*X, Y, cat=None, check=True, base=Integer Ring*)

has_coerce_map_from_impl(*S*)

natural_map()

class **SchemeHomset_projective_coordinates_field**(*X, S*)
Set of points on X defined over the base ring of X, and given by explicit tuples.

points(*B=0*)

class **SchemeHomset_projective_coordinates_ring**(*X, S*)
Set of points on X defined over the base ring of X, and given by explicit tuples.

points(*B=0*)

class **SchemeHomset_spec**(*X, Y, cat=None, check=True, base=Integer Ring*)

enum_affine_finite_field(*X*)

enum_affine_rational_field(*X, B*)

enum_projective_finite_field(*X*)

enum_projective_rational_field(*X, B*)

is_SchemeHomset(*H*)

37.12 Scheme morphism

AUTHORS:

- David Kohel, William Stein
- William Stein (2006-02-11): fixed bug where P(0,0,0) was allowed as a projective point.

class PyMorphism(*parent*)

category()

codomain()

domain()

is_endomorphism()

class SchemeMorphism(*parent*)

A scheme morphism

glue_along_domains(*other*)

Assuming that self and other are open immersions with the same domain, return scheme obtained by gluing along the images.

EXAMPLES: We construct a scheme isomorphic to the projective line over $\mathrm{Spec}(\mathbf{Q})$ by gluing two copies of $\mathbb{A}^1$ minus a point.

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: S.<xbar, ybar> = R.quotient(x*y - 1)
sage: Rx = PolynomialRing(QQ, 'x')
sage: i1 = Rx.hom([xbar])
sage: Ry = PolynomialRing(QQ, 'y')
sage: i2 = Ry.hom([ybar])
sage: Sch = Schemes()
sage: f1 = Sch(i1)
sage: f2 = Sch(i2)
```

Now f1 and f2 have the same domain, which is a $\mathbb{A}^1$ minus a point. We glue along the domain:

```
sage: P1 = f1.glue_along_domains(f2)
sage: P1
Scheme obtained by gluing X and Y along U, where
  X: Spectrum of Univariate Polynomial Ring in x over Rational Field
  Y: Spectrum of Univariate Polynomial Ring in y over Rational Field
  U: Spectrum of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the

sage: a, b = P1.gluing_maps()
sage: a
Affine Scheme morphism:
 From: Spectrum of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by t
  To:   Spectrum of Univariate Polynomial Ring in x over Rational Field
  Defn: Ring morphism:
          From: Univariate Polynomial Ring in x over Rational Field
          To:   Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the
          Defn: x |--> xbar
sage: b
Affine Scheme morphism:
  From: Spectrum of Quotient of Multivariate Polynomial Ring in x, y over Rational Field by
  To:   Spectrum of Univariate Polynomial Ring in y over Rational Field
  Defn: Ring morphism:
```

```
From: Univariate Polynomial Ring in y over Rational Field
To:   Quotient of Multivariate Polynomial Ring in x, y over Rational Field by the
Defn: y |--> ybar
```

class SchemeMorphism_abelian_variety_coordinates_field(*X, v, check=True*)

class SchemeMorphism_affine_coordinates(*X, v, check=True*)

A morphism determined by giving coordinates in a ring.

INPUT:

- X - a subscheme of an ambient affine space over a ring R.
- v - a list or tuple of coordinates in R

EXAMPLES:

```
sage: A = AffineSpace(2, QQ)
sage: A(1,2)
(1, 2)
```

class SchemeMorphism_coordinates(*parent*)

scheme()

class SchemeMorphism_id(*X*)

Return the identity morphism from X to itself.

EXAMPLES:

```
sage: X = Spec(ZZ)
sage: X.identity_morphism()
Scheme endomorphism of Spectrum of Integer Ring
  Defn: Identity map
```

class SchemeMorphism_on_points(*parent*)

A morphism of schemes determined by rational functions that define what the morphism does on points in the ambient space.

EXAMPLES: An example involving the affine plane:

```
sage: R.<x,y> = QQ[]
sage: A2 = AffineSpace(R)
sage: H = A2.Hom(A2)
sage: f = H([x-y, x*y])
sage: f([0,1])
(-1, 0)
```

An example involving the projective line:

```
sage: R.<x,y> = QQ[]
sage: P1 = ProjectiveSpace(R)
sage: H = P1.Hom(P1)
sage: f = H([x^2+y^2,x*y])
sage: f([0,1])
(1 : 0)
```

class SchemeMorphism_on_points_affine_space(*parent, polys, check=True*)

A morphism of schemes determined by rational functions that define what the morphism does on points in the ambient affine space.

defining_polynomials()

class SchemeMorphism_on_points_projective_space(*parent, polys, check=True*)
A morphism of schemes determined by rational functions that define what the morphism does on points in the ambient projective space.

defining_polynomials()

class SchemeMorphism_projective_coordinates_field(*X, v, check=True*)
A morphism determined by giving coordinates in a field.

INPUT:

- X - a subscheme of an ambient projective space over a field K
- v - a list or tuple of coordinates in K

EXAMPLES:

```
sage: P = ProjectiveSpace(3, RR)
sage: P(2,3,4,5)
(0.400000000000000 : 0.600000000000000 : 0.800000000000000 : 1.00000000000000)


sage: P = ProjectiveSpace(3, QQ)
sage: P(0,0,0,0)
...
ValueError: [0, 0, 0, 0] does not define a valid point since all entries are 0
```

class SchemeMorphism_projective_coordinates_ring(*X, v, check=True*)
A morphism determined by giving coordinates in a ring (how?).

class SchemeMorphism_spec(*parent, phi, check=True*)
A morphism of spectrums of rings

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: phi = R.hom([QQ(7)]); phi
Ring morphism:
  From: Univariate Polynomial Ring in x over Rational Field
  To:   Rational Field
  Defn: x |--> 7


sage: X = Spec(QQ); Y = Spec(R)
sage: f = X.hom(phi); f
Affine Scheme morphism:
  From: Spectrum of Rational Field
  To:   Spectrum of Univariate Polynomial Ring in x over Rational Field
  Defn: Ring morphism:
          From: Univariate Polynomial Ring in x over Rational Field
          To:   Rational Field
          Defn: x |--> 7


sage: f.ring_homomorphism()
Ring morphism:
  From: Univariate Polynomial Ring in x over Rational Field
  To:   Rational Field
  Defn: x |--> 7
```

ring_homomorphism()

class SchemeMorphism_structure_map(*parent*)

is_SchemeMorphism(*f*)

37.13 Divisors on schemes

AUTHORS:

- William Stein
- David Kohel
- David Joyner

EXAMPLES:

```
sage: x,y,z = ProjectiveSpace(2, GF(5), names='x,y,z').gens()
sage: C = Curve(y^2*z^7 - x^9 - x*z^8)
sage: pts = C.rational_points(); pts
[(0 : 0 : 1), (0 : 1 : 0), (2 : 2 : 1), (2 : 3 : 1), (3 : 1 : 1), (3 : 4 : 1)]
sage: D1 = C.divisor(pts[0])*3
sage: D2 = C.divisor(pts[1])
sage: D3 = 10*C.divisor(pts[5])
sage: D1.parent() is D2.parent()
True
sage: D = D1 - D2 + D3; D
-(x, z) + 3*(x, y) + 10*(x + 2*z, y + z)
sage: D[1][0]
3
sage: D[1][1]
Ideal (x, y) of Multivariate Polynomial Ring in x, y, z over Finite Field of size 5
sage: C.divisor([(3, pts[0]), (-1, pts[1]), (10,pts[5])])
-(x, z) + 3*(x, y) + 10*(x + 2*z, y + z)
```

CurvePointToIdeal(*C, P*)

class Divisor_curve(*v, check=True, reduce=True, parent=None*)

For any curve C, use `C.divisor(v)` to construct a divisor on C. Here v can be either

- a rational point on C
- a list of rational points
- a list of 2-tuples (c, P), where c is an integer and P is a rational point.

TODO: Divisors shouldn't be restricted to rational points. The problem is that the divisor group is the formal sum of the group of points on the curve, and there's no implemented notion of point on E/K that has coordinates in L. This is what should be implemented, by adding an appropriate class to `schemes/generic/morphism.py`.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])
sage: P = E(0,0)
sage: 10*P
(161/16 : -2065/64 : 1)
sage: D = E.divisor(P)
sage: D
```

```
(x, y)
sage: 10*D
10*(x, y)
sage: E.divisor([P, P])
2*(x, y)
sage: E.divisor([(3,P), (-4,5*P)])
3*(x, y) - 4*(x - 1/4*z, y + 5/8*z)
```

coeff(*P*)

Return the coefficient of a given point P in this divisor.

EXAMPLES:

```
sage: x,y = AffineSpace(2, GF(5), names='xy').gens()
sage: C = Curve(y^2 - x^9 - x)
sage: pts = C.rational_points(); pts
[(0, 0), (2, 2), (2, 3), (3, 1), (3, 4)]
sage: D = C.divisor(pts[0])
sage: D.coeff(pts[0])
1
sage: D = C.divisor([(3,pts[0]), (-1,pts[1])]); D
3*(x, y) - (x - 2, y - 2)
sage: D.coeff(pts[0])
3
sage: D.coeff(pts[1])
-1
```

support()

Return the support of this divisor, which is the set of points that occur in this divisor with nonzero coefficients.

EXAMPLES:

```
sage: x,y = AffineSpace(2, GF(5), names='xy').gens()
sage: C = Curve(y^2 - x^9 - x)
sage: pts = C.rational_points(); pts
[(0, 0), (2, 2), (2, 3), (3, 1), (3, 4)]
sage: D = C.divisor([(3,pts[0]), (-1, pts[1])]); D
3*(x, y) - (x - 2, y - 2)
sage: D.support()
[(0, 0), (2, 2)]
```

class Divisor_generic(*v, check=True, reduce=True, parent=None*)

scheme()

Return the scheme that this divisor is on.

EXAMPLES:

```
sage: A.<x, y> = AffineSpace(2, GF(5))
sage: C = Curve(y^2 - x^9 - x)
sage: pts = C.rational_points(); pts
[(0, 0), (2, 2), (2, 3), (3, 1), (3, 4)]
sage: D = C.divisor(pts[0])*3 - C.divisor(pts[1]); D
3*(x, y) - (x - 2, y - 2)
sage: D.scheme()
Affine Curve over Finite Field of size 5 defined by -x^9 + y^2 - x
```

is_Divisor(*Div*)

is_DivisorGroup(*Div*)

CHAPTER

THIRTYEIGHT

ELLIPTIC AND PLANE CURVES

38.1 Plane curve constructors

AUTHORS:

- William Stein (2005-11-13)
- David Kohel (2006-01)

Curve(*F*)

Return the plane or space curve defined by F, where F can be either a multivariate polynomial, a list or tuple of polynomials, or an algebraic scheme.

If F is in two variables the curve is affine, and if it is homogenous in 3 variables, then the curve is projective.

EXAMPLE: A projective plane curve

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: C = Curve(x^3 + y^3 + z^3); C
Projective Curve over Rational Field defined by x^3 + y^3 + z^3
sage: C.genus()
1
```

EXAMPLE: Affine plane curves

```
sage: x,y = GF(7)['x,y'].gens()
sage: C = Curve(y^2 + x^3 + x^10); C
Affine Curve over Finite Field of size 7 defined by x^10 + x^3 + y^2
sage: C.genus()
0
sage: x, y = QQ['x,y'].gens()
sage: Curve(x^3 + y^3 + 1)
Affine Curve over Rational Field defined by x^3 + y^3 + 1
```

EXAMPLE: A projective space curve

```
sage: x,y,z,w = QQ['x,y,z,w'].gens()
sage: C = Curve([x^3 + y^3 - z^3 - w^3, x^5 - y*z^4]); C
Projective Space Curve over Rational Field defined by x^3 + y^3 - z^3 - w^3
sage: C.genus()
13
```

EXAMPLE: An affine space curve

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: C = Curve([y^2 + x^3 + x^10 + z^7,  x^2 + y^2]); C
Affine Space Curve over Rational Field defined by x^10 + z^7 + x^3 + y^2
sage: C.genus()
47
```

EXAMPLE: We can also make non-reduced non-irreducible curves.

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: Curve((x-y)*(x+y))
Projective Curve over Rational Field defined by x^2 - y^2
sage: Curve((x-y)^2*(x+y)^2)
Projective Curve over Rational Field defined by x^4 - 2*x^2*y^2 + y^4
```

EXAMPLE: A union of curves is a curve.

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: C = Curve(x^3 + y^3 + z^3)
sage: D = Curve(x^4 + y^4 + z^4)
sage: C.union(D)
Projective Curve over Rational Field defined by
x^7 + x^4*y^3 + x^3*y^4 + y^7 + x^4*z^3 + y^4*z^3 + x^3*z^4 + y^3*z^4 + z^7
```

The intersection is not a curve, though it is a scheme.

```
sage: X = C.intersection(D); X
Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
 x^3 + y^3 + z^3
 x^4 + y^4 + z^4
```

Note that the intersection has dimension 0.

```
sage: X.dimension()
0
sage: I = X.defining_ideal(); I
Ideal (x^3 + y^3 + z^3, x^4 + y^4 + z^4) of Multivariate Polynomial Ring in x, y, z over Rationa
```

EXAMPLE: In three variables, the defining equation must be homogeneous.

If the parent polynomial ring is in three variables, then the defining ideal must be homogeneous.

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: Curve(x^2+y^2)
Projective Curve over Rational Field defined by x^2 + y^2
sage: Curve(x^2+y^2+z)
...
TypeError: The polynomial(s) x^2 + y^2 + z must be homogeneous
```

The defining polynomial must always be nonzero:

```
sage: P1.<x,y> = ProjectiveSpace(1,GF(5))
sage: Curve(0*x)
...
ValueError: defining polynomial of curve must be nonzero
```

38.2 Affine plane curves over a general ring

AUTHORS:

- William Stein (2005-11-13)
- David Joyner (2005-11-13)
- David Kohel (2006-01)

class AffineCurve_finite_field(*A, f*)

rational_points(*algorithm='enum'*)

Return sorted list of all rational points on this curve.

Use *very* naive point enumeration to find all rational points on this curve over a finite field.

EXAMPLE:

```
sage: A, (x,y) = AffineSpace(2,GF(9,'a')).objgens()
sage: C = Curve(x^2 + y^2 - 1)
sage: C
Affine Curve over Finite Field in a of size 3^2 defined by x0^2 + x1^2 - 1
sage: C.rational_points()
[(0, 1), (0, 2), (1, 0), (2, 0), (a + 1, a + 1), (a + 1, 2*a + 2), (2*a + 2, a + 1), (2*a +
```

class AffineCurve_generic(*A, f*)

divisor_of_function(*r*)

Return the divisor of a function on a curve.

INPUT: r is a rational function on X

OUTPUT:

•`list` - The divisor of r represented as a list of coefficients and points. (TODO: This will change to a more structural output in the future.)

EXAMPLES:

```
sage: F = GF(5)
sage: P2 = AffineSpace(2, F, names = 'xy')
sage: R = P2.coordinate_ring()
sage: x, y = R.gens()
sage: f = y^2 - x^9 - x
sage: C = Curve(f)
sage: K = FractionField(R)
sage: r = 1/x
sage: C.divisor_of_function(r)      # todo: not implemented (broken)
      [[-1, (0, 0, 1)]]
sage: r = 1/x^3
sage: C.divisor_of_function(r)      # todo: not implemented (broken)
      [[-3, (0, 0, 1)]]
```

local_coordinates(*pt, n*)

Return local coordinates to precision n at the given point.

Behaviour is flakey - some choices of n are worst that others.

INPUT:

•`pt` - an F-rational point on X which is not a point of ramification for the projection (x,y) - x.

•`n` - the number of terms desired

OUTPUT: x = x0 + t y = y0 + power series in t

EXAMPLES:

```
sage: F = GF(5)
sage: pt = (2,3)
sage: R = PolynomialRing(F,2, names = ['x','y'])
sage: x,y = R.gens()
sage: f = y^2-x^9-x
sage: C = Curve(f)
sage: C.local_coordinates(pt, 9)
[t + 2, -2*t^12 - 2*t^11 + 2*t^9 + t^8 - 2*t^7 - 2*t^6 - 2*t^4 + t^3 - 2*t^2 - 2]
```

plot(**args, **kwds*)

Plot the real points on this affine plane curve.

INPUT:

- `self` - an affine plane curve
- `*args` - optional tuples (variable, minimum, maximum) for plotting dimensions
- `**kwds` - optional keyword arguments passed on to `implicit_plot`

EXAMPLES:

A cuspidal curve:

```
sage: R.<x, y> = QQ[]
sage: C = Curve(x^3 - y^2)
sage: C.plot()
```

A 5-nodal curve of degree 11. This example also illustrates some of the optional arguments:

```
sage: R.<x, y> = ZZ[]
sage: C = Curve(32*x^2 - 2097152*y^11 + 1441792*y^9 - 360448*y^7 + 39424*y^5 - 1760*y^3 + 22
sage: C.plot((x, -1, 1), (y, -1, 1), plot_points=400)
```

A line over $\mathbf{RR}$:

```
sage: R.<x, y> = RR[]
sage: C = Curve(R(y - sqrt(2)*x))
sage: C.plot()
```

class **AffineCurve_prime_finite_field**(*A, f*)

rational_points(*algorithm='enum'*)

Return sorted list of all rational points on this curve.

INPUT:

- `algorithm` - string:
- `'enum'` - straightforward enumeration
- `'bn'` - via Singular's Brill-Noether package.
- `'all'` - use all implemented algorithms and verify that they give the same answer, then return it

Note: The Brill-Noether package does not always work. When it fails a RuntimeError exception is raised.

EXAMPLE:

```
sage: x, y = (GF(5)['x,y']).gens()
sage: f = y^2 - x^9 - x
sage: C = Curve(f); C
Affine Curve over Finite Field of size 5 defined by -x^9 + y^2 - x
sage: C.rational_points(algorithm='bn')
[(0, 0), (2, 2), (2, 3), (3, 1), (3, 4)]
```

```
sage: C = Curve(x - y + 1)
sage: C.rational_points()
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
```

The following seems to run fine on Linux but *crashes* on OS X intel:

```
sage: x, y = (GF(17)['x,y']).gens()
sage: C = Curve(x^2 + y^5 + x*y - 19)
sage: v = C.rational_points(algorithm='bn')       # not tested
sage: w = C.rational_points(algorithm='enum')     # not tested
sage: len(v)                                      # not tested
20
sage: v == w                                      # not tested
True
```

riemann_roch_basis(*D*)

Interfaces with Singular's BrillNoether command.

INPUT:

- `self` - a plane curve defined by a polynomial eqn f(x,y) = 0 over a prime finite field F = GF(p) in 2 variables x,y representing a curve X: f(x,y) = 0 having n F-rational points (see the Sage function places_on_curve)
- `D` - an n-tuple of integers $(d1, ..., dn)$ representing the divisor $Div = d1 * P1 + ... + dn * Pn$, where $X(F) = \{P1, ..., Pn\}$. *The ordering is that dictated by places_on_curve.*

OUTPUT: basis of L(Div)

EXAMPLE:

```
sage: R = PolynomialRing(GF(5),2,names = ["x","y"])
sage: x, y = R.gens()
sage: f = y^2 - x^9 - x
sage: C = Curve(f)
sage: D = [6,0,0,0,0,0]
sage: C.riemann_roch_basis(D)
[1, (y^2*z^4 - x*z^5)/x^6, (y^2*z^5 - x*z^6)/x^7, (y^2*z^6 - x*z^7)/x^8]
```

class **AffineSpaceCurve_generic**(*A, X*)

38.3 Projective plane curves over a general ring

AUTHORS:

- William Stein (2005-11-13)
- David Joyner (2005-11-13)
- David Kohel (2006-01)

Hasse_bounds(*q, genus=1*)

Return the Hasse-Weil bounds for the cardinality of a nonsingular curve defined over $\mathbf{F}_q$ of given `genus`.

INPUT:

- `q` (int) – a prime power
- `genus` (int, default 1) – a non-negative integer,

OUTPUT:

(tuple) The Hasse bounds (lb,ub) for the cardinality of a curve of genus `genus` defined over $\mathbf{F}_q$.

EXAMPLES:

```
sage: Hasse_bounds(2)
(1, 5)
sage: Hasse_bounds(next_prime(10^30))
(999999999999998000000000000058, 1000000000000002000000000000058)
```

class `ProjectiveCurve_finite_field`(*A, f*)

`rational_points`(*algorithm='enum', sort=True*)

Return the rational points on this curve computed via enumeration.

Note: This is a slow Python-level implementation.

class `ProjectiveCurve_generic`(*A, f*)

`arithmetic_genus`()

Return the arithmetic genus of this curve.

This is the arithmetic genus $g_a(C)$ as defined in Hartshorne. If the curve has degree d then this is simply $(d-1)(d-2)/2$. It need *not* equal the geometric genus (the genus of the normalization of the curve).

EXAMPLE:

```
sage: x,y,z = PolynomialRing(GF(5), 3, 'xyz').gens()
sage: C = Curve(y^2*z^7 - x^9 - x*z^8); C
Projective Curve over Finite Field of size 5 defined by -x^9 + y^2*z^7 - x*z^8
sage: C.arithmetic_genus()
28
sage: C.genus()
4
```

`divisor_of_function`(*r*)

Return the divisor of a function on a curve.

INPUT: r is a rational function on X

OUTPUT:

- `list` - The divisor of r represented as a list of coefficients and points. (TODO: This will change to a more structural output in the future.)

EXAMPLES:

```
sage: FF = FiniteField(5)
sage: P2 = ProjectiveSpace(2, FF, names = ['x','y','z'])
sage: R = P2.coordinate_ring()
sage: x, y, z = R.gens()
sage: f = y^2*z^7 - x^9 - x*z^8
sage: C = Curve(f)
sage: K = FractionField(R)
sage: r = 1/x
sage: C.divisor_of_function(r)      # todo: not implemented  !!!!
[[-1, (0, 0, 1)]]
sage: r = 1/x^3
sage: C.divisor_of_function(r)      # todo: not implemented  !!!!
[[-3, (0, 0, 1)]]
```

`local_coordinates`(*pt, n*)

Return local coordinates to precision n at the given point.

Behaviour is flakey - some choices of n are worst that others.

INPUT:

- `pt` - an F-rational point on X which is not a point of ramification for the projection (x,y) - x.
- `n` - the number of terms desired

OUTPUT: x = x0 + t y = y0 + power series in t

EXAMPLES:

```
sage: FF = FiniteField(5)
sage: P2 = ProjectiveSpace(2, FF, names = ['x','y','z'])
sage: x, y, z = P2.coordinate_ring().gens()
sage: C = Curve(y^2*z^7-x^9-x*z^8)
sage: pt = C([2,3,1])
sage: C.local_coordinates(pt,9)      # todo: not implemented  !!!!
      [2 + t, 3 + 3*t^2 + t^3 + 3*t^4 + 3*t^6 + 3*t^7 + t^8 + 2*t^9 + 3*t^11 + 3*t^12]
```

plot(**args, **kwds*)

Plot the real points of an affine patch of this projective plane curve.

INPUT:

- `self` - an affine plane curve
- `patch` - (optional) the affine patch to be plotted; if not specified, the patch corresponding to the last projective coordinate being nonzero
- `*args` - optional tuples (variable, minimum, maximum) for plotting dimensions
- `**kwds` - optional keyword arguments passed on to `implicit_plot`

EXAMPLES:

A cuspidal curve:

```
sage: R.<x, y, z> = QQ[]
sage: C = Curve(x^3 - y^2*z)
sage: C.plot()
```

The other affine patches of the same curve:

```
sage: C.plot(patch=0)
sage: C.plot(patch=1)
```

An elliptic curve:

```
sage: E = EllipticCurve('101a')
sage: C = Curve(E)
sage: C.plot()
sage: C.plot(patch=0)
sage: C.plot(patch=1)
```

A hyperelliptic curve:

```
sage: P.<x> = QQ[]
sage: f = 4*x^5 - 30*x^3 + 45*x - 22
sage: C = HyperellipticCurve(f)
sage: C.plot()
sage: C.plot(patch=0)
sage: C.plot(patch=1)
```

class **ProjectiveCurve_prime_finite_field**(*A, f*)

rational_points(*algorithm='enum', sort=True*)

INPUT:

- `algorithm` - string:

- 'enum' - straightforward enumeration
- 'bn' - via Singular's brnoeth package.

EXAMPLE:

```
sage: x, y, z = PolynomialRing(GF(5), 3, 'xyz').gens()
sage: f = y^2*z^7 - x^9 - x*z^8
sage: C = Curve(f); C
Projective Curve over Finite Field of size 5 defined by -x^9 + y^2*z^7 - x*z^8
sage: C.rational_points()
[(0 : 0 : 1), (0 : 1 : 0), (2 : 2 : 1), (2 : 3 : 1), (3 : 1 : 1), (3 : 4 : 1)]
sage: C = Curve(x - y + z)
sage: C.rational_points()
[(0 : 1 : 1), (1 : 1 : 0), (1 : 2 : 1), (2 : 3 : 1), (3 : 4 : 1), (4 : 0 : 1)]
```

Note: The Brill-Noether package does not always work (i.e., the 'bn' algorithm. When it fails a RuntimeError exception is raised.

riemann_roch_basis(*D*)

Return a basis for the Riemann-Roch space corresponding to D.

Warning: This function calls a Singular function that appears to be very buggy and should not be trusted.

This uses Singular's Brill-Noether implementation.

INPUT:

- sort - bool (default: True), if True return the point list sorted. If False, returns the pointes in the order computed by Singular.

EXAMPLE:

```
sage: R.<x,y,z> = GF(2)[]
sage: f = x^3*y + y^3*z + x*z^3
sage: C = Curve(f); pts = C.rational_points()
sage: D = C.divisor([ (4, pts[0]), (0,pts[1]), (4, pts[2]) ])
sage: C.riemann_roch_basis(D)
[x/y, 1, z/y, z^2/y^2, z/x, z^2/(x*y)]
```

The following example illustrates that the Riemann-Roch space function in Singular doesn't *not* work correctly.

```
sage: R.<x,y,z> = GF(5)[]
sage: f = x^7 + y^7 + z^7
sage: C = Curve(f); pts = C.rational_points()
sage: D = C.divisor([ (3, pts[0]), (-1,pts[1]), (10, pts[5]) ])
sage: C.riemann_roch_basis(D)     # output is random (!!!!)
[x/(y + x), (z + y)/(y + x)]
```

The answer has dimension 2 (confirmed via Magma). But it varies between 1 and quite large with Singular.

class ProjectiveSpaceCurve_generic(*A, X*)

38.4 Elliptic curve constructor

AUTHORS:

- William Stein (2005): Initial version
- John Cremona (2008-01): EllipticCurve(j) fixed for all cases

EllipticCurve (*x=None, y=None, j=None*)

There are several ways to construct an elliptic curve:

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$

- EllipticCurve([a1,a2,a3,a4,a6]): Elliptic curve with given a-invariants. The invariants are coerced into the parent of the first element. If all are integers, they are coerced into the rational numbers.
- EllipticCurve([a4,a6]): Same as above, but a1=a2=a3=0.
- EllipticCurve(label): Returns the elliptic curve over Q from the Cremona database with the given label. The label is a string, such as "11a" or "37b2". The letters in the label *must* be lower case (Cremona's new labeling).
- EllipticCurve(R, [a1,a2,a3,a4,a6]): Create the elliptic curve over R with given a-invariants. Here R can be an arbitrary ring. Note that addition need not be defined.
- EllipticCurve(j): Return an elliptic curve with j-invariant j. Warning: this is deprecated. Use `EllipticCurve_from_j(j)` or `EllipticCurve(j=j)` instead.

EXAMPLES: We illustrate creating elliptic curves.

```
sage: EllipticCurve([0,0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

We create a curve from a Cremona label:

```
sage: EllipticCurve('37b2')
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 1873*x - 31833 over Rational Field
sage: EllipticCurve('5077a')
Elliptic Curve defined by y^2 + y = x^3 - 7*x + 6 over Rational Field
sage: EllipticCurve('389a')
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
```

We create curves over a finite field as follows:

```
sage: EllipticCurve([GF(5)(0),0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 + 4*x over Finite Field of size 5
sage: EllipticCurve(GF(5), [0, 0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 + 4*x over Finite Field of size 5
```

Elliptic curves over $\mathbf{Z}/N\mathbf{Z}$ with N prime are of type "elliptic curve over a finite field":

```
sage: F = Zmod(101)
sage: EllipticCurve(F, [2, 3])
Elliptic Curve defined by y^2 = x^3 + 2*x + 3 over Ring of integers modulo 101
sage: E = EllipticCurve([F(2), F(3)])
sage: type(E)
<class 'sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field'>
```

In contrast, elliptic curves over $\mathbf{Z}/N\mathbf{Z}$ with N composite are of type "generic elliptic curve":

```
sage: F = Zmod(95)
sage: EllipticCurve(F, [2, 3])
Elliptic Curve defined by y^2 = x^3 + 2*x + 3 over Ring of integers modulo 95
sage: E = EllipticCurve([F(2), F(3)])
sage: type(E)
<class 'sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic'>
```

The following is a curve over the complex numbers:

```
sage: E = EllipticCurve(CC, [0,0,1,-1,0])
sage: E
Elliptic Curve defined by y^2 + 1.00000000000000*y = x^3 + (-1.00000000000000)*x over Complex Fi
sage: E.j_invariant()
2988.97297297297
```

We can also create elliptic curves by giving the Weierstrass equation:

```
sage: x, y = var('x,y')
sage: EllipticCurve(y^2 + y ==  x^3 + x - 9)
Elliptic Curve defined by y^2 + y = x^3 + x - 9 over Rational Field

sage: R.<x,y> = GF(5)[]
sage: EllipticCurve(x^3 + x^2 + 2 - y^2 - y*x)
Elliptic Curve defined by y^2 + x*y  = x^3 + x^2 + 2 over Finite Field of size 5
```

We can explicitly specify the j-invariant:

```
sage: E = EllipticCurve(j=1728); E; E.j_invariant(); E.label()
Elliptic Curve defined by y^2 = x^3 - x over Rational Field
1728
'32a2'

sage: E = EllipticCurve(j=GF(5)(2)); E; E.j_invariant()
Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 5
2
```

TESTS:

```
sage: R = ZZ['u', 'v']
sage: EllipticCurve(R, [1,1])
Elliptic Curve defined by y^2 = x^3 + x + 1 over Multivariate Polynomial Ring in u, v
over Integer Ring
```

We create a curve and a point over QQbar:

```
sage: E = EllipticCurve(QQbar,[0,1])
sage: E(0)
(0 : 1 : 0)
```

EllipticCurve_from_c4c6(*c4, c6*)

Return an elliptic curve with given c_4 and c_6 invariants.

EXAMPLES:

```
sage: E = EllipticCurve_from_c4c6(17, -2005)
sage: E
Elliptic Curve defined by y^2  = x^3 - 17/48*x + 2005/864 over Rational Field
sage: E.c_invariants()
(17, -2005)
```

EllipticCurve_from_cubic(*F, P*)

Given a nonsingular homogenous cubic polynomial F over $\mathbf{Q}$ in three variables x, y, z and a projective solution P=[a,b,c] to F(P)=0, find the minimal Weierstrass equation of the elliptic curve over $\mathbf{Q}$ that is isomorphic to the curve defined by $F = 0$.

Note: USES MAGMA - This function will not work on computers that do not have magma installed. (HELP WANTED - somebody implement this independent of MAGMA.)

EXAMPLES: First we find that the Fermat cubic is isomorphic to the curve with Cremona label 27a1:

```
sage: E = EllipticCurve_from_cubic('x^3 + y^3 + z^3', [1,-1,0])  # optional - magma
sage: E         # optional - magma
Elliptic Curve defined by y^2 + y = x^3 - 7 over Rational Field
sage: E.cremona_label()     # optional - magma
'27a1'
```

Next we find the minimal model and conductor of the Jacobian of the Selmer curve.

```
sage: E = EllipticCurve_from_cubic('x^3 + y^3 + 60*z^3', [1,-1,0])   # optional - magma
sage: E                # optional - magma
Elliptic Curve defined by y^2  = x^3 - 24300 over Rational Field
sage: E.conductor()    # optional - magma
24300
```

EllipticCurve_from_j(*j*)

Return an elliptic curve with given j-invariant.

EXAMPLES:

```
sage: E = EllipticCurve_from_j(0); E; E.j_invariant(); E.label()
Elliptic Curve defined by y^2 + y = x^3 over Rational Field
0
'27a3'

sage: E = EllipticCurve_from_j(1728); E; E.j_invariant(); E.label()
Elliptic Curve defined by y^2 = x^3 - x over Rational Field
1728
'32a2'

sage: E = EllipticCurve_from_j(1); E; E.j_invariant()
Elliptic Curve defined by y^2 + x*y = x^3 + 36*x + 3455 over Rational Field
1
```

EllipticCurves_with_good_reduction_outside_S(*S=, [], proof=None, verbose=False*)

Returns a sorted list of all elliptic curves defined over Q with good reduction outside the set S of primes.

INPUT:

- `S` - list of primes (default: empty list).
- `proof` - True/False (default True): the MW basis for auxiliary curves will be computed with this proof flag.
- `verbose` - True/False (default False): if True, some details of the computation will be output.

Note: Proof flag: The algorithm used requires determining all S-integral points on several auxiliary curves, which in turn requires the computation of their generators. This is not always possible (even in theory) using current knowledge.

The value of this flag is passed to the function which computes generators of various auxiliary elliptic curves, in order to find their S-integral points. Set to False if the default (True) causes warning messages, but note that you can then not rely on the set of curves returned being complete.

EXAMPLES:

```
sage: EllipticCurves_with_good_reduction_outside_S([])
[]
sage: elist = EllipticCurves_with_good_reduction_outside_S([2])
sage: elist
[Elliptic Curve defined by y^2 = x^3 + 4*x over Rational Field,
Elliptic Curve defined by y^2 = x^3 - x over Rational Field,
```

```
...
Elliptic Curve defined by y^2 = x^3 - x^2 - 13*x + 21 over Rational Field]
sage: len(elist)
24
sage: ', '.join([e.label() for e in elist])
'32a1, 32a2, 32a3, 32a4, 64a1, 64a2, 64a3, 64a4, 128a1, 128a2, 128b1, 128b2, 128c1, 128c2, 128d1
```

 # Without the "Proof=False", this example gives two warnings: sage: elist = EllipticCurves_with_good_reduction_outside_S([11],proof=False) sage: len(elist) 12 sage: ', '.join([e.label() for e in elist]) '11a1, 11a2, 11a3, 121a1, 121a2, 121b1, 121b2, 121c1, 121c2, 121d1, 121d2, 121d3'

sage: elist = EllipticCurves_with_good_reduction_outside_S([2,3]) # long time (~35s) sage: len(elist) # long time 752 sage: max([e.conductor() for e in elist]) # long time 62208 sage: [N.factor() for N in Set([e.conductor() for e in elist])] # long time [2^7, 2^8, 2^3 * 3^4, 2^2 * 3^3, 2^8 * 3^4, 2^4 * 3^4, 2^3 * 3, 2^7 * 3, 2^3 * 3^5, 3^3, 2^8 * 3, 2^5 * 3^4, 2^4 * 3, 2 * 3^4, 2^2 * 3^2, 2^6 * 3^4, 2^6, 2^7 * 3^2, 2^4 * 3^5, 2^4 * 3^3, 2 * 3^3, 2^6 * 3^3, 2^6 * 3, 2^5, 2^2 * 3^4, 2^3 * 3^2, 2^5 * 3, 2^7 * 3^4, 2^2 * 3^5, 2^8 * 3^2, 2^5 * 3^2, 2^7 * 3^5, 2^8 * 3^5, 2^3 * 3^3, 2^8 * 3^3, 2^5 * 3^5, 2^4 * 3^2, 2 * 3^5, 2^5 * 3^3, 2^6 * 3^5, 2^7 * 3^3, 3^5, 2^6 * 3^2]

38.5 Elliptic curves over a general ring.

Elliptic curves are always represented by 'Weierstrass Models' with five coefficients $[a_1, a_2, a_3, a_4, a_6]$ in standard notation. In Magma, 'Weierstrass Model' means a model with a1=a2=a3=0, which is called 'Short Weierstrass Model' in Sage; these only exist in characteristics other than 2 and 3.

EXAMPLES:

We construct an elliptic curve over an elaborate base ring:

```
sage: p = 97; a=1; b=3
sage: R, u = PolynomialRing(GF(p), 'u').objgen()
sage: S, v = PolynomialRing(R, 'v').objgen()
sage: T = S.fraction_field()
sage: E = EllipticCurve(T, [a, b]); E
Elliptic Curve defined by y^2  = x^3 + x + 3 over Fraction Field of Univariate Polynomial Ring in v o
sage: latex(E)
y^2  = x^3 + x + 3
```

AUTHORS:

- William Stein (2005): Initial version
- Robert Bradshaw et al....
- John Cremona (2008-01): isomorphisms, automorphisms and twists in all characteristics

class `EllipticCurve_generic` (*ainvs, extra=None*)

Elliptic curve over a generic base ring.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3/4,7,19]); E
Elliptic Curve defined by y^2 + x*y + 3/4*y = x^3 + 2*x^2 + 7*x + 19 over Rational Field
sage: loads(E.dumps()) == E
True
sage: E = EllipticCurve([1,3])
```

```
sage: P = E([-1,1,1])
sage: -5*P
(179051/80089 : -91814227/22665187 : 1)
```

a1()

Returns the a_1 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,6])
sage: E.a1()
1
```

a2()

Returns the a_2 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,6])
sage: E.a2()
2
```

a3()

Returns the a_3 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,6])
sage: E.a3()
3
```

a4()

Returns the a_4 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,6])
sage: E.a4()
4
```

a6()

Returns the a_6 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,6])
sage: E.a6()
6
```

a_invariants()

The a-invariants of this elliptic curve, as a list.

OUTPUT:

(list) - a (new) list of the 5 a-invariants.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.a_invariants()
[1, 2, 3, 4, 5]
sage: E = EllipticCurve([0,1])
sage: E
Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
sage: E.a_invariants()
[0, 0, 0, 0, 1]
sage: E = EllipticCurve([GF(7)(3),5])
```

```
sage: E.a_invariants()
[0, 0, 0, 3, 5]
```

We check that a new list is returned:

```
sage: E = EllipticCurve([1,0,0,0,1])
sage: E.a_invariants()[0] = 100000000
sage: E
Elliptic Curve defined by y^2 + x*y = x^3 + 1 over Rational Field
```

ainvs()

The a-invariants of this elliptic curve, as a list.

OUTPUT:

(list) - a (new) list of the 5 a-invariants.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.a_invariants()
[1, 2, 3, 4, 5]
sage: E = EllipticCurve([0,1])
sage: E
Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
sage: E.a_invariants()
[0, 0, 0, 0, 1]
sage: E = EllipticCurve([GF(7)(3),5])
sage: E.a_invariants()
[0, 0, 0, 3, 5]
```

We check that a new list is returned:

```
sage: E = EllipticCurve([1,0,0,0,1])
sage: E.a_invariants()[0] = 100000000
sage: E
Elliptic Curve defined by y^2 + x*y = x^3 + 1 over Rational Field
```

automorphisms(*field=None*)

Return the set of isomorphisms from self to itself (as a list).

INPUT:

- `field` (default None) – a field into which the coefficients of the curve may be coerced (by default, uses the base field of the curve).

OUTPUT:

(list) A list of `WeierstrassIsomorphism` objects consisting of all the isomorphisms from the curve `self` to itself defined over `field`.

EXAMPLES:

```
sage: E = EllipticCurve_from_j(QQ(0)) # a curve with j=0 over QQ
sage: E.automorphisms();
[Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 + y = x^3
Via:  (u,r,s,t) = (-1, 0, 0, -1), Generic endomorphism of Abelian group of points on Ellipti
Via:  (u,r,s,t) = (1, 0, 0, 0)]
```

We can also find automorphisms defined over extension fields:

```
sage: K.<a> = NumberField(x^2+3) # adjoin roots of unity
sage: E.automorphisms(K)
[Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 + y = x^3
Via:  (u,r,s,t) = (1, 0, 0, 0),
...
```

```
Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 o
Via:  (u,r,s,t) = (-1/2*a - 1/2, 0, 0, 0)]

sage: [ len(EllipticCurve_from_j(GF(q,'a')(0)).automorphisms()) for q in [2,4,3,9,5,25,7,49]
[2, 24, 2, 12, 2, 6, 6, 6]
```

b2()

Returns the b_2 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.b2()
9
```

b4()

Returns the b_4 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.b4()
11
```

b6()

Returns the b_6 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.b6()
29
```

b8()

Returns the b_8 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.b8()
35
```

b_invariants()

Returns the b-invariants of this elliptic curve, as a tuple.

OUTPUT:

(tuple) - a 4-tuple of the b-invariants of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.b_invariants()
(-4, -20, -79, -21)
sage: E = EllipticCurve([-4,0])
sage: E.b_invariants()
(0, -8, 0, -16)

sage: E = EllipticCurve([1,2,3,4,5])
sage: E.b_invariants()
(9, 11, 29, 35)
sage: E.b2()
9
sage: E.b4()
11
sage: E.b6()
```

```
29
sage: E.b8()
35
```

ALGORITHM:

These are simple functions of the a-invariants.

AUTHORS:

- William Stein (2005-04-25)

base_extend(*R*)

Returns a new curve with the same a-invariants but defined over a new ring.

INPUT:

- R – either a ring into which the curve's a-invariants may be coerced, or a morphism which may be applied to them.

OUTPUT:

A new elliptic curve with the same a-invariants, defined over the new ring.

EXAMPLES:

```
sage: E=EllipticCurve(GF(5),[1,1]); E
Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 5
sage: E1=E.base_extend(GF(125,'a')); E1
Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field in a of size 5^3
sage: F2=GF(5^2,'a'); a=F2.gen()
sage: F4=GF(5^4,'b'); b=F4.gen()
sage: h=F2.hom([a.charpoly().roots(ring=F4,multiplicities=False)[0]],F4)
sage: E=EllipticCurve(F2,[1,a]); E
Elliptic Curve defined by y^2 = x^3 + x + a over Finite Field in a of size 5^2
sage: E.base_extend(h)
Elliptic Curve defined by y^2 = x^3 + x + (4*b^3+4*b^2+4*b+3) over Finite Field in b of size
```

base_ring()

Returns the base ring of the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve(GF(49, 'a'), [3,5])
sage: E.base_ring()
Finite Field in a of size 7^2

sage: E = EllipticCurve([1,1])
sage: E.base_ring()
Rational Field

sage: E = EllipticCurve(ZZ, [3,5])
sage: E.base_ring()
Integer Ring
```

c4()

Returns the c_4 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.c4()
496
```

c6()

Returns the c_6 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.c6()
20008
```

c_invariants()

Returns the c-invariants of this elliptic curve, as a tuple.

OUTPUT:

(tuple) - a 2-tuple of the c-invariants of the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.c_invariants()
(496, 20008)
sage: E = EllipticCurve([-4,0])
sage: E.c_invariants()
(192, 0)
```

ALGORITHM:

These are simple functions of the a-invariants.

AUTHORS:

- William Stein (2005-04-25)

change_ring(R)

Returns a new curve with the same a-invariants but defined over a new ring.

INPUT:

- R – either a ring into which the curve's a-invariants may be coerced, or a morphism which may be applied to them.

OUTPUT:

A new elliptic curve with the same a-invariants, defined over the new ring.

EXAMPLES:

```
sage: E=EllipticCurve(GF(5),[1,1]); E
Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 5
sage: E1=E.base_extend(GF(125,'a')); E1
Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field in a of size 5^3
sage: F2=GF(5^2,'a'); a=F2.gen()
sage: F4=GF(5^4,'b'); b=F4.gen()
sage: h=F2.hom([a.charpoly().roots(ring=F4,multiplicities=False)[0]],F4)
sage: E=EllipticCurve(F2,[1,a]); E
Elliptic Curve defined by y^2 = x^3 + x + a over Finite Field in a of size 5^2
sage: E.base_extend(h)
Elliptic Curve defined by y^2 = x^3 + x + (4*b^3+4*b^2+4*b+3) over Finite Field in b of size
```

change_weierstrass_model(*$urst$)

Return a new Weierstrass model of self under the standard transformation $(u, r, s, , t)$

$$(x, y) \mapsto (x', y') = (u^2xr, u^3y + su^2x' + t).$$

EXAMPLES:

```
sage: E = EllipticCurve('15a')
sage: F1 = E.change_weierstrass_model([1/2,0,0,0]); F1
Elliptic Curve defined by y^2 + 2*x*y + 8*y = x^3 + 4*x^2 - 160*x - 640 over Rational Field
sage: F2 = E.change_weierstrass_model([7,2,1/3,5]); F2
Elliptic Curve defined by y^2 + 5/21*x*y + 13/343*y = x^3 + 59/441*x^2 - 10/7203*x - 58/1176
sage: F1.is_isomorphic(F2)
True
```

discriminant()

Returns the discriminant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E.discriminant()
37
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.discriminant()
-161051

sage: E = EllipticCurve([GF(7)(2),1])
sage: E.discriminant()
1
```

division_polynomial(*m, x=None, two_torsion_multiplicity=2*)

Returns the m^{th} division polynomial of this elliptic curve evaluated at x.

INPUT:

- m - positive integer.
- x - optional ring element to use as the "x" variable. If x is None, then a new polynomial ring will be constructed over the base ring of the elliptic curve, and its generator will be used as x. Note that x does not need to be a generator of a polynomial ring; any ring element is ok. This permits fast calculation of the torsion polynomial *evaluated* on any element of a ring.
- two_torsion_multiplicity - 0,1 or 2

  If 0: for even m when x is None, a univariate polynomial over the base ring of the curve is returned, which omits factors whose roots are the x-coordinates of the 2-torsion points. Similarly when x is not none, the evaluation of such a polynomial at x is returned.
  If 2: for even m when x is None, a univariate polynomial over the base ring of the curve is returned, which includes a factor of degree 3 whose roots are the x-coordinates of the 2-torsion points. Similarly when x is not none, the evaluation of such a polynomial at x is returned.
  If 1: when x is None, a bivariate polynomial over the base ring of the curve is returned, which includes a factor $2 * y + a1 * x + a3$ which has simple zeros at the 2-torsion points. When x is not none, it should be a tuple of length 2, and the evaluation of such a polynomial at x is returned.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E.division_polynomial(1)
1
sage: E.division_polynomial(2, two_torsion_multiplicity=0)
1
sage: E.division_polynomial(2, two_torsion_multiplicity=1)
2*y + 1
sage: E.division_polynomial(2, two_torsion_multiplicity=2)
4*x^3 - 4*x + 1
sage: E.division_polynomial(2)
4*x^3 - 4*x + 1
sage: [E.division_polynomial(3, two_torsion_multiplicity=i) for i in range(3)]
[3*x^4 - 6*x^2 + 3*x - 1, 3*x^4 - 6*x^2 + 3*x - 1, 3*x^4 - 6*x^2 + 3*x - 1]
sage: [type(E.division_polynomial(3, two_torsion_multiplicity=i)) for i in range(3)]
[<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_rational_dense'>,
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>,
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_rational_dense'>]
```

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: R.<z>=PolynomialRing(QQ)
sage: E.division_polynomial(4,z,0)
2*z^6 - 4*z^5 - 100*z^4 - 790*z^3 - 210*z^2 - 1496*z - 5821
sage: E.division_polynomial(4,z)
8*z^9 - 24*z^8 - 464*z^7 - 2758*z^6 + 6636*z^5 + 34356*z^4 + 53510*z^3 + 99714*z^2 + 351024*
```

This does not work, since when two_torsion_multiplicity is 1, we compute a bivariate polynomial, and must evaluate at a tuple of length 2:

```
sage: E.division_polynomial(4,z,1)
...
ValueError: x should be a tuple of length 2 (or None) when two_torsion_multiplicity is 1
sage: R.<z,w>=PolynomialRing(QQ,2)
sage: E.division_polynomial(4,(z,w),1).factor()
(2*w + 1) * (2*z^6 - 4*z^5 - 100*z^4 - 790*z^3 - 210*z^2 - 1496*z - 5821)
```

We can also evaluate this bivariate polynomial at a point:

```
sage: P = E(5,5)
sage: E.division_polynomial(4,P,two_torsion_multiplicity=1)
-1771561
```

division_polynomial_0 (*n, x=None, cache=None*)

Returns the n^{th} torsion (division) polynomial, without the 2-torsion factor if n is even, as a polynomial in x.

These are the polynomials g_n defined in Mazur/Tate ("The p-adic sigma function"), but with the sign flipped for even n, so that the leading coefficient is always positive.

Note: This function is intended for internal use; users should use `division_polynomial()`.

See Also:

`multiple_x_numerator()` `multiple_x_denominator()` `division_polynomial()`

INPUT:

- `n` - positive integer, or the special values -1 and -2 which mean $B_6 = (2y + a_1x + a_3)^2$ and B_6^2 respectively (in the notation of Mazur/Tate).
- `x` - optional ring element to use as the "x" variable. If x is None, then a new polynomial ring will be constructed over the base ring of the elliptic curve, and its generator will be used as x. Note that x does not need to be a generator of a polynomial ring; any ring element is ok. This permits fast calculation of the torsion polynomial *evaluated* on any element of a ring.
- `cache` - optional dictionary, with integer keys. If the key m is in cache, then cache[m] is assumed to be the value of division_polynomial_0(m) for the supplied x. New entries will be added to the cache as they are computed.

ALGORITHM:

Recursion described in Mazur/Tate. The recursive formulae are evaluated $O((logn)^2)$ times.

AUTHORS:

- David Harvey (2006-09-24): initial version
- John Cremona (2008-08-26): unified division polynomial code

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.division_polynomial_0(1)
1
sage: E.division_polynomial_0(2)
1
sage: E.division_polynomial_0(3)
3*x^4 - 6*x^2 + 3*x - 1
```

```
sage: E.division_polynomial_0(4)
2*x^6 - 10*x^4 + 10*x^3 - 10*x^2 + 2*x + 1
sage: E.division_polynomial_0(5)
5*x^12 - 62*x^10 + 95*x^9 - 105*x^8 - 60*x^7 + 285*x^6 - 174*x^5 - 5*x^4 - 5*x^3 + 35*x^2 -
sage: E.division_polynomial_0(6)
3*x^16 - 72*x^14 + 168*x^13 - 364*x^12 + 1120*x^10 - 1144*x^9 + 300*x^8 - 540*x^7 + 1120*x^6
sage: E.division_polynomial_0(7)
7*x^24 - 308*x^22 + 986*x^21 - 2954*x^20 + 28*x^19 + 17171*x^18 - 23142*x^17 + 511*x^16 - 50
sage: E.division_polynomial_0(8)
4*x^30 - 292*x^28 + 1252*x^27 - 5436*x^26 + 2340*x^25 + 39834*x^24 - 79560*x^23 + 51432*x^22

sage: E.division_polynomial_0(18) % E.division_polynomial_0(6) == 0
True
```

An example to illustrate the relationship with torsion points:

```
sage: F = GF(11)
sage: E = EllipticCurve(F, [0, 2]); E
Elliptic Curve defined by y^2  = x^3 + 2 over Finite Field of size 11
sage: f = E.division_polynomial_0(5); f
5*x^12 + x^9 + 8*x^6 + 4*x^3 + 7
sage: f.factor()
(5) * (x^2 + 5) * (x^2 + 2*x + 5) * (x^2 + 5*x + 7) * (x^2 + 7*x + 7) * (x^2 + 9*x + 5) * (x
```

This indicates that the x-coordinates of all the 5-torsion points of E are in $GF(11^2)$, and therefore the y-coordinates are in $\mathbf{F}_(11^4)$:

```
sage: K = GF(11^4, 'a')
sage: X = E.change_ring(K)
sage: f = X.division_polynomial_0(5)
sage: x_coords = f.roots(multiplicities=False); x_coords
[10*a^3 + 4*a^2 + 5*a + 6,
 9*a^3 + 8*a^2 + 10*a + 8,
 8*a^3 + a^2 + 4*a + 10,
 8*a^3 + a^2 + 4*a + 8,
 8*a^3 + a^2 + 4*a + 4,
 6*a^3 + 9*a^2 + 3*a + 4,
 5*a^3 + 2*a^2 + 8*a + 7,
 3*a^3 + 10*a^2 + 7*a + 8,
 3*a^3 + 10*a^2 + 7*a + 3,
 3*a^3 + 10*a^2 + 7*a + 1,
 2*a^3 + 3*a^2 + a + 7,
 a^3 + 7*a^2 + 6*a]
```

Now we check that these are exactly the x-coordinates of the 5-torsion points of E:

```
sage: for x in x_coords:
...       assert X.lift_x(x).order() == 5
```

The roots of the polynomial are the x-coordinates of the points P such that $mP = 0$ but $2P \neq 0$:

```
sage: E=EllipticCurve('14a1')
sage: T=E.torsion_subgroup()
sage: [n*T.0 for n in range(6)]
[(0 : 1 : 0),
(9 : 23 : 1),
(2 : 2 : 1),
(1 : -1 : 1),
(2 : -5 : 1),
(9 : -33 : 1)]
sage: pol=E.division_polynomial_0(6)
```

```
sage: xlist=pol.roots(multiplicities=False); xlist
[9, 2, -1/3, -5]
sage: [E.lift_x(x, all=True) for x in xlist]
[[(9 : 23 : 1), (9 : -33 : 1)], [(2 : 2 : 1), (2 : -5 : 1)], [], []]
```

Note: The point of order 2 and the identity do not appear. The points with $x = -1/3$ and $x = -5$ are not rational.

formal()

The formal group associated to this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.formal_group()
Formal Group associated to the Elliptic Curve defined by y^2 + y = x^3 - x over Rational Fie
```

formal_group()

The formal group associated to this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.formal_group()
Formal Group associated to the Elliptic Curve defined by y^2 + y = x^3 - x over Rational Fie
```

gen(i)

Function returning the i'th generator of this elliptic curve.

Note: Relies on gens() being implemented.

EXAMPLES:

```
sage: R.<a1,a2,a3,a4,a6>=QQ[]
sage: E=EllipticCurve([a1,a2,a3,a4,a6])
sage: E.gen(0)
...
NotImplementedError: not implemented.
```

gens()

Placeholder function to return generators of an elliptic curve.

Note: This functionality is implemented in certain derived classes, such as EllipticCurve_rational_field.

EXAMPLES:

```
sage: R.<a1,a2,a3,a4,a6>=QQ[]
sage: E=EllipticCurve([a1,a2,a3,a4,a6])
sage: E.gens()
...
NotImplementedError: not implemented.
sage: E=EllipticCurve(QQ,[1,1])
sage: E.gens()
[(0 : 1 : 1)]
```

hyperelliptic_polynomials()

Returns a pair of polynomials $g(x)$, $h(x)$ such that this elliptic curve can be defined by the standard hyperelliptic equation

$$y^2 + h(x)y = g(x).$$

EXAMPLES:

```
sage: R.<a1,a2,a3,a4,a6>=QQ[]
sage: E=EllipticCurve([a1,a2,a3,a4,a6])
```

```
sage: E.hyperelliptic_polynomials()
(x^3 + a2*x^2 + a4*x + a6, a1*x + a3)
```

is_isomorphic(*other, field=None*)

Returns whether or not self is isomorphic to other.

INPUT:

- `other` – another elliptic curve.
- `field` (default None) – a field into which the coefficients of the curves may be coerced (by default, uses the base field of the curves).

OUTPUT:

(bool) True if there is an isomorphism from curve `self` to curve `other` defined over `field`.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: F = E.change_weierstrass_model([2,3,4,5]); F
Elliptic Curve defined by y^2 + 4*x*y + 11/8*y = x^3 - 3/2*x^2 - 13/16*x over Rational Field
sage: E.is_isomorphic(F)
True
sage: E.is_isomorphic(F.change_ring(CC))
False
```

is_on_curve(*x, y*)

Returns True if (x, y) is an affine point on this curve.

INPUT:

- `x, y` - elements of the base ring of the curve.

EXAMPLES:

```
sage: E=EllipticCurve(QQ,[1,1])
sage: E.is_on_curve(0,1)
True
sage: E.is_on_curve(1,1)
False
```

is_x_coord(*x*)

Returns True if `x` is the x-coordinate of a point on this curve.

Note: See also `lift_x()` to find the point(s) with a given x-coordinate. This function may be useful in cases where testing an element of the base field for being a square is faster than finding its square root.

EXAMPLES:

```
sage: E = EllipticCurve('37a'); E
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: E.is_x_coord(1)
True
sage: E.is_x_coord(2)
True
```

There are no rational points with x-coordinate 3:

```
sage: E.is_x_coord(3)
False
```

However, there are such points in $E(\mathbf{R})$:

```
sage: E.change_ring(RR).is_x_coord(3)
True
```

And of course it always works in $E(\mathbf{C})$:

```
sage: E.change_ring(RR).is_x_coord(-3)
False
sage: E.change_ring(CC).is_x_coord(-3)
True
```

AUTHORS:

•John Cremona (2008-08-07): adapted from lift_x()

TEST:

```
sage: E=EllipticCurve('5077a1')
sage: [x for x in srange(-10,10) if E.is_x_coord (x)]
[-3, -2, -1, 0, 1, 2, 3, 4, 8]

sage: F=GF(32,'a')
sage: E=EllipticCurve(F,[1,0,0,0,1])
sage: set([P[0] for P in E.points() if P!=E(0)]) == set([x for x in F if E.is_x_coord(x)])
True
```

isomorphism_to(*other*)

Given another weierstrass model `other` of self, return an isomorphism from self to `other`.

INPUT:

•`other` – an elliptic curve isomorphic to `self`.

OUTPUT:

(Weierstrassmorphism) An isomorphism from self to other.

Note: If the curves in question are not isomorphic, a ValueError is raised.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: F = E.short_weierstrass_model()
sage: w = E.isomorphism_to(F); w
Generic morphism:
From: Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 - x over Rational F
To:   Abelian group of points on Elliptic Curve defined by y^2  = x^3 - 16*x + 16 over Ratio
Via:  (u,r,s,t) = (1/2, 0, 0, -1/2)
sage: P = E(0,-1,1)
sage: w(P)
(0 : -4 : 1)
sage: w(5*P)
(1 : 1 : 1)
sage: 5*w(P)
(1 : 1 : 1)
sage: 120*w(P) == w(120*P)
True
```

We can also handle injections to different base rings:

```
sage: K.<a> = NumberField(x^3-7)
sage: E.isomorphism_to(E.change_ring(K))
Generic morphism:
  From: Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 - x over Rational
  To:   Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 + (-1)*x over Num
  Via:  (u,r,s,t) = (1, 0, 0, 0)
```

isomorphisms(*other, field=None*)

Return the set of isomorphisms from self to other (as a list).

INPUT:

•`other` – another elliptic curve.

- `field` (default None) – a field into which the coefficients of the curves may be coerced (by default, uses the base field of the curves).

OUTPUT:

(list) A list of `WeierstrassIsomorphism` objects consisting of all the isomorphisms from the curve `self` to the curve `other` defined over `field`.

EXAMPLES:

```
sage: E = EllipticCurve_from_j(QQ(0)) # a curve with j=0 over QQ
sage: F = EllipticCurve('27a3') # should be the same one
sage: E.isomorphisms(F);
[Generic morphism:
From: Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 over Rational Field
To:   Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 over Rational Field
Via:  (u,r,s,t) = (-1, 0, 0, -1), Generic morphism:
From: Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 over Rational Field
To:   Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 over Rational Field
Via:  (u,r,s,t) = (1, 0, 0, 0)]
```

We can also find istomorphisms defined over extension fields:

```
sage: E=EllipticCurve(GF(7),[0,0,0,1,1])
sage: F=EllipticCurve(GF(7),[0,0,0,1,-1])
sage: E.isomorphisms(F)
[]
sage: E.isomorphisms(F,GF(49,'a'))
[Generic morphism:
From: Abelian group of points on Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Fie
To:   Abelian group of points on Elliptic Curve defined by y^2 = x^3 + x + 6 over Finite Fie
Via:  (u,r,s,t) = (a + 3, 0, 0, 0), Generic morphism:
From: Abelian group of points on Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Fie
To:   Abelian group of points on Elliptic Curve defined by y^2 = x^3 + x + 6 over Finite Fie
Via:  (u,r,s,t) = (6*a + 4, 0, 0, 0)]
```

j_invariant()

Returns the j-invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E.j_invariant()
110592/37
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.j_invariant()
-122023936/161051
sage: E = EllipticCurve([-4,0])
sage: E.j_invariant()
1728

sage: E = EllipticCurve([GF(7)(2),1])
sage: E.j_invariant()
1
```

lift_x(*x*, *all=False*)

Returns one or all points with given x-coordinate.

INPUT:

- `x` – an element of the base ring of the curve.
- `all` (bool, default False) – if True, return a (possibly empty) list of all points; if False, return just one point, or raise a ValueError if there are none.

Note: See also `is_x_coord()`.

EXAMPLES:

```
sage: E = EllipticCurve('37a'); E
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: E.lift_x(1)
(1 : 0 : 1)
sage: E.lift_x(2)
(2 : 2 : 1)
sage: E.lift_x(1/4, all=True)
[(1/4 : -3/8 : 1), (1/4 : -5/8 : 1)]
```

There are no rational points with x-coordinate 3:

```
sage: E.lift_x(3)
...
ValueError: No point with x-coordinate 3 on Elliptic Curve defined by y^2 + y = x^3 - x over
```

However, there are two such points in $E(\mathbf{R})$:

```
sage: E.change_ring(RR).lift_x(3, all=True)
[(3.00000000000000 : 4.42442890089805 : 1.00000000000000), (3.00000000000000 : -5.4244289008
```

And of course it always works in $E(\mathbf{C})$:

```
sage: E.change_ring(RR).lift_x(.5, all=True)
[]
sage: E.change_ring(CC).lift_x(.5)
(0.500000000000000 : -0.500000000000000 + 0.353553390593274*I : 1.00000000000000)
```

We can perform these operations over finite fields too:

```
sage: E = E.change_ring(GF(17)); E
Elliptic Curve defined by y^2 + y = x^3 + 16*x over Finite Field of size 17
sage: E.lift_x(7)
(7 : 11 : 1)
sage: E.lift_x(3)
...
ValueError: No point with x-coordinate 3 on Elliptic Curve defined by y^2 + y = x^3 + 16*x o
```

Note that there is only one lift with x-coordinate 10 in $E(\mathbf{F}_{17})$:

```
sage: E.lift_x(10, all=True)
[(10 : 8 : 1)]
```

We can lift over more exotic rings too:

```
sage: E = EllipticCurve('37a');
sage: E.lift_x(pAdicField(17, 5)(6))
(6 + O(17^5) : 2 + 16*17 + 16*17^2 + 16*17^3 + 16*17^4 + O(17^5) : 1 + O(17^5))
sage: K.<t> = PowerSeriesRing(QQ, 't', 5)
sage: E.lift_x(1+t)
(1 + t : 2*t - t^2 + 5*t^3 - 21*t^4 + O(t^5) : 1)
sage: K.<a> = GF(16)
sage: E = E.change_ring(K)
sage: E.lift_x(a^3)
(a^3 : a^3 + a : 1)
```

AUTHOR:

- Robert Bradshaw (2007-04-24)

TEST:

```
sage: E = EllipticCurve('37a').short_weierstrass_model().change_ring(GF(17))
sage: E.lift_x(3, all=True)
```

```
[]
sage: E.lift_x(7, all=True)
[(7 : 3 : 1), (7 : 14 : 1)]
```

multiplication_by_m(*m, x_only=False*)

Return the multiplication-by-m map from self to self as a pair of rational functions in two variables x,'y'.

INPUT:

- m - a nonzero integer
- x_only - bool (default: False) if True, return only the x-coordinate of the map.

OUTPUT:

(2-tuple) $(f(x), g(x,y))$, where f and g are rational functions with the degree of y in $g(x,y)$ exactly 1.

EXAMPLES:

```
sage: E = EllipticCurve([-1,3])
```

We verify that multiplication by 1 is just the identity:

```
sage: E.multiplication_by_m(1)
(x, y)
```

Multiplication by 2 is more complicated:

```
sage: f = E.multiplication_by_m(2)
sage: f
((x^4 + 2*x^2 - 24*x + 1)/(4*x^3 - 4*x + 12), (8*x^6*y - 40*x^4*y + 480*x^3*y - 40*x^2*y + 9
```

Grab only the x-coordinate (less work):

```
sage: E.multiplication_by_m(2, x_only=True)
(x^4 + 2*x^2 - 24*x + 1)/(4*x^3 - 4*x + 12)
```

We check that it works on a point:

```
sage: P = E([2,3])
sage: eval = lambda f,P: [fi(P[0],P[1]) for fi in f]
sage: assert E(eval(f,P)) == 2*P
```

We do the same but with multiplication by 3:

```
sage: f = E.multiplication_by_m(3)
sage: assert E(eval(f,P)) == 3*P
```

And the same with multiplication by 4:

```
sage: f = E.multiplication_by_m(4)
sage: assert E(eval(f,P)) == 4*P
```

And the same with multiplication by -1,-2,-3,-4:

```
sage: for m in [-1,-2,-3,-4]:
...       f = E.multiplication_by_m(m)
...       assert E(eval(f,P)) == m*P
```

TESTS:

Verify for this fairly random looking curve and point that multiplication by m returns the right result for the first 10 integers:

```
sage: E = EllipticCurve([23,-105])
sage: P = E([129/4, 1479/8])
sage: for n in [1..10]:
...       f = E.multiplication_by_m(n)
...       Q = n*P
```

```
...       assert Q == E(eval(f,P))
...       f = E.multiplication_by_m(-n)
...       Q = -n*P
...       assert Q == E(eval(f,P))
```

The following test shows that #4364 is indeed fixed:

```
sage: p = next_prime(2^30-41)
sage: a = GF(p)(1)
sage: b = GF(p)(1)
sage: E = EllipticCurve([a, b])
sage: P = E.random_point()
sage: my_eval = lambda f,P: [fi(P[0],P[1]) for fi in f]
sage: f = E.multiplication_by_m(2)
sage: assert(E(eval(f,P)) == 2*P)
```

pari_curve(*prec=53*)

Return the PARI curve corresponding to this elliptic curve.

Note: The result is cached; on subsequent calls the cached value is returned provided that it has sufficient precision, otherwise pari is called again with the new precision.

EXAMPLES:

```
sage: E = EllipticCurve([RR(0), RR(0), RR(1), RR(-1), RR(0)])
sage: e = E.pari_curve()
sage: type(e)
<type 'sage.libs.pari.gen.gen'>
sage: e.type()
't_VEC'
sage: e.disc()
37.0000000000000
```

plot(*xmin=None, xmax=None, **args*)

Draw a graph of this elliptic curve.

INPUT:

- `xmin, xmax` - (optional) points will be computed at least within this range, but possibly farther.
- `**args` - all other options are passed to the line graphing primitive.

EXAMPLES:

```
sage: E = EllipticCurve([0,-1])
sage: plot(E, rgbcolor=hue(0.7))
sage: E = EllipticCurve('37a')
sage: plot(E)
sage: plot(E, xmin=25,xmax=25)
```

rst_transform(*r, s, t*)

Returns the transform of the curve by (r, s, t) (with $u = 1$).

INPUT:

- `r, s, t` – three elements of the base ring.

OUTPUT:

The elliptic curve obtained from self by the standard Weierstrass transformation (u, r, s, t) with $u = 1$.

Note: This is just a special case of `change_weierstrass_model()`, with $u = 1$.

EXAMPLES:

```
sage: R.<r,s,t>=QQ[]
sage: E=EllipticCurve([1,2,3,4,5])
sage: E.rst_transform(r,s,t)
Elliptic Curve defined by y^2 + (2*s+1)*x*y + (r+2*t+3)*y = x^3 + (-s^2+3*r-s+2)*x^2 + (3*r^
```

scale_curve(*u*)

Returns the transform of the curve by scale factor u.
INPUT:

•u – an invertible element of the base ring.

OUTPUT:
The elliptic curve obtained from self by the standard Weierstrass transformation (u, r, s, t) with $r = s = t = 0$.
Note: This is just a special case of change_weierstrass_model(), with $r = s = t = 0$.

EXAMPLES:

```
sage: K=Frac(PolynomialRing(QQ,'u'))
sage: u=K.gen()
sage: E=EllipticCurve([1,2,3,4,5])
sage: E.scale_curve(u)
Elliptic Curve defined by y^2 + u*x*y + 3*u^3*y = x^3 + 2*u^2*x^2 + 4*u^4*x + 5*u^6 over Fra
```

short_weierstrass_model(*complete_cube=True*)

Returns a short Weierstrass model for self.
INPUT:

•complete_cube - bool (default: True); for meaning, see below.

OUTPUT:
An elliptic curve.
If complete_cube=True: Return a model of the form $y^2 = x^3 + a * x + b$ for this curve. The characteristic must not be 2; in characteristic 3, it is only possible if $b_2 = 0$.
If complete_cube=False: Return a model of the form $y^2 = x^3 + ax^2 + bx + c$ for this curve. The characteristic must not be 2.
EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: print E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over Rational Field
sage: F = E.short_weierstrass_model()
sage: print F
Elliptic Curve defined by y^2  = x^3 + 4941*x + 185166 over Rational Field
sage: E.is_isomorphic(F)
True
sage: F = E.short_weierstrass_model(complete_cube=False)
sage: print F
Elliptic Curve defined by y^2  = x^3 + 9*x^2 + 88*x + 464 over Rational Field
sage: print E.is_isomorphic(F)
True

sage: E = EllipticCurve(GF(3),[1,2,3,4,5])
sage: E.short_weierstrass_model(complete_cube=False)
Elliptic Curve defined by y^2 = x^3 + x + 2 over Finite Field of size 3
```

This used to be different see trac #3973:

```
sage: E.short_weierstrass_model()
Elliptic Curve defined by y^2 = x^3 + x + 2 over Finite Field of size 3
```

More tests in characteristic 3:

```
sage: E = EllipticCurve(GF(3),[0,2,1,2,1])
sage: E.short_weierstrass_model()
...
ValueError: short_weierstrass_model(): no short model for Elliptic Curve defined by y^2 + y
```

```
sage: E.short_weierstrass_model(complete_cube=False)
Elliptic Curve defined by y^2 = x^3 + 2*x^2 + 2*x + 2 over Finite Field of size 3
sage: E.short_weierstrass_model(complete_cube=False).is_isomorphic(E)
True
```

torsion_polynomial(*m, x=None, two_torsion_multiplicity=2*)

Returns the m^{th} division polynomial of this elliptic curve evaluated at x.

INPUT:

- m - positive integer.
- x - optional ring element to use as the "x" variable. If x is None, then a new polynomial ring will be constructed over the base ring of the elliptic curve, and its generator will be used as x. Note that x does not need to be a generator of a polynomial ring; any ring element is ok. This permits fast calculation of the torsion polynomial *evaluated* on any element of a ring.
- two_torsion_multiplicity - 0,1 or 2

  If 0: for even m when x is None, a univariate polynomial over the base ring of the curve is returned, which omits factors whose roots are the x-coordinates of the 2-torsion points. Similarly when x is not none, the evaluation of such a polynomial at x is returned.
  If 2: for even m when x is None, a univariate polynomial over the base ring of the curve is returned, which includes a factor of degree 3 whose roots are the x-coordinates of the 2-torsion points. Similarly when x is not none, the evaluation of such a polynomial at x is returned.
  If 1: when x is None, a bivariate polynomial over the base ring of the curve is returned, which includes a factor $2 * y + a1 * x + a3$ which has simple zeros at the 2-torsion points. When x is not none, it should be a tuple of length 2, and the evaluation of such a polynomial at x is returned.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E.division_polynomial(1)
1
sage: E.division_polynomial(2, two_torsion_multiplicity=0)
1
sage: E.division_polynomial(2, two_torsion_multiplicity=1)
2*y + 1
sage: E.division_polynomial(2, two_torsion_multiplicity=2)
4*x^3 - 4*x + 1
sage: E.division_polynomial(2)
4*x^3 - 4*x + 1
sage: [E.division_polynomial(3, two_torsion_multiplicity=i) for i in range(3)]
[3*x^4 - 6*x^2 + 3*x - 1, 3*x^4 - 6*x^2 + 3*x - 1, 3*x^4 - 6*x^2 + 3*x - 1]
sage: [type(E.division_polynomial(3, two_torsion_multiplicity=i)) for i in range(3)]
[<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_rational_dense'>,
<type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>,
<class 'sage.rings.polynomial.polynomial_element_generic.Polynomial_rational_dense'>]

sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: R.<z>=PolynomialRing(QQ)
sage: E.division_polynomial(4,z,0)
2*z^6 - 4*z^5 - 100*z^4 - 790*z^3 - 210*z^2 - 1496*z - 5821
sage: E.division_polynomial(4,z)
8*z^9 - 24*z^8 - 464*z^7 - 2758*z^6 + 6636*z^5 + 34356*z^4 + 53510*z^3 + 99714*z^2 + 351024*
```

This does not work, since when two_torsion_multiplicity is 1, we compute a bivariate polynomial, and must evaluate at a tuple of length 2:

```
sage: E.division_polynomial(4,z,1)
...
ValueError: x should be a tuple of length 2 (or None) when two_torsion_multiplicity is 1
sage: R.<z,w>=PolynomialRing(QQ,2)
sage: E.division_polynomial(4,(z,w),1).factor()
(2*w + 1) * (2*z^6 - 4*z^5 - 100*z^4 - 790*z^3 - 210*z^2 - 1496*z - 5821)
```

We can also evaluate this bivariate polynomial at a point:

```
sage: P = E(5,5)
sage: E.division_polynomial(4,P,two_torsion_multiplicity=1)
-1771561
```

two_division_polynomial(*x=None*)

Returns the 2-division polynomial of this elliptic curve evaluated at `x`.

INPUT:

- `x` - optional ring element to use as the x variable. If `x` is `None`, then a new polynomial ring will be constructed over the base ring of the elliptic curve, and its generator will be used as `x`. Note that `x` does not need to be a generator of a polynomial ring; any ring element is ok. This permits fast calculation of the torsion polynomial *evaluated* on any element of a ring.

EXAMPLES:

```
sage: E=EllipticCurve('5077a1')
sage: E.two_division_polynomial()
4*x^3 - 28*x + 25
sage: E=EllipticCurve(GF(3^2,'a'),[1,1,1,1,1])
sage: E.two_division_polynomial()
x^3 + 2*x^2 + 2
sage: E.two_division_polynomial().roots()
[(2, 1), (2*a, 1), (a + 2, 1)]
```

is_EllipticCurve(*x*)

Utility function to test if `x` is an instance of an Elliptic Curve class.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_generic import is_EllipticCurve
sage: E = EllipticCurve([1,2,3/4,7,19])
sage: is_EllipticCurve(E)
True
sage: is_EllipticCurve(0)
False
```

38.6 Elliptic curves over a general field

This module defines the class `EllipticCurve_field`, based on `EllipticCurve_generic`, for elliptic curves over general fields.

class EllipticCurve_field(*ainvs, extra=None*)

base_field()

Returns the base ring of the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve(GF(49, 'a'), [3,5])
sage: E.base_ring()
Finite Field in a of size 7^2

sage: E = EllipticCurve([1,1])
sage: E.base_ring()
Rational Field

sage: E = EllipticCurve(ZZ, [3,5])
sage: E.base_ring()
Integer Ring
```

is_quadratic_twist(*other*)

Determine whether this curve is a quadratic twist of another.

INPUT:

- `other` – an elliptic curves with the same base field as self.

OUTPUT:

Either 0, if the curves are not quadratic twists, or D if `other` is `self.quadratic_twist(D)` (up to isomorphism). If `self` and `other` are isomorphic, returns 1.

Note: Not fully implemented in characteristic 2, or in characteristic 3 when both j-invariants are 0.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: Et = E.quadratic_twist(-24)
sage: E.is_quadratic_twist(Et)
-6

sage: E1=EllipticCurve([0,0,1,0,0])
sage: E1.j_invariant()
0
sage: E2=EllipticCurve([0,0,0,0,2])
sage: E1.is_quadratic_twist(E2)
2

sage: E1=EllipticCurve([0,0,0,1,0])
sage: E1.j_invariant()
1728
sage: E2=EllipticCurve([0,0,0,2,0])
sage: E1.is_quadratic_twist(E2)
0
sage: E2=EllipticCurve([0,0,0,25,0])
sage: E1.is_quadratic_twist(E2)
5

sage: F = GF(101)
sage: E1 = EllipticCurve(F,[4,7])
sage: E2 = E1.quadratic_twist()
sage: D = E1.is_quadratic_twist(E2); D!=0
True
sage: F = GF(101)
sage: E1 = EllipticCurve(F,[4,7])
sage: E2 = E1.quadratic_twist()
sage: D = E1.is_quadratic_twist(E2)
sage: E1.quadratic_twist(D).is_isomorphic(E2)
True
sage: E1.is_isomorphic(E2)
False
```

```
sage: F2 = GF(101^2,'a')
sage: E1.change_ring(F2).is_isomorphic(E2.change_ring(F2))
True
```

A characteristic 3 example:

```
sage: F = GF(3^5,'a')
sage: E1 = EllipticCurve_from_j(F(1))
sage: E2 = E1.quadratic_twist(-1)
sage: D = E1.is_quadratic_twist(E2); D!=0
True
sage: E1.quadratic_twist(D).is_isomorphic(E2)
True

sage: E1 = EllipticCurve_from_j(F(0))
sage: E2 = E1.quadratic_twist()
sage: D = E1.is_quadratic_twist(E2); D
1
sage: E1.is_isomorphic(E2)
True
```

is_quartic_twist(*other*)

Determine whether this curve is a quartic twist of another.

INPUT:

- `other` – an elliptic curves with the same base field as self.

OUTPUT:

Either 0, if the curves are not quartic twists, or D if `other` is `self.quartic_twist(D)` (up to isomorphism). If `self` and `other` are isomorphic, returns 1.

Note: Not fully implemented in characteristics 2 or 3.

EXAMPLES:

```
sage: E = EllipticCurve_from_j(GF(13)(1728))
sage: E1 = E.quartic_twist(2)
sage: D = E.is_quartic_twist(E1); D!=0
True
sage: E.quartic_twist(D).is_isomorphic(E1)
True

sage: E = EllipticCurve_from_j(1728)
sage: E1 = E.quartic_twist(12345)
sage: D = E.is_quartic_twist(E1); D
15999120
sage: (D/12345).is_perfect_power(4)
True
```

is_sextic_twist(*other*)

Determine whether this curve is a sextic twist of another.

INPUT:

- `other` – an elliptic curves with the same base field as self.

OUTPUT:

Either 0, if the curves are not sextic twists, or D if `other` is `self.sextic_twist(D)` (up to isomorphism). If `self` and `other` are isomorphic, returns 1.

Note: Not fully implemented in characteristics 2 or 3.

EXAMPLES:

```
sage: E = EllipticCurve_from_j(GF(13)(0))
sage: E1 = E.sextic_twist(2)
sage: D = E.is_sextic_twist(E1); D!=0
True
sage: E.sextic_twist(D).is_isomorphic(E1)
True

sage: E = EllipticCurve_from_j(0)
sage: E1 = E.sextic_twist(12345)
sage: D = E.is_sextic_twist(E1); D
575968320
sage: (D/12345).is_perfect_power(6)
True
```

quadratic_twist(*D=None*)

Return the quadratic twist of this curve by D.

INPUT:

- D (default None) the twisting parameter (see below).

In characteristics other than 2, D must be nonzero, and the twist is isomorphic to self after adjoining $\sqrt{(D)}$ to the base.

In characteristic 2, D is arbitrary, and the twist is isomorphic to self after adjoining a root of $x^2 + x + D$ to the base.

In characteristic 2 when $j = 0$, this is not implemented.

If the base field F is finite, D need not be specified, and the curve returned is the unique curve (up to isomorphism) defined over F isomorphic to the original curve over the quadratic extension of F but not over F itself. Over infinte fields, an error is raised if D is not given.

EXAMPLES:

```
sage: E = EllipticCurve([GF(1103)(1), 0, 0, 107, 340]); E
Elliptic Curve defined by y^2 + x*y  = x^3 + 107*x + 340 over Finite Field of size 1103
sage: F=E.quadratic_twist(-1); F
Elliptic Curve defined by y^2  = x^3 + 1102*x^2 + 609*x + 300 over Finite Field of size 1103
sage: E.is_isomorphic(F)
False
sage: E.is_isomorphic(F,GF(1103^2,'a'))
True
```

A characteristic 2 example:

```
sage: E=EllipticCurve(GF(2),[1,0,1,1,1])
sage: E1=E.quadratic_twist(1)
sage: E.is_isomorphic(E1)
False
sage: E.is_isomorphic(E1,GF(4,'a'))
True
```

Over finite fields, the twisting parameter may be omitted:

```
sage: k.<a> = GF(2^10)
sage: E = EllipticCurve(k,[a^2,a,1,a+1,1])
sage: Et = E.quadratic_twist()
sage: Et # random (only determined up to isomorphism)
Elliptic Curve defined by y^2 + x*y  = x^3 + (a^7+a^4+a^3+a^2+a+1)*x^2 + (a^8+a^6+a^4+1) ove
sage: E.is_isomorphic(Et)
False
sage: E.j_invariant()==Et.j_invariant()
True
```

```
sage: p=next_prime(10^10)
sage: k = GF(p)
sage: E = EllipticCurve(k,[1,2,3,4,5])
sage: Et = E.quadratic_twist()
sage: Et # random (only determined up to isomorphism)
Elliptic Curve defined by y^2  = x^3 + 7860088097*x^2 + 9495240877*x + 3048660957 over Finit
sage: E.is_isomorphic(Et)
False
sage: k2 = GF(p^2,'a')
sage: E.change_ring(k2).is_isomorphic(Et.change_ring(k2))
True
```

quartic_twist(*D*)

Return the quartic twist of this curve by D.

INPUT:

•D (must be nonzero) – the twisting parameter..

Note: The characteristic must not be 2 or 3, and the j-invariant must be 1728.

EXAMPLES:

```
sage: E=EllipticCurve_from_j(GF(13)(1728)); E
Elliptic Curve defined by y^2  = x^3 + x over Finite Field of size 13
sage: E1=E.quartic_twist(2); E1
Elliptic Curve defined by y^2  = x^3 + 5*x over Finite Field of size 13
sage: E.is_isomorphic(E1)
False
sage: E.is_isomorphic(E1,GF(13^2,'a'))
False
sage: E.is_isomorphic(E1,GF(13^4,'a'))
True
```

sextic_twist(*D*)

Return the quartic twist of this curve by D.

INPUT:

•D (must be nonzero) – the twisting parameter..

Note: The characteristic must not be 2 or 3, and the j-invariant must be 0.

EXAMPLES:

```
sage: E=EllipticCurve_from_j(GF(13)(0)); E
Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 13
sage: E1=E.sextic_twist(2); E1
Elliptic Curve defined by y^2 = x^3 + 11 over Finite Field of size 13
sage: E.is_isomorphic(E1)
False
sage: E.is_isomorphic(E1,GF(13^2,'a'))
False
sage: E.is_isomorphic(E1,GF(13^4,'a'))
False
sage: E.is_isomorphic(E1,GF(13^6,'a'))
True
```

38.7 Elliptic curves over the rational numbers

AUTHORS:

- William Stein (2005): first version

- William Stein (2006-02-26): fixed Lseries_extended which didn't work because of changes elsewhere in Sage.
- David Harvey (2006-09): Added padic_E2, padic_sigma, padic_height, padic_regulator methods.
- David Harvey (2007-02): reworked padic-height related code
- Christian Wuthrich (2007): added padic sha computation
- David Roe (2007-09): moved sha, l-series and p-adic functionality to separate files.
- John Cremona (2008-01)
- Tobias Nagel and Michael Mardaus (2008-07): added integral_points
- John Cremona (2008-07): further work on integral_points

class **EllipticCurve_rational_field**(*ainvs, extra=None*)

Elliptic curve over the Rational Field.

INPUT:

- `ainvs` (list or string) – either $[a_1, a_2, a_3, a_4, a_6]$ or $[a_4, a_6]$ (with $a_1 = a_2 = a_3 = 0$) or a valid label from the database.

Note: See constructor.py for more variants.

EXAMPLES:

Construction from Weierstrass coeffiecients (a-invariants), long form:

```
sage: E = EllipticCurve([1,2,3,4,5]); E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over Rational Field
```

Construction from Weierstrass coeffiecients (a-invariants), short form (sets $a_1 = a_2 = a_3 = 0$):

```
sage: EllipticCurve([4,5]).ainvs()
[0, 0, 0, 4, 5]
```

Construction from a database label:

```
sage: EllipticCurve('389a1')
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
```

CPS_height_bound()

Return the Cremona-Prickett-Siksek height bound. This is a floating point number B such that if P is a point on the curve, then the naive logarithmic height of P differs from the canonical height by at most B.

EXAMPLES:

```
sage: E = EllipticCurve("11a")
sage: E.CPS_height_bound()
2.8774743273580445
sage: E = EllipticCurve("5077a")
sage: E.CPS_height_bound()
0.0
sage: E = EllipticCurve([1,2,3,4,1])
sage: E.CPS_height_bound()
...
RuntimeError: curve must be minimal.
sage: F = E.quadratic_twist(-19)
sage: F
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 + 1376*x - 130 over Rational Field
sage: F.CPS_height_bound()
0.65551583769728516
```

IMPLEMENTATION:

Call the corresponding mwrank C++ library function.

Lambda (*s, prec*)

Returns the value of the Lambda-series of the elliptic curve E at s, where s can be any complex number.

IMPLEMENTATION: Fairly *slow* computation using the definitions and implemented in Python.

Uses prec terms of the power series.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: E.Lambda(1.4+0.5*I, 50)
-0.354172680517... + 0.874518681720...*I
```

Np (*p*)

The number of points on E modulo p.

INPUT:

- p (int) – a prime, not necessarily of good reduction.

OUTPUT:

(int) The number ofpoints on the reduction of E modulo p (including the singular point when p is a prime of bad reduction).

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.Np(2)
5
sage: E.Np(3)
5
sage: E.conductor()
11
sage: E.Np(11)
11
```

S_integral_points (*S, mw_base='auto', both_signs=False, verbose=False, proof=None*)

Computes all S-integral points (up to sign) on this elliptic curve.

INPUT:

- S - list of primes
- mw_base - list of EllipticCurvePoint generating the Mordell-Weil group of E (default: 'auto' - calls gens())
- both_signs - True/False (default False): if True the output contains both P and -P, otherwise only one of each pair.
- verbose - True/False (default False): if True, some details of the computation are output.
- proof - True/False (default True): if True ALL S-integral points will be returned. If False, the MW basis will be computed with the proof=False flag, and also the time-consuming final call to S_integral_x_coords_with_abs_bounded_by(abs_bound) is omitted. Use this only if the computation takes too long, but be warned that then it cannot be guaranteed that all S-integral points will be found.

OUTPUT:

A sorted list of all the S-integral points on E (up to sign unless both_signs is True)

Note: The complexity increases exponentially in the rank of curve E and in the length of S. The computation time (but not the output!) depends on the Mordell-Weil basis. If mw_base is given but is not a basis for the Mordell-Weil group (modulo torsion), S-integral points which are not in the subgroup generated by the given points will almost certainly not be listed.

EXAMPLES:

A curve of rank 3 with no torsion points:

```
sage: E=EllipticCurve([0,0,1,-7,6])
sage: P1=E.point((2,0)); P2=E.point((-1,3)); P3=E.point((4,6))
sage: a=E.S_integral_points(S=[2,3], mw_base=[P1,P2,P3], verbose=True);a
max_S: 3 len_S: 3 len_tors: 1
lambda 0.485997517468...
k1,k2,k3,k4 6.68597129142710e234 1.31952866480763 3.31908110593519e9 2.42767548272846e17
p= 2 : trying with p_prec =  30
mw_base_p_log_val =  [2, 2, 1]
min_psi =  2 + 2^3 + 2^6 + 2^7 + 2^8 + 2^9 + 2^11 + 2^12 + 2^13 + 2^16 + 2^17 + 2^19 + 2^20
p= 3 : trying with p_prec =  30
mw_base_p_log_val =  [1, 2, 1]
min_psi =  3 + 3^2 + 2*3^3 + 3^6 + 2*3^7 + 2*3^8 + 3^9 + 2*3^11 + 2*3^12 + 2*3^13 + 3^15 + 2
mw_base [(1 : -1 : 1), (2 : 0 : 1), (0 : -3 : 1)]
mw_base_log [0.667789378224099, 0.552642660712417, 0.818477222895703]
mp [5, 7]
mw_base_p_log [[2^2 + 2^3 + 2^6 + 2^7 + 2^8 + 2^9 + 2^14 + 2^15 + 2^18 + 2^19 + 2^24 + 2^29
k5,k6,k7 0.321154513240... 1.55246328915... 0.161999172489...
initial bound 2.6227097483365...e117
bound_list [58, 58, 58]
bound_list [8, 9, 9]
bound_list [8, 7, 7]
bound_list [8, 7, 7]
starting search of points using coefficient bound  8
x-coords of S-integral points via linear combination of mw_base and torsion:
[-3, -26/9, -8159/2916, -2759/1024, -151/64, -1343/576, -2, -7/4, -1, -47/256, 0, 1/4, 4/9,
starting search of extra S-integer points with absolute value bounded by 3.89321964979420
x-coords of points with bounded absolute value
[-3, -2, -1, 0, 1, 2]
Total number of S-integral points: 43
[(-3 : 0 : 1), (-26/9 : 28/27 : 1), (-8159/2916 : 233461/157464 : 1), (-2759/1024 : 60819/32
```

It is not necessary to specify mw_base; if it is not provided, then the Mordell-Weil basis must be computed, which may take much longer.

```
sage: a = E.S_integral_points([2,3])
sage: len(a)
43
```

An example with negative discriminant:

```
sage: EllipticCurve('900d1').S_integral_points([17], both_signs=True)
[(-11 : -27 : 1), (-11 : 27 : 1), (-4 : -34 : 1), (-4 : 34 : 1), (4 : -18 : 1), (4 : 18 : 1)
```

Output checked with Magma (corrected in 3 cases):

```
sage: [len(e.S_integral_points([2], both_signs=False)) for e in cremona_curves([11..100])] #
[2, 0, 2, 3, 3, 1, 3, 1, 3, 5, 3, 5, 4, 1, 1, 2, 2, 2, 3, 1, 2, 1, 0, 1, 3, 3, 1, 1, 5, 3, 4
```

An example from [PZGH]:

```
sage: E = EllipticCurve([0,0,0,-172,505])
sage: E.rank(), len(E.S_integral_points([3,5,7]))  # long time (~7s)
(4, 72)
```

This is curve "7690e1" which failed until #4805 was fixed:

```
sage: EllipticCurve([1,1,1,-301,-1821]).S_integral_points([13,2])
[(-13 : 16 : 1),
(-9 : 20 : 1),
(-7 : 4 : 1),
(21 : 30 : 1),
(23 : 52 : 1),
```

```
(63 : 452 : 1),
(71 : 548 : 1),
(87 : 756 : 1),
(2711 : 139828 : 1),
(7323 : 623052 : 1),
(17687 : 2343476 : 1)]
```

REFERENCES:

- [PZGH] Petho A., Zimmer H.G., Gebel J. and Herrmann E., Computing all S-integral points on elliptic curves Math. Proc. Camb. Phil. Soc. (1999), 127, 383-402
- Some parts of this implementation are partially based on the function integral_points()

AUTHORS:

- Tobias Nagel (2008-12)
- Michael Mardaus (2008-12)
- John Cremona (2008-12)

an(*n*)

The n-th Fourier coefficient of the modular form corresponding to this elliptic curve, where n is a positive integer.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: [E.an(n) for n in range(20) if n>0]
[1, -2, -3, 2, -2, 6, -1, 0, 6, 4, -5, -6, -2, 2, 6, -4, 0, -12, 0]
```

analytic_rank(*algorithm='cremona'*)

Return an integer that is *probably* the analytic rank of this elliptic curve.

INPUT:

- algorithm -
  - 'cremona' (default) - Use the Buhler-Gross algorithm as implemented in GP by Tom Womack and John Cremona, who note that their implementation is practical for any rank and conductor $\leq 10^{10}$ in 10 minutes.
  - 'sympow' -use Watkins's program sympow
  - 'rubinstein' - use Rubinstein's L-function C++ program lcalc.
  - 'magma' - use MAGMA
  - 'all' - compute with all other free algorithms, check that the answers agree, and return the common answer.

Note: If the curve is loaded from the large Cremona database, then the modular degree is taken from the database.

Of the three above, probably Rubinstein's is the most efficient (in some limited testing I've done).

Note: It is an open problem to *prove* that *any* particular elliptic curve has analytic rank ≥ 4.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: E.analytic_rank(algorithm='cremona')
2
sage: E.analytic_rank(algorithm='rubinstein')
2
sage: E.analytic_rank(algorithm='sympow')
2
sage: E.analytic_rank(algorithm='magma')     # optional - magma
2
sage: E.analytic_rank(algorithm='all')
2
```

TESTS:

When the input is horrendous, some of the algorithms just bomb out with a RuntimeError:

```
sage: EllipticCurve([1234567,89101112]).analytic_rank(algorithm='rubinstein')
...
RuntimeError: unable to compute analytic rank using rubinstein algorithm ('unable to convert
sage: EllipticCurve([1234567,89101112]).analytic_rank(algorithm='sympow')
...
RuntimeError: failed to compute analytic rank
```

anlist(*n, python_ints=False*)

The Fourier coefficients up to and including a_n of the modular form attached to this elliptic curve. The i-th element of the return list is a[i].

INPUT:

- n - integer
- python_ints - bool (default: False); if True return a list of Python ints instead of Sage integers.

OUTPUT: list of integers

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.anlist(3)
[0, 1, -2, -1]

sage: E = EllipticCurve([0,1])
sage: E.anlist(20)
[0, 1, 0, 0, 0, 0, 0, -4, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 8, 0]
```

antilogarithm(*z, prec=None*)

Returns the rational point (if any) associated to this complex number; the inverse of the elliptic logarithm function.

INPUT:

- z - a complex number representing an element of CC/L where L is the period lattice of the elliptic curve
- precision - an integer specifying the precision (in bits) which will be used for the computation (default real precision if None)

OUTPUT: The rational point which is the image of z under the Weierstrass parametrization, if it exists and can be determined from z with default precision.

Note: This uses the function ellztopoint from the pari library

TODO: Extend the wrapping of ellztopoint() to allow passing of the precision parameter.

ap(*p*)

The p-th Fourier coefficient of the modular form corresponding to this elliptic curve, where p is prime.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: [E.ap(p) for p in prime_range(50)]
[-2, -3, -2, -1, -5, -2, 0, 0, 2, 6, -4, -1, -9, 2, -9]
```

aplist(*n, python_ints=False*)

The Fourier coefficients a_p of the modular form attached to this elliptic curve, for all primes $p \leq n$.

INPUT:

- n - integer
- python_ints - bool (default: False); if True return a list of Python ints instead of Sage integers.

OUTPUT: list of integers

EXAMPLES:

```
sage: e = EllipticCurve('37a')
sage: e.aplist(1)
[]
sage: e.aplist(2)
[-2]
sage: e.aplist(10)
[-2, -3, -2, -1]
sage: v = e.aplist(13); v
[-2, -3, -2, -1, -5, -2]
sage: type(v[0])
<type 'sage.rings.integer.Integer'>
sage: type(e.aplist(13, python_ints=True)[0])
<type 'int'>
```

cm_discriminant()

Returns the associated quadratic discriminant if this elliptic curve has Complex Multiplication.

A ValueError is raised if the curve does not have CM (see the function has_cm()).

EXAMPLES:

```
sage: E=EllipticCurve('32a1')
sage: E.cm_discriminant()
-4
sage: E=EllipticCurve('121b1')
sage: E.cm_discriminant()
-11
sage: E=EllipticCurve('37a1')
sage: E.cm_discriminant()
...
ValueError: Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field does not have CM
```

conductor(*algorithm='pari'*)

Returns the conductor of the elliptic curve.

INPUT:

- `algorithm` - str, (default: "pari")
  - `"pari"` - use the PARI C-library ellglobalred implementation of Tate's algorithm
  - `"mwrank"` - use Cremona's mwrank implementation of Tate's algorithm; can be faster if the curve has integer coefficients (TODO: limited to small conductor until mwrank gets integer factorization)
  - `"gp"` - use the GP interpreter.
  - `"generic"` - use the general number field implementation
  - `"all"` - use all four implementations, verify that the results are the same (or raise an error), and output the common value.

EXAMPLE:

```
sage: E = EllipticCurve([1, -1, 1, -29372, -1932937])
sage: E.conductor(algorithm="pari")
3006
sage: E.conductor(algorithm="mwrank")
3006
sage: E.conductor(algorithm="gp")
3006
sage: E.conductor(algorithm="generic")
3006
sage: E.conductor(algorithm="all")
3006
```

Note: The conductor computed using each algorithm is cached separately. Thus calling `E.conductor('pari')`, then `E.conductor('mwrank')` and getting the same result checks that both systems compute the same answer.

congruence_number()

Let X be the subspace of $S_2(\Gamma_0(N))$ spanned by the newform associated with this elliptic curve, and Y be orthogonal compliment of X under the Petersson inner product. Let S_X and S_Y be the intersections of X and Y with $S_2(\Gamma_0(N)m\mathbf{Z})$. The congruence number is defined to be $[S_X \oplus S_Y : S_2(\Gamma_0(N), \mathbf{Z})]$. It measures congruences between f and elements of $S_2(\Gamma_0(N), \mathbf{Z})$ orthogonal to f.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.congruence_number()
2
sage: E.congruence_number()
2
sage: E = EllipticCurve('54b')
sage: E.congruence_number()
6
sage: E.modular_degree()
2
sage: E = EllipticCurve('242a1')
sage: E.modular_degree()
16
sage: E.congruence_number()  # long time
176
```

It is a theorem of Ribet that the congruence number is equal to the modular degree in the case of square free conductor. It is a conjecture of Agashe, Ribet, and Stein that $ord_p(c_f/m_f) \leq ord_p(N)/2$.

TESTS:

```
sage: E = EllipticCurve('11a')
sage: E.congruence_number()
1
```

cremona_label(*space=False*)

Return the Cremona label associated to (the minimal model) of this curve, if it is known. If not, raise a RuntimeError exception.

EXAMPLES:

```
sage: E=EllipticCurve('389a1')
sage: E.cremona_label()
'389a1'
```

The default database only contains conductors up to 10000, so any curve with conductor greater than that will cause an error to be raised. The optional package 'database_cremona_ellcurve-20071019' contains more curves, with conductors up to 130000.

```
sage: E = EllipticCurve([1, -1, 0, -79, 289])
sage: E.conductor()
234446
sage: E.cremona_label()
...
RuntimeError: Cremona label not known for Elliptic Curve defined by y^2 + x*y = x^3 - x^2 -
```

database_curve()

Return the curve in the elliptic curve database isomorphic to this curve, if possible. Otherwise raise a RuntimeError exception.

EXAMPLES:

```
sage: E = EllipticCurve([0,1,2,3,4])
sage: E.database_curve()
Elliptic Curve defined by y^2  = x^3 + x^2 + 3*x + 5 over Rational Field
```

Note: The model of the curve in the database can be different from the Weierstrass model for this curve, e.g., database models are always minimal.

eval_modular_form(*points, prec*)

Evaluate the L-series of this elliptic curve at points in CC

INPUT:

- points - a list of points in the half-plane of convergence
- prec - precision

OUTPUT: A list of values L(E,s) for s in points

Note: Better examples are welcome.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.eval_modular_form([1.5+I,2.0+I,2.5+I],0.000001)
[0, 0, 0]
```

gens(*verbose=False, rank1_search=10, algorithm='mwrank_shell', only_use_mwrank=True, proof=None, use_database=True*)

Compute and return generators for the Mordell-Weil group E(Q) *modulo* torsion.

HINT: If you would like to control the height bounds used in the 2-descent, first call the two_descent function with those height bounds. However that function, while it displays a lot of output, returns no values.

TODO: Allow passing of command-line parameters to mwrank.

Warning: If the program fails to give a provably correct result, it prints a warning message, but does not raise an exception. Use the gens_certain command to find out if this warning message was printed.

INPUT:

- verbose - (default: None), if specified changes the verbosity of mwrank computations.
- rank1_search - (default: 16), if the curve has analytic rank 1, try to find a generator by a direct search up to this logarithmic height. If this fails the usual mwrank procedure is called. algorithm -
- – 'mwrank_shell' (default) - call mwrank shell command
- – 'mwrank_lib' - call mwrank c library
- only_use_mwrank - bool (default True) if False, attempts to first use more naive, natively implemented methods.
- proof - bool or None (default None, see proof.elliptic_curve or sage.structure.proof).
- use_database - bool (default True) if True, attempts to find curve and gens in the (optional) database

OUTPUT:

- generators - List of generators for the Mordell-Weil group modulo torsion.

IMPLEMENTATION: Uses Cremona's mwrank C library.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: E.gens()                  # random output
[(-1 : 1 : 1), (0 : 0 : 1)]
```

A non-integral example:

```
sage: E = EllipticCurve([-3/8,-2/3])
sage: E.gens() # random (up to sign)
[(10/9 : 29/54 : 1)]
```

A non-minimal example:

```
sage: E = EllipticCurve('389a1')
sage: E1 = E.change_weierstrass_model([1/20,0,0,0]); E1
Elliptic Curve defined by y^2 + 8000*y = x^3 + 400*x^2 - 320000*x over Rational Field
sage: E1.gens() # random (if database not used)
[(-400 : 8000 : 1), (0 : -8000 : 1)]
```

gens_certain()

Return True if the generators have been proven correct.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.gens()                    # random (up to sign)
[(0 : -1 : 1)]
sage: E.gens_certain()
True
```

global_integral_model()

Return a model of self which is integral at all primes.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1/216, -7/1296, 1/7776])
sage: F = E.global_integral_model(); F
Elliptic Curve defined by y^2 + y = x^3 - 7*x + 6 over Rational Field
sage: F == EllipticCurve('5077a1')
True
```

has_cm()

Returns True iff this elliptic curve has Complex Multiplication.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.has_cm()
False
sage: E=EllipticCurve('32a1')
sage: E.has_cm()
True
sage: E.j_invariant()
1728
```

has_good_reduction_outside_S(*S=, []*)

Tests if this elliptic curve has good reduction outside S.

INPUT:

- S - list of primes (default: empty list).

Note: Primality of elements of S is not checked, and the output is undefined if S is not a list or contains non-primes.

This only tests the given model, so should only be applied to minimal models.

EXAMPLES:

```
sage: EllipticCurve('11a1').has_good_reduction_outside_S([11])
True
sage: EllipticCurve('11a1').has_good_reduction_outside_S([2])
False
```

```
sage: EllipticCurve('2310a1').has_good_reduction_outside_S([2,3,5,7])
False
sage: EllipticCurve('2310a1').has_good_reduction_outside_S([2,3,5,7,11])
True
```

heegner_discriminants(*bound*)

Return the list of self's Heegner discriminants between -1 and -bound.

INPUT:

- `bound (int)` - upper bound for -discriminant

OUTPUT: The list of Heegner discriminants between -1 and -bound for the given elliptic curve.

EXAMPLES:

```
sage: E=EllipticCurve('11a')
sage: E.heegner_discriminants(30)
[-7, -8, -19, -24]
```

heegner_discriminants_list(*n*)

Return the list of self's first n Heegner discriminants smaller than -5.

INPUT:

- `n (int)` - the number of discriminants to compute

OUTPUT: The list of the first n Heegner discriminants smaller than -5 for the given elliptic curve.

EXAMPLE:

```
sage: E=EllipticCurve('11a')
sage: E.heegner_discriminants_list(4)
[-7, -8, -19, -24]
```

heegner_index(*D, min_p=2, prec=5, verbose=False*)

Return an interval that contains the index of the Heegner point y_K in the group of K-rational points modulo torsion on this elliptic curve, computed using the Gross-Zagier formula and/or a point search, or the index divided by 2.

Note: If `min_p` is bigger than 2 then the index can be off by any prime less than `min_p`. This function returns the index divided by 2 exactly when $E(\mathbf{Q})_{/tor}$ has index 2 in $E(K)_{/tor}$.

INPUT:

- `D (int)` - Heegner discriminant
- `min_p (int)` - (default: 2) only rule out primes = min_p dividing the index.
- `verbose (bool)` - (default: False); print lots of mwrank search status information when computing regulator
- `prec (int)` - (default: 5), use prec*sqrt(N) + 20 terms of L-series in computations, where N is the conductor.

OUTPUT: an interval that contains the index

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: E.heegner_discriminants(50)
[-7, -8, -19, -24, -35, -39, -40, -43]
sage: E.heegner_index(-7)
1.00000?

sage: E = EllipticCurve('37b')
sage: E.heegner_discriminants(100)
[-3, -4, -7, -11, -40, -47, -67, -71, -83, -84, -95]
sage: E.heegner_index(-95)           # long time (1 second)
2.00000?
```

This tests doing direct computation of the Mordell-Weil group.

```
sage: EllipticCurve('675b').heegner_index(-11)
3.0000?
```

Currently discriminants -3 and -4 are not supported:

```
sage: E.heegner_index(-3)
...
ArithmeticError: Discriminant (=-3) must not be -3 or -4.
```

The curve 681b returns an interval that contains $3/2$. This is because $E(\mathbf{Q})$ is not saturated in $E(K)$. The true index is 3:

```
sage: E = EllipticCurve('681b')
sage: I = E.heegner_index(-8); I
1.50000?
sage: 2*I
3.0000?
```

In fact, whenever the returned index has a denominator of 2, the true index is got by multiplying the returned index by 2. Unfortunately, this is not an if and only if condition, i.e., sometimes the index must be multiplied by 2 even though the denominator is not 2.

heegner_index_bound(*D=0, prec=5, verbose=True, max_height=21*)

Assume self has rank 0.

Return a list v of primes such that if an odd prime p divides the index of the Heegner point in the group of rational points *modulo torsion*, then p is in v.

If 0 is in the interval of the height of the Heegner point computed to the given prec, then this function returns v = 0. This does not mean that the Heegner point is torsion, just that it is very likely torsion.

If we obtain no information from a search up to max_height, e.g., if the Siksek et al. bound is bigger than max_height, then we return v = -1.

INPUT:

- `D (int)` - (default: 0) Heegner discriminant; if 0, use the first discriminant -4 that satisfies the Heegner hypothesis
- `verbose (bool)` - (default: True)
- `prec (int)` - (default: 5), use prec*sqrt(N) + 20 terms of L-series in computations, where N is the conductor.
- `max_height (float)` - should be = 21; bound on logarithmic naive height used in point searches. Make smaller to make this function faster, at the expense of possibly obtaining a worse answer. A good range is between 13 and 21.

OUTPUT:

- `v` - list or int (bad primes or 0 or -1)
- `D` - the discriminant that was used (this is useful if D was automatically selected).

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.heegner_index_bound(verbose=False)
([2], -7)
```

heegner_point_height(*D, prec=2*)

Use the Gross-Zagier formula to compute the Neron-Tate canonical height over K of the Heegner point corresponding to D, as an Interval (since it's computed to some precision using L-functions).

INPUT:

- `D (int)` - fundamental discriminant (=/= -3, -4)
- `prec (int)` - (default: 2), use prec*sqrt(N) + 20 terms of L-series in computations, where N is the conductor.

OUTPUT: Interval that contains the height of the Heegner point.

EXAMPLE:

```
sage: E = EllipticCurve('11a')
sage: E.heegner_point_height(-7)
0.22227?
```

heegner_sha_an (*D, prec=53*)

Return the conjectural (analytic) order of Sha

INPUT:

- D – negative integer; the Heegner discriminant
- prec – integer (default: 53); bits of precision to compute analytic order of Sha

OUTPUT:

(floating point number) an approximation to the conjectural order of Sha.

Note: Often you'll want to do `proof.elliptic_curve(False)` when using this function, since often the twisted elliptic curves that come up have enormous conductor, and Sha is nontrivial, which makes provably finding the Mordell-Weil group using 2-descent difficult.

EXAMPLES:

An example where E has conductor 11:

```
sage: E = EllipticCurve('11a')
sage: E.heegner_sha_an(-7)                                  # long
1.00000000000000
```

The cache works:

```
sage: E.heegner_sha_an(-7) is E.heegner_sha_an(-7)          # long
True
```

Lower precision:

```
sage: E.heegner_sha_an(-7,10)                               # long
1.0
```

Checking that the cache works for any precision:

```
sage: E.heegner_sha_an(-7,10) is E.heegner_sha_an(-7,10)    # long
True
```

A rank 1 curve with nontrivial Sha over the quadratic imaginary field K; however, there is no Sha for E over QQ or for the quadratic twist of E:

```
sage: E = EllipticCurve('37a')
sage: E.heegner_sha_an(-40)                                 # long
4.00000000000000
sage: E.quadratic_twist(-40).sha().an()                     # long
1
sage: E.sha().an()                                          # long
1
```

A rank 2 curve:

```
sage: E = EllipticCurve('389a')                             # long
sage: E.heegner_sha_an(-7)                                  # long
1.00000000000000
```

If we remove the hypothesis that E(K) has rank 1 in Conjecture 2.3 in [Gross-Zagier, 1986, page 311], then that conjecture is false, as the following example shows:

```
sage: E = EllipticCurve('65a')                              # long
sage: E.heegner_sha_an(-56)                                 # long
```

```
1.00000000000000
sage: E.torsion_order()                                    # long
2
sage: E.tamagawa_product()                                 # long
1
sage: E.quadratic_twist(-56).rank()                        # long
2
```

height (*precision=None*)

Returns the real height of this elliptic curve. This is used in integral_points()

INPUT:

- `precision` - desired real precision of the result (default real precision if None)

EXAMPLES:

```
sage: E=EllipticCurve('5077a1')
sage: E.height()
17.4513334798896
sage: E.height(100)
17.451333479889612702508579399
sage: E=EllipticCurve([0,0,0,0,1])
sage: E.height()
1.38629436111989
sage: E=EllipticCurve([0,0,0,1,0])
sage: E.height()
7.45471994936400
```

height_pairing_matrix (*points=None, precision=None*)

Returns the height pairing matrix of the given points on this curve, which must be defined over $\mathbf{Q}$.

INPUT:

- points - either a list of points, which must be on this curve, or (default) None, in which case self.gens() will be used. precision - number of bits of precision of result (default: None, for default RealField precision)

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])
sage: E.height_pairing_matrix()
[0.0511114082399688]
```

For rank 0 curves, the result is a valid 0x0 matrix:

```
sage: EllipticCurve('11a').height_pairing_matrix()
[]
sage: E=EllipticCurve('5077a1')
sage: E.height_pairing_matrix([E.lift_x(x) for x in [-2,-7/4,1]], precision=100)
[  1.3685725053539301120518194471  -1.3095767070865761992624519454 -0.6348671578371559206447
[ -1.3095767070865761992624519454   2.7173593928122930896610589220   1.0998184305667292139777571432  0.6682051656519279350331
```

integral_model ()

Return a model of self which is integral at all primes.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1/216, -7/1296, 1/7776])
sage: F = E.global_integral_model(); F
Elliptic Curve defined by y^2 + y = x^3 - 7*x + 6 over Rational Field
sage: F == EllipticCurve('5077a1')
True
```

integral_points(*mw_base='auto', both_signs=False, verbose=False*)

Computes all integral points (up to sign) on this elliptic curve.

INPUT:

- `mw_base` - list of EllipticCurvePoint generating the Mordell-Weil group of E (default: 'auto' - calls self.gens())
- `both_signs` - True/False (default False): if True the output contains both P and -P, otherwise only one of each pair.
- `verbose` - True/False (default False): if True, some details of the computation are output

OUTPUT: A sorted list of all the integral points on E (up to sign unless both_signs is True)

Note: The complexity increases exponentially in the rank of curve E. The computation time (but not the output!) depends on the Mordell-Weil basis. If mw_base is given but is not a basis for the Mordell-Weil group (modulo torsion), integral points which are not in the subgroup generated by the given points will almost certainly not be listed.

EXAMPLES: A curve of rank 3 with no torsion points

```
sage: E=EllipticCurve([0,0,1,-7,6])
sage: P1=E.point((2,0)); P2=E.point((-1,3)); P3=E.point((4,6))
sage: a=E.integral_points([P1,P2,P3]); a
[(-3 : 0 : 1), (-2 : 3 : 1), (-1 : 3 : 1), (0 : 2 : 1), (1 : 0 : 1), (2 : 0 : 1), (3 : 3 : 1
```

```
sage: a = E.integral_points([P1,P2,P3], verbose=True)
Using mw_basis  [(2 : 0 : 1), (3 : -4 : 1), (8 : -22 : 1)]
e1,e2,e3:  -3.0124303725933... 1.0658205476962... 1.94660982489710
Minimal eigenvalue of height pairing matrix:  0.63792081458500...
x-coords of points on compact component with  -3 <=x<= 1
[-3, -2, -1, 0, 1]
x-coords of points on non-compact component with  2 <=x<= 6
[2, 3, 4]
starting search of remaining points using coefficient bound  5
x-coords of extra integral points:
[2, 3, 4, 8, 11, 14, 21, 37, 52, 93, 342, 406, 816]
Total number of integral points: 18
```

It is not necessary to specify mw_base; if it is not provided, then the Mordell-Weil basis must be computed, which may take much longer.

```
sage: E=EllipticCurve([0,0,1,-7,6])
sage: a=E.integral_points(both_signs=True); a
[(-3 : -1 : 1), (-3 : 0 : 1), (-2 : -4 : 1), (-2 : 3 : 1), (-1 : -4 : 1), (-1 : 3 : 1), (0 :
```

An example with negative discriminant:

```
sage: EllipticCurve('900d1').integral_points()
[(-11 : 27 : 1), (-4 : 34 : 1), (4 : 18 : 1), (16 : 54 : 1)]
```

Another example with rank 5 and no torsion points

```
sage: E=EllipticCurve([-879984,319138704])
sage: P1=E.point((540,1188)); P2=E.point((576,1836))
sage: P3=E.point((468,3132)); P4=E.point((612,3132))
sage: P5=E.point((432,4428))
sage: a=E.integral_points([P1,P2,P3,P4,P5]); len(a)  # long time (400s!)
54
```

The bug reported on trac #4525 is now fixed:

```
sage: EllipticCurve('91b1').integral_points()
[(-1 : 3 : 1), (1 : 0 : 1), (3 : 4 : 1)]
```

```
sage: [len(e.integral_points(both_signs=False)) for e in cremona_curves([11..100])] # long t
[2, 0, 2, 3, 2, 1, 3, 0, 2, 4, 2, 4, 3, 0, 0, 1, 2, 1, 2, 0, 2, 1, 0, 1, 3, 3, 1, 1, 4, 2, 3
```

The bug reported at #4897 is now fixed:

```
sage: [P[0] for P in EllipticCurve([0,0,0,-468,2592]).integral_points()]
[-24, -18, -14, -6, -3, 4, 6, 18, 21, 24, 36, 46, 102, 168, 186, 381, 1476, 2034, 67246]
```

Note: This function uses the algorithm given in [Co1].

REFERENCES:

•[Co1] Cohen H., Number Theory Vol I: Tools and Diophantine Equations GTM 239, Springer 2007

AUTHORS:

•Michael Mardaus (2008-07)

•Tobias Nagel (2008-07)

•John Cremona (2008-07)

integral_short_weierstrass_model()

Return a model of the form $y^2 = x^3 + a * x + b$ for this curve with $a, b \in \mathbf{Z}$.

EXAMPLES:

```
sage: E = EllipticCurve('17a1')
sage: E.integral_short_weierstrass_model()
Elliptic Curve defined by y^2  = x^3 - 11*x - 890 over Rational Field
```

integral_weierstrass_model()

Return a model of the form $y^2 = x^3 + a * x + b$ for this curve with $a, b \in \mathbf{Z}$.

Note that this function is deprecated, and that you should use integral_short_weierstrass_model instead as this will be disappearing in the near future.

EXAMPLES:

```
sage: E = EllipticCurve('17a1')
sage: E.integral_weierstrass_model() #random
doctest:1: DeprecationWarning: integral_weierstrass_model is deprecated, use integral_short_
Elliptic Curve defined by y^2  = x^3 - 11*x - 890 over Rational Field
```

integral_x_coords_in_interval(*xmin, xmax*)

Returns the set of integers x with $xmin \leq x \leq xmax$ which are x-coordinates of points on this curve.

is_global_integral_model()

Return true iff self is integral at all primes.

EXAMPLES:

```
sage: E=EllipticCurve([1/2,1/5,1/5,1/5,1/5])
sage: E.is_global_integral_model()
False
sage: Emin=E.global_integral_model()
sage: Emin.is_global_integral_model()
True
```

is_good(*p, check=True*)

Return True if p is a prime of good reduction for E.

INPUT:

•p - a prime

OUTPUT: bool

EXAMPLES:

```
sage: e = EllipticCurve('11a')
sage: e.is_good(-8)
...
ValueError: p must be prime
sage: e.is_good(-8, check=False)
True
```

is_integral()

Returns True if this elliptic curve has integral coefficients (in Z)

EXAMPLES:

```
sage: E=EllipticCurve(QQ,[1,1]); E
Elliptic Curve defined by y^2 = x^3 + x + 1 over Rational Field
sage: E.is_integral()
True
sage: E2=E.change_weierstrass_model(2,0,0,0); E2
Elliptic Curve defined by y^2 = x^3 + 1/16*x + 1/64 over Rational Field
sage: E2.is_integral()
False
```

is_irreducible(p)

Return True if the mod p representation is irreducible.

EXAMPLES:

```
sage: e = EllipticCurve('37b')
sage: e.is_irreducible(2)
True
sage: e.is_irreducible(3)
False
sage: e.is_reducible(2)
False
sage: e.is_reducible(3)
True
```

is_local_integral_model(*p)

Tests if self is integral at the prime p, or at all the primes if p is a list or tuple of primes

EXAMPLES:

```
sage: E=EllipticCurve([1/2,1/5,1/5,1/5,1/5])
sage: [E.is_local_integral_model(p) for p in (2,3,5)]
[False, True, False]
sage: E.is_local_integral_model(2,3,5)
False
sage: Eint2=E.local_integral_model(2)
sage: Eint2.is_local_integral_model(2)
True
```

is_minimal()

Return True iff this elliptic curve is a reduced minimal model.

The unique minimal Weierstrass equation for this elliptic curve. This is the model with minimal discriminant and $a_1, a_2, a_3 \in \{0, \pm 1\}$.

TO DO: This is not very efficient since it just computes the minimal model and compares. A better implementation using the Kraus conditions would be preferable.

EXAMPLES:

```
sage: E=EllipticCurve([10,100,1000,10000,1000000])
sage: E.is_minimal()
False
```

```
sage: E=E.minimal_model()
sage: E.is_minimal()
True
```

is_ordinary(*p, ell=None*)

Return True precisely when the mod-p representation attached to this elliptic curve is ordinary at ell.

INPUT:

- p - a prime ell - a prime (default: p)

OUTPUT: bool

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.is_ordinary(37)
True
sage: E=EllipticCurve('32a1')
sage: E.is_ordinary(2)
False
sage: [p for p in prime_range(50) if E.is_ordinary(p)]
[5, 13, 17, 29, 37, 41]
```

is_p_integral(*p*)

Returns True if this elliptic curve has p-integral coefficients.

INPUT:

- p - a prime integer

EXAMPLES:

```
sage: E=EllipticCurve(QQ,[1,1]); E
Elliptic Curve defined by y^2 = x^3 + x + 1 over Rational Field
sage: E.is_p_integral(2)
True
sage: E2=E.change_weierstrass_model(2,0,0,0); E2
Elliptic Curve defined by y^2 = x^3 + 1/16*x + 1/64 over Rational Field
sage: E2.is_p_integral(2)
False
sage: E2.is_p_integral(3)
True
```

is_p_minimal(*p*)

Tests if curve is p-minimal at a given prime p.

INPUT: p - a primeOUTPUT: True - if curve is p-minimal

- False - if curve isn't p-minimal

EXAMPLES:

```
sage: E = EllipticCurve('441a2')
sage: E.is_p_minimal(7)
True

sage: E = EllipticCurve([0,0,0,0,(2*5*11)**10])
sage: [E.is_p_minimal(p) for p in prime_range(2,24)]
[False, True, False, True, False, True, True, True, True]
```

is_reducible(*p*)

Return True if the mod-p representation attached to E is reducible.

INPUT:

- p - a prime number

Note: The answer is cached.

EXAMPLES:

```
sage: E = EllipticCurve('121a'); E
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 30*x - 76 over Rational Field
sage: E.is_reducible(7)
False
sage: E.is_reducible(11)
True
sage: EllipticCurve('11a').is_reducible(5)
True
sage: e = EllipticCurve('11a2')
sage: e.is_reducible(5)
True
sage: e.torsion_order()
1
```

is_semistable()

Return True iff this elliptic curve is semi-stable at all primes.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.is_semistable()
True
sage: E=EllipticCurve('90a1')
sage: E.is_semistable()
False
```

is_supersingular(*p, ell=None*)

Return True precisely when p is a prime of good reduction and the mod-p representation attached to this elliptic curve is supersingular at ell.

INPUT:

- p - a prime ell - a prime (default: p)

OUTPUT: bool

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.is_supersingular(37)
False
sage: E=EllipticCurve('32a1')
sage: E.is_supersingular(2)
False
sage: E.is_supersingular(7)
True
sage: [p for p in prime_range(50) if E.is_supersingular(p)]
[3, 7, 11, 19, 23, 31, 43, 47]
```

is_surjective(*p, A=1000*)

Return True if the mod-p representation attached to E is surjective, False if it is not, or None if we were unable to determine whether it is or not.

Note: The answer is cached.

INPUT:

- p - int (a prime number)
- A - int (a bound on the number of a_p to use)

OUTPUT:

A 2-tuple:

- surjective or (probably) not
- information about what it is if not surjective

EXAMPLES:

```
sage: e = EllipticCurve('37b')
sage: e.is_surjective(2)
(True, None)
sage: e.is_surjective(3)
(False, '3-torsion')
```

REMARKS:

1. If $p = 5$ then the mod-p representation is surjective if and only if the p-adic representation is surjective. When $p = 2, 3$ there are counterexamples. See a very recent paper of Elkies for more details when $p = 3$.
2. When p = 3 this function always gives the correct result irregardless of A, since it explicitly determines the p-division polynomial.

isogeny_class(*algorithm='mwrank', verbose=False*)

Return all curves over $\mathbf{Q}$ in the isogeny class of this elliptic curve.

WARNING: The result is emph{not} provably correct, in the sense that when the numbers are huge isogenies could be missed because of precision issues.

INPUT:

- `verbose` – bool (default: False)
- `algorithm` - string:
  - "mwrank" - (default) use the mwrank C++ library
  - "database" - use the Cremona database (only works if curve is isomorphic to a curve in the database)

OUTPUT: Returns the sorted list of the curves isogenous to self. If algorithm is "mwrank", also returns the isogeny matrix (otherwise returns None as second return value).

Note: The ordering depends on which algorithm is used.

EXAMPLES:

```
sage: I, A = EllipticCurve('37b').isogeny_class('mwrank')
sage: I   # randomly ordered
[Elliptic Curve defined by y^2 + y = x^3 + x^2 - 23*x - 50 over Rational Field,
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 1873*x - 31833 over Rational Field,
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 3*x +1 over Rational Field]
sage: A
[0 3 3]
[3 0 0]
[3 0 0]

sage: I, _ = EllipticCurve('37b').isogeny_class('database'); I
[Elliptic Curve defined by y^2 + y = x^3 + x^2 - 1873*x - 31833 over Rational Field,
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 23*x - 50 over Rational Field,
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 3*x + 1 over Rational Field]
```

This is an example of a curve with a 37-isogeny:

```
sage: E = EllipticCurve([1,1,1,-8,6])
sage: E.isogeny_class ()
([Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 8*x + 6 over Rational Field,
  Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 208083*x - 36621194 over Rational Fi
 [ 0 37]
 [37  0])
```

This curve had numerous 2-isogenies:

```
sage: e=EllipticCurve([1,0,0,-39,90])
sage: e.isogeny_class ()
([Elliptic Curve defined by y^2 + x*y = x^3 - 39*x + 90 over Rational Field,
  Elliptic Curve defined by y^2 + x*y = x^3 - 4*x - 1 over Rational Field,
  Elliptic Curve defined by y^2 + x*y = x^3 + x over Rational Field,
  Elliptic Curve defined by y^2 + x*y = x^3 - 49*x - 136 over Rational Field,
  Elliptic Curve defined by y^2 + x*y = x^3 - 34*x - 217 over Rational Field,
  Elliptic Curve defined by y^2 + x*y = x^3 - 784*x - 8515 over Rational Field],
 [0 2 0 0 0 0]
 [2 0 2 2 0 0]
 [0 2 0 0 0 0]
 [0 2 0 0 2 2]
 [0 0 0 2 0 0]
 [0 0 0 2 0 0])
```

See http://math.harvard.edu/~elkies/nature.html for more interesting examples of isogeny structures.

isogeny_graph()

Returns a graph representing the isogeny class of this elliptic curve, where the vertices are isogenous curves over $\mathbf{Q}$ and the edges are prime degree isogenies

Warning: The result is *not* provably correct, in the sense that when the numbers are huge isogenies could be missed because of precision issues.

EXAMPLES:

```
sage: LL = []
sage: for e in cremona_optimal_curves(range(1, 38)):
...     G = e.isogeny_graph()
...     already = False
...     for H in LL:
...         if G.is_isomorphic(H):
...             already = True
...             break
...     if not already:
...         LL.append(G)
...
sage: graphs_list.show_graphs(LL)

sage: E = EllipticCurve('195a')
sage: G = E.isogeny_graph()
sage: for v in G: print v, G.get_vertex(v)
...
0 Elliptic Curve defined by y^2 + x*y  = x^3 - 110*x + 435 over Rational Field
1 Elliptic Curve defined by y^2 + x*y  = x^3 - 115*x + 392 over Rational Field
2 Elliptic Curve defined by y^2 + x*y  = x^3 + 210*x + 2277 over Rational Field
3 Elliptic Curve defined by y^2 + x*y  = x^3 - 520*x - 4225 over Rational Field
4 Elliptic Curve defined by y^2 + x*y  = x^3 + 605*x - 19750 over Rational Field
5 Elliptic Curve defined by y^2 + x*y  = x^3 - 8125*x - 282568 over Rational Field
6 Elliptic Curve defined by y^2 + x*y  = x^3 - 7930*x - 296725 over Rational Field
7 Elliptic Curve defined by y^2 + x*y  = x^3 - 130000*x - 18051943 over Rational Field
sage: G.plot(edge_labels=True)
```

kodaira_symbol(p)

Local Kodaira type of the elliptic curve at p.

INPUT:

- p, an integral prime

OUTPUT:

•the Kodaira type of this elliptic curve at p, as a KodairaSymbol.

EXAMPLES:

```
sage: E = EllipticCurve('124a')
sage: E.kodaira_type(2)
IV
```

kodaira_type(*p*)

Local Kodaira type of the elliptic curve at p.

INPUT:

•p, an integral prime

OUTPUT:

•the Kodaira type of this elliptic curve at p, as a KodairaSymbol.

EXAMPLES:

```
sage: E = EllipticCurve('124a')
sage: E.kodaira_type(2)
IV
```

kodaira_type_old(*p*)

Local Kodaira type of the elliptic curve at p.

INPUT:

•p, an integral prime

OUTPUT:

•the kodaira type of this elliptic curve at p, as a KodairaSymbol.

EXAMPLES:

```
sage: E = EllipticCurve('124a')
sage: E.kodaira_type_old(2)
IV
```

label(*space=False*)

Return the Cremona label associated to (the minimal model) of this curve, if it is known. If not, raise a RuntimeError exception.

EXAMPLES:

```
sage: E=EllipticCurve('389a1')
sage: E.cremona_label()
'389a1'
```

The default database only contains conductors up to 10000, so any curve with conductor greater than that will cause an error to be raised. The optional package 'database_cremona_ellcurve-20071019' contains more curves, with conductors up to 130000.

```
sage: E = EllipticCurve([1, -1, 0, -79, 289])
sage: E.conductor()
234446
sage: E.cremona_label()
...
RuntimeError: Cremona label not known for Elliptic Curve defined by y^2 + x*y = x^3 - x^2 -
```

lll_reduce(*points, height_matrix=None*)

Returns an LLL-reduced basis from a given basis, with transform matrix.

INPUT:

•`points` - a list of points on this elliptic curve, which should be independent.

•`height_matrix` - the height-pairing matrix of the points, or None. If None, it will be computed.

OUTPUT: A tuple (newpoints,U) where U is a unimodular integer matrix, new_points is the transform of points by U, such that new_points has LLL-reduced height pairing matrix

Note: If the input points are not independent, the output depends on the undocumented behaviour of pari's `qflllgram()` function when applied to a gram matrix which is not positive definite.

EXAMPLE:

```
sage: E = EllipticCurve([0, 1, 1, -2, 42])
sage: Pi = E.gens(); Pi
[(-4 : 1 : 1), (-3 : 5 : 1), (-11/4 : 43/8 : 1), (-2 : 6 : 1)]
sage: Qi, U = E.lll_reduce(Pi)
sage: Qi
[(0 : 6 : 1), (1 : -7 : 1), (-4 : 1 : 1), (-3 : 5 : 1)]
sage: U.det()
1
sage: E.regulator_of_points(Pi)
4.59088036960574
sage: E.regulator_of_points(Qi)
4.59088036960574

sage: E = EllipticCurve([1,0,1,-120039822036992245303534619191166796374,504224992484910670010
sage: xi = [2005024558054813068,            -4690836759490453344,            4700156326649800
sage: points = [E.lift_x(x) for x in xi]
sage: newpoints, U = E.lll_reduce(points) # long time (2m)
sage: [P[0] for P in newpoints]           # long time
[6823803569166584943, 5949539878899294213, 2005024558054813068, 5864879778877955778, 2395526
```

local_integral_model(*p*)

Return a model of self which is integral at the prime p.

EXAMPLES:

```
sage: E=EllipticCurve([0, 0, 1/216, -7/1296, 1/7776])
sage: E.local_integral_model(2)
Elliptic Curve defined by y^2 + 1/27*y = x^3 - 7/81*x + 2/243 over Rational Field
sage: E.local_integral_model(3)
Elliptic Curve defined by y^2 + 1/8*y = x^3 - 7/16*x + 3/32 over Rational Field
sage: E.local_integral_model(2).local_integral_model(3) == EllipticCurve('5077a1')
True
```

lseries()

Returns the L-series of this elliptic curve.

Further documentation is available for the functions which apply to the L-series.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.lseries()
Complex L-series of the Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

manin_constant()

Return the Manin constant of this elliptic curve.

OUTPUT:

an integer

This function only works if the curve is in the installed Cremona database. Sage includes by default a small databases; for the full database you have to install an optional package.

Warning: The result is *not* provably correct, in the sense that when the numbers are huge isogenies could be missed because of precision issues.

EXAMPLES:

```
sage: EllipticCurve('11a1').manin_constant()
1
sage: EllipticCurve('11a2').manin_constant()
5
sage: EllipticCurve('11a3').manin_constant()
5
```

Check that it works even if the curve is non-minimal:

```
sage: EllipticCurve('11a1').short_weierstrass_model().manin_constant()
1
```

An example where the isogeny class is large, so it's not completely trivial to find the minimal degree of an isogeny to the optimal curve:

```
sage: [EllipticCurve('210b%s'%i).manin_constant() for i in [1..8]]
[1, 2, 3, 4, 4, 6, 12, 12]
```

Make sure the special case 990h is treated correctly, i.e., that 990h3 has Manin constant 1:

```
sage: [EllipticCurve('990h%s'%i).manin_constant() for i in [1..4]]
[3, 6, 1, 2]
```

This plots helps you see that the above Manin constants are right. Note that the vertex labels are 0-based unlinke the Cremona isogeny labels:

```
sage: EllipticCurve('210b1').isogeny_graph().plot(edge_labels=True)
```

matrix_of_frobenius (*p, prec=20, check=False, check_hypotheses=True, algorithm='auto'*)
See the parameters and documentation for padic_E2.

minimal_model()
Return the unique minimal Weierstrass equation for this elliptic curve. This is the model with minimal discriminant and $a_1, a_2, a_3 \in \{0, \pm 1\}$.

EXAMPLES:

```
sage: E=EllipticCurve([10,100,1000,10000,1000000])
sage: E.minimal_model()
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + x + 1 over Rational Field
```

minimal_quadratic_twist()
Determines a quadratic twist with minimal conductor. Returns a global minimal model of the twist and the fundamental discriminant of the quadratic field over which they are isomorphic.

Note: If there is more than one curve with minimal conductor, the one returned is the one with smallest label (if in the database), or the one with minimal a-invariant list (otherwise).

EXAMPLES:

```
sage: E = EllipticCurve('121d1')
sage: E.minimal_quadratic_twist()
(Elliptic Curve defined by y^2 + y = x^3 - x^2 over Rational Field, -11)
sage: Et, D = EllipticCurve('32a1').minimal_quadratic_twist()
sage: D
1

sage: E = EllipticCurve('11a1')
sage: Et, D = E.quadratic_twist(-24).minimal_quadratic_twist()
sage: E == Et
True
sage: D
-24
```

```
sage: E = EllipticCurve([0,0,0,0,1000])
sage: E.minimal_quadratic_twist()
(Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field, 40)
sage: E = EllipticCurve([0,0,0,1600,0])
sage: E.minimal_quadratic_twist()
(Elliptic Curve defined by y^2 = x^3 + 4*x over Rational Field, 5)
```

mod5family()

Return the family of all elliptic curves with the same mod-5 representation as self.

EXAMPLES:

```
sage: E=EllipticCurve('32a1')
sage: E.mod5family()
Elliptic Curve defined by y^2  = x^3 + 4*x over Fraction Field of Univariate Polynomial Ring
```

modular_degree(*algorithm='sympow'*)

Return the modular degree of this elliptic curve.

The result is cached. Subsequent calls, even with a different algorithm, just returned the cached result.

INPUT:

- `algorithm` - string:
- `'sympow'` - (default) use Mark Watkin's (newer) C program sympow
- `'magma'` - requires that MAGMA be installed (also implemented by Mark Watkins)

Note: On 64-bit computers ec does not work, so Sage uses sympow even if ec is selected on a 64-bit computer.

The correctness of this function when called with algorithm "sympow" is subject to the following three hypothesis:

- Manin's conjecture: the Manin constant is 1
- Steven's conjecture: the $X_1(N)$-optimal quotient is the curve with minimal Faltings height. (This is proved in most cases.)
- The modular degree fits in a machine double, so it better be less than about 50-some bits. (If you use sympow this constraint does not apply.)

Moreover for all algorithms, computing a certain value of an L-function "uses a heuristic method that discerns when the real-number approximation to the modular degree is within epsilon [=0.01 for algorithm="sympow"] of the same integer for 3 consecutive trials (which occur maybe every 25000 coefficients or so). Probably it could just round at some point. For rigour, you would need to bound the tail by assuming (essentially) that all the a_n are as large as possible, but in practise they exhibit significant (square root) cancellation. One difficulty is that it doesn't do the sum in 1-2-3-4 order; it uses 1-2-4-8–3-6-12-24-9-18- (Euler product style) instead, and so you have to guess ahead of time at what point to curtail this expansion." (Quote from an email of Mark Watkins.)

Note: If the curve is loaded from the large Cremona database, then the modular degree is taken from the database.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: E.modular_degree()
1
sage: E = EllipticCurve('5077a')
sage: E.modular_degree()
1984
sage: factor(1984)
2^6 * 31
```

```
sage: EllipticCurve([0, 0, 1, -7, 6]).modular_degree()
1984
sage: EllipticCurve([0, 0, 1, -7, 6]).modular_degree(algorithm='sympow')
1984
sage: EllipticCurve([0, 0, 1, -7, 6]).modular_degree(algorithm='magma')  # optional - magma
1984
```

We compute the modular degree of the curve with rank 4 having smallest (known) conductor:

```
sage: E = EllipticCurve([1, -1, 0, -79, 289])
sage: factor(E.conductor())  # conductor is 234446
2 * 117223
sage: factor(E.modular_degree())
2^7 * 2617
```

modular_form()

Return the cuspidal modular form associated to this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: f = E.modular_form()
sage: f
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + O(q^6)
```

If you need to see more terms in the q-expansion:

```
sage: f.q_expansion(20)
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + 6*q^6 - q^7 + 6*q^9 + 4*q^10 - 5*q^11 - 6*q^12 - 2*q^13
```

Note: If you just want the q-expansion, use `q_expansion()`.

modular_parametrization()

Computes and returns the modular parametrization of this elliptic curve.

The curve is converted to a minimal model.

OUTPUT: A list of two Laurent series [X(x),Y(x)] of degrees -2, -3 respectively, which satisfy the equation of the (minimal model of the) elliptic curve. There are modular functions on $\Gamma_0(N)$ where N is the conductor.

X.deriv()/(2*Y+a1*X+a3) should equal f(q)dq/q where f is self.q_expansion().

EXAMPLES:

```
sage: E=EllipticCurve('389a1')
sage: X,Y=E.modular_parametrization()
sage: X
q^-2 + 2*q^-1 + 4 + 7*q + 13*q^2 + 18*q^3 + 31*q^4 + 49*q^5 + 74*q^6 + 111*q^7 + 173*q^8 + 2
sage: Y
-q^-3 - 3*q^-2 - 8*q^-1 - 17 - 33*q - 61*q^2 - 110*q^3 - 186*q^4 - 320*q^5 - 528*q^6 - 861*q
```

The following should give 0, but only approximately:

```
sage: q = X.parent().gen()
sage: E.defining_polynomial()(X,Y,1) + O(q^11) == 0
True
```

Note that below we have to change variable from x to q

```
sage: a1,_,a3,_,_=E.a_invariants()
sage: f=E.q_expansion(17)
sage: q=f.parent().gen()
sage: f/q == (X.derivative()/(2*Y+a1*X+a3))
True
```

modular_symbol (*sign=1, use_eclib=False, normalize='L_ratio'*)

Return the modular symbol associated to this elliptic curve, with given sign and base ring. This is the map that sends r/s to a fixed multiple of the integral of $2\pi i f(z)dz$ from ∞ to r/s, normalized so that all values of this map take values in $\mathbf{Q}$.

The normalization is such that for sign +1, the value at the cusp 0 is equal to the quotient of $L(E,1)$ by the least positive period of E (unlike in `L_ratio` of `lseries()`, where the value is also divided by the number of connected components of $E(\mathbf{R})$). In particular the modular symbol depends on E and not only the isogeny class of E.

INPUT:

- `sign` - -1, or 1
- `base_ring` - a ring
- `normalize` - (default: True); if True, the modular symbol is correctly normalized (up to possibly a factor of -1 or 2). If False, the modular symbol is almost certainly not correctly normalized, i.e., all values will be a fixed scalar multiple of what they should be. But the initial computation of the modular symbol is much faster, though evaluation of it after computing it won't be any faster.
- `use_eclib` - (default: False); if True the computation is done with John Cremona's implementation of modular symbols in `eclib`. But this is only possible for `sign` = +1.
- `normalize` - (default: 'L_ratio'); either 'L_ratio', 'period', or 'none'; For 'L_ratio', the modular symbol is correctly normalized as explained above by comparing it to `L_ratio` for the curve and some small twists. The normalization 'period' is only available if `use_eclib=False`. It uses the `integral_period_map` for modular symbols and is known to be equal to the above normalization up to the sign and a possible power of 2. For 'none', the modular symbol is almost certainly not correctly normalized, i.e. all values will be a fixed scalar multiple of what they should be. But the initial computation of the modular symbol is much faster if `use_eclib=False`, though evaluation of it after computing it won't be any faster.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: M=E.modular_symbol(); M
Modular symbol with sign 1 over Rational Field attached to Elliptic Curve defined by y^2 + y
sage: M(1/2)
0
sage: M(1/5)
1

sage: E=EllipticCurve('121b1')
sage: M=E.modular_symbol()
sage: M(1/7)
2

sage: E=EllipticCurve('11a1')
sage: E.modular_symbol()(0)
1/5
sage: E=EllipticCurve('11a2')
sage: E.modular_symbol()(0)
1
sage: E=EllipticCurve('11a3')
sage: E.modular_symbol()(0)
1/25

sage: E=EllipticCurve('11a2')
sage: E.modular_symbol(use_eclib=True, normalize='L_ratio')(0)
1
sage: E.modular_symbol(use_eclib=True, normalize='none')(0)
1/5
```

```
sage: E.modular_symbol(use_eclib=True, normalize='period')(0)
...
ValueError: no normalization 'period' known for modular symbols using John Cremona's eclib
sage: E.modular_symbol(use_eclib=False, normalize='L_ratio')(0)
1
sage: E.modular_symbol(use_eclib=False, normalize='none')(0)
1
sage: E.modular_symbol(use_eclib=False, normalize='period')(0)
1

sage: E=EllipticCurve('11a3')
sage: E.modular_symbol(use_eclib=True, normalize='L_ratio')(0)
1/25
sage: E.modular_symbol(use_eclib=True, normalize='none')(0)
1/5
sage: E.modular_symbol(use_eclib=True, normalize='period')(0)
...
ValueError: no normalization 'period' known for modular symbols using John Cremona's eclib
sage: E.modular_symbol(use_eclib=False, normalize='L_ratio')(0)
1/25
sage: E.modular_symbol(use_eclib=False, normalize='none')(0)
1
sage: E.modular_symbol(use_eclib=False, normalize='period')(0)
1/25
```

modular_symbol_space(*sign=1, base_ring=Rational Field, bound=None*)

Return the space of cuspidal modular symbols associated to this elliptic curve, with given sign and base ring.

INPUT:

- `sign` - 0, -1, or 1
- `base_ring` - a ring

EXAMPLES:

```
sage: f = EllipticCurve('37b')
sage: f.modular_symbol_space()
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0(
sage: f.modular_symbol_space(-1)
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 for Gamma_0(
sage: f.modular_symbol_space(0, bound=3)
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 5 for Gamma_0(
```

Note: If you just want the q-expansion, use `q_expansion()`.

mwrank(*options=''*)

Run Cremona's mwrank program on this elliptic curve and return the result as a string.

INPUT:

- `options` (string) – run-time options passed when starting mwrank. The format is as follows (see below for examples of usage):
  - `-v n` (verbosity level) sets verbosity to n (default=1)
  - `-o` (PARI/GP style output flag) turns ON extra PARI/GP short output (default is OFF)
  - `-p n` (precision) sets precision to n decimals (default=15)
  - `-b n` (quartic bound) bound on quartic point search (default=10)
  - `-x n` (n_aux) number of aux primes used for sieving (default=6)
  - `-l` (generator list flag) turns ON listing of points (default ON unless v=0)
  - `-s` (selmer_only flag) if set, computes Selmer rank only (default: not set)

– -d (skip_2nd_descent flag) if set, skips the second descent for curves with 2-torsion (default: not set)

– -S n (sat_bd) upper bound on saturation primes (default=100, -1 for automatic)

OUTPUT:

- string - output of mwrank on this curve

Note: The output is a raw string and completely illegible using automatic display, so it is recommended to use print for legible output.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.mwrank() #random
...
sage: print E.mwrank()
Curve [0,0,1,-1,0] :        Basic pair: I=48, J=-432
disc=255744
...
Generator 1 is [0:-1:1]; height 0.05111...

Regulator = 0.05111...

The rank and full Mordell-Weil basis have been determined unconditionally.
...
```

Options to mwrank can be passed:

```
sage: E = EllipticCurve([0,0,0,877,0])
```

Run mwrank with 'verbose' flag set to 0 but list generators if found

```
sage: print E.mwrank('-v0 -l')
Curve [0,0,0,877,0] :   0 <= rank <= 1
Regulator = 1
```

Run mwrank again, this time with a higher bound for point searching on homogeneous spaces:

```
sage: print E.mwrank('-v0 -l -b11')
Curve [0,0,0,877,0] :   Rank = 1
Generator 1 is [29604565304828237474403861024284371796799791624792913256602210:-25625626798
Regulator = 95.980371987964
```

mwrank_curve(*verbose=False*)

Construct an mwrank_EllipticCurve from this elliptic curve

The resulting mwrank_EllipticCurve has available methods from John Cremona's eclib library.

EXAMPLES:

```
sage: E=EllipticCurve('11a1')
sage: EE=E.mwrank_curve()
sage: EE
y^2+ y = x^3 - x^2 - 10*x - 20
sage: type(EE)
<class 'sage.libs.mwrank.interface.mwrank_EllipticCurve'>
sage: EE.isogeny_class()
([[0, -1, 1, -10, -20], [0, -1, 1, -7820, -263580], [0, -1, 1, 0, 0]],
[[0, 5, 5], [5, 0, 0], [5, 0, 0]])
```

newform()

Same as self.modular_form().

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.newform()
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + O(q^6)
sage: E.newform() == E.modular_form()
True
```

ngens (*proof=None*)

Return the number of generators of this elliptic curve.

Note: See :meth:'.gens' for further documentation. The function `ngens()` calls `gens()` if not already done, but only with default parameters. Better results may be obtained by calling `mwrank()` with carefully chosen parameters.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.ngens()
1
```

TO DO: This example should not cause a run-time error.

```
sage: E=EllipticCurve([0,0,0,877,0])
sage: # E.ngens()  ######## causes run-time error

sage: print E.mwrank('-v0 -b12 -l')
Curve [0,0,0,877,0] :   Rank = 1
Generator 1 is [29604565304828237474403861024284371796799791624792913256602210:-256256267988
Regulator = 95.980...
```

non_surjective (*A=1000*)

Returns a list of primes p such that the mod-p representation $\rho_{E,p}$ *might* not be surjective (this list usually contains 2, because of shortcomings of the algorithm). If p is not in the returned list, then rho_E,p is provably surjective (see A. Cojocaru's paper). If the curve has CM then infinitely many representations are not surjective, so we simply return the sequence [(0,"cm")] and do no further computation.

INPUT:

- `A` - an integer

OUTPUT:

- `list` - if curve has CM, returns [(0,"cm")]. Otherwise, returns a list of primes where mod-p representation is very likely not surjective. At any prime not in this list, the representation is definitely surjective.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -38, 90])  # 361A
sage: E.non_surjective()   # CM curve
[(0, 'cm')]

sage: E = EllipticCurve([0, -1, 1, 0, 0]) # X_1(11)
sage: E.non_surjective()
[(5, '5-torsion')]

sage: E = EllipticCurve([0, 0, 1, -1, 0]) # 37A
sage: E.non_surjective()
[]

sage: E = EllipticCurve([0,-1,1,-2,-1])   # 141C
sage: E.non_surjective()
[(13, [1])]
```

ALGORITHM: When p=3 use division polynomials. For 5 = p = B, where B is Cojocaru's bound, use the results in Section 2 of Serre's inventiones paper"Sur Les Representations Modulaires Deg Degre 2 de Galqbar Over Q."

optimal_curve()

Given an elliptic curve that is in the installed Cremona database, return the optimal curve isogenous to it.

EXAMPLES:

The following curve is not optimal:

```
sage: E = EllipticCurve('11a2'); E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 7820*x - 263580 over Rational Field
sage: E.optimal_curve()
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: E.optimal_curve().cremona_label()
'11a1'
```

Note that 990h is the special case where the optimal curve isn't the first in the Cremona labeling:

```
sage: E = EllipticCurve('990h4'); E
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 + 6112*x - 41533 over Rational Field
sage: F = E.optimal_curve(); F
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 - 1568*x - 4669 over Rational Field
sage: F.cremona_label()
'990h3'
sage: EllipticCurve('990a1').optimal_curve().cremona_label()    # a isn't h.
'990a1'
```

If the input curve is optimal, this function returns that curve (not just a copy of it or a curve isomorphic to it!):

```
sage: E = EllipticCurve('37a1')
sage: E.optimal_curve() is E
True
```

Also, if this curve is optimal but not given by a minimal model, this curve will still be returned, so this function need not return a minimal model in general.

```
sage: F = E.short_weierstrass_model(); F
Elliptic Curve defined by y^2  = x^3 - 16*x + 16 over Rational Field
sage: F.optimal_curve()
Elliptic Curve defined by y^2  = x^3 - 16*x + 16 over Rational Field
```

ordinary_primes(*B*)

Return a list of all ordinary primes for this elliptic curve up to and possibly including B.

EXAMPLES:

```
sage: e = EllipticCurve('11a')
sage: e.aplist(20)
[-2, -1, 1, -2, 1, 4, -2, 0]
sage: e.ordinary_primes(97)
[3, 5, 7, 11, 13, 17, 23, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
sage: e = EllipticCurve('49a')
sage: e.aplist(20)
[1, 0, 0, 0, 4, 0, 0, 0]
sage: e.supersingular_primes(97)
[3, 5, 13, 17, 19, 31, 41, 47, 59, 61, 73, 83, 89, 97]
sage: e.ordinary_primes(97)
[2, 11, 23, 29, 37, 43, 53, 67, 71, 79]
sage: e.ordinary_primes(3)
[2]
sage: e.ordinary_primes(2)
```

```
[2]
sage: e.ordinary_primes(1)
[]
```

p_isogenous_curves (*p=None*)

Return a list of pairs (p, L) where p is a prime and L is a list of the elliptic curves over $\mathbf{Q}$ that are p-isogenous to this elliptic curve.

INPUT:

- p - prime or None (default: None); if a prime, returns a list of the p-isogenous curves. Otherwise, returns a list of all prime-degree isogenous curves sorted by isogeny degree.

This is implemented using Cremona's GP script allisog.gp.

EXAMPLES:

```
sage: E = EllipticCurve([0,-1,0,-24649,1355209])
sage: E.p_isogenous_curves()
[(2, [Elliptic Curve defined by y^2  = x^3 - x^2 - 91809*x - 9215775 over Rational Field, El
```

The isogeny class of the curve 11a2 has three curves in it. But p_isogenous_curves only returns one curves, since there is only one curve 5-isogenous to 11a2.

```
sage: E = EllipticCurve('11a2')
sage: E.p_isogenous_curves()
[(5, [Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field])]
sage: E.p_isogenous_curves(5)
[Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field]
sage: E.p_isogenous_curves(3)
[]
```

In contrast, the curve 11a1 admits two 5-isogenies:

```
sage: E = EllipticCurve('11a1')
sage: E.p_isogenous_curves(5)
[Elliptic Curve defined by y^2 + y = x^3 - x^2 - 7820*x - 263580 over Rational Field,
 Elliptic Curve defined by y^2 + y = x^3 - x^2 over Rational Field]
```

padic_E2 (*p, prec=20, check=False, check_hypotheses=True, algorithm='auto'*)

Returns the value of the p-adic modular form $E2$ for (E, ω) where ω is the usual invariant differential $dx/(2y + a_1x + a_3)$.

INPUT:

- p - prime (= 5) for which E is good and ordinary
- prec - (relative) p-adic precision (= 1) for result
- check - boolean, whether to perform a consistency check. This will slow down the computation by a constant factor 2. (The consistency check is to compute the whole matrix of frobenius on Monsky-Washnitzer cohomology, and verify that its trace is correct to the specified precision. Otherwise, the trace is used to compute one column from the other one (possibly after a change of basis).)
- check_hypotheses - boolean, whether to check that this is a curve for which the p-adic sigma function makes sense
- algorithm - one of "standard", "sqrtp", or "auto". This selects which version of Kedlaya's algorithm is used. The "standard" one is the one described in Kedlaya's paper. The "sqrtp" one has better performance for large p, but only works when $p > 6N$ (N = prec). The "auto" option selects "sqrtp" whenever possible.
  Note that if the "sqrtp" algorithm is used, a consistency check will automatically be applied, regardless of the setting of the "check" flag.

OUTPUT: p-adic number to precision prec

Note: If the discriminant of the curve has nonzero valuation at p, then the result will not be returned mod p^{prec}, but it still *will* have prec *digits* of precision.

TODO: - Once we have a better implementation of the "standard" algorithm, the algorithm selection strategy for "auto" needs to be revisited.

AUTHORS:

- David Harvey (2006-09-01): partly based on code written by Robert Bradshaw at the MSRI 2006 modular forms workshop

ACKNOWLEDGMENT: - discussion with Eyal Goren that led to the trace trick.

EXAMPLES: Here is the example discussed in the paper "Computation of p-adic Heights and Log Convergence" (Mazur, Stein, Tate):

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^12 + 2*5^14 + 3*
```

Let's try to higher precision (this is the same answer the MAGMA implementation gives):

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 100)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^12 + 2*5^14 + 3*
```

Check it works at low precision too:

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 1)
2 + O(5)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 2)
2 + 4*5 + O(5^2)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 3)
2 + 4*5 + O(5^3)
```

TODO: With the old(-er), i.e., = sage-2.4 p-adics we got $5 + O(5^2)$ as output, i.e., relative precision 1, but with the newer p-adics we get relative precision 0 and absolute precision 1.

```
sage: EllipticCurve([1, 1, 1, 1, 1]).padic_E2(5, 1)
O(5)
```

Check it works for different models of the same curve (37a), even when the discriminant changes by a power of p (note that E2 depends on the differential too, which is why it gets scaled in some of the examples below):

```
sage: X1 = EllipticCurve([-1, 1/4])
sage: X1.j_invariant(), X1.discriminant()
 (110592/37, 37)
sage: X1.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)

sage: X2 = EllipticCurve([0, 0, 1, -1, 0])
sage: X2.j_invariant(), X2.discriminant()
 (110592/37, 37)
sage: X2.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)

sage: X3 = EllipticCurve([-1*(2**4), 1/4*(2**6)])
sage: X3.j_invariant(), X3.discriminant() / 2**12
 (110592/37, 37)
sage: 2**(-2) * X3.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)

sage: X4 = EllipticCurve([-1*(7**4), 1/4*(7**6)])
sage: X4.j_invariant(), X4.discriminant() / 7**12
 (110592/37, 37)
sage: 7**(-2) * X4.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)
```

```
sage: X5 = EllipticCurve([-1*(5**4), 1/4*(5**6)])
sage: X5.j_invariant(), X5.discriminant() / 5**12
 (110592/37, 37)
sage: 5**(-2) * X5.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)

sage: X6 = EllipticCurve([-1/(5**4), 1/4/(5**6)])
sage: X6.j_invariant(), X6.discriminant() * 5**12
 (110592/37, 37)
sage: 5**2 * X6.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)
```

Test check=True vs check=False:

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 1, check=False)
2 + O(5)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 1, check=True)
2 + O(5)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 30, check=False)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^12 + 2*5^14 + 3*
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 30, check=True)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^12 + 2*5^14 + 3*
```

Here's one using the $p^{1/2}$ algorithm:

```
sage: EllipticCurve([-1, 1/4]).padic_E2(3001, 3, algorithm="sqrtp")
1907 + 2819*3001 + 1124*3001^2 + O(3001^3)
```

padic_height (*p, prec=20, sigma=None, check_hypotheses=True*)

Computes the cyclotomic p-adic height.

The equation of the curve must be minimal at p.

INPUT:

- `p` - prime = 5 for which the curve has semi-stable reduction
- `prec` - integer = 1, desired precision of result
- `sigma` - precomputed value of sigma. If not supplied, this function will call padic_sigma to compute it.
- `check_hypotheses` - boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: A function that accepts two parameters:

- a Q-rational point on the curve whose height should be computed
- optional boolean flag 'check': if False, it skips some input checking, and returns the p-adic height of that point to the desired precision.
- The normalization (sign and a factor 1/2 with respect to some other normalizations that appear in the literature) is chosen in such a way as to make the p-adic Birch Swinnerton-Dyer conjecture hold as stated in [Mazur-Tate-Teitelbaum].

AUTHORS:

- Jennifer Balakrishnan: original code developed at the 2006 MSRI graduate workshop on modular forms
- David Harvey (2006-09-13): integrated into Sage, optimised to speed up repeated evaluations of the returned height function, addressed some thorny precision questions
- David Harvey (2006-09-30): rewrote to use division polynomials for computing denominator of nP.
- David Harvey (2007-02): cleaned up according to algorithms in "Efficient Computation of p-adic Heights"
- Chris Wuthrich (2007-05): added supersingular and multiplicative heights

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: P = E.gens()[0]
sage: h = E.padic_height(5, 10)
sage: h(P)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

An anomalous case:

```
sage: h = E.padic_height(53, 10)
sage: h(P)
26*53^-1 + 30 + 20*53 + 47*53^2 + 10*53^3 + 32*53^4 + 9*53^5 + 22*53^6 + 35*53^7 + 30*53^8 +
```

Boundary case:

```
sage: E.padic_height(5, 3)(P)
5 + 5^2 + O(5^3)
```

A case that works the division polynomial code a little harder:

```
sage: E.padic_height(5, 10)(5*P)
5^3 + 5^4 + 5^5 + 3*5^8 + 4*5^9 + O(5^10)
```

Check that answers agree over a range of precisions:

```
sage: max_prec = 30    # make sure we get past p^2    # long time
sage: full = E.padic_height(5, max_prec)(P)           # long time
sage: for prec in range(1, max_prec):                 # long time
...       assert E.padic_height(5, prec)(P) == full   # long time
```

A supersingular prime for a curve:

```
sage: E = EllipticCurve('37a')
sage: E.is_supersingular(3)
True
sage: h = E.padic_height(3, 5)
sage: h(E.gens()[0])
(3 + 3^3 + O(3^6), 2*3^2 + 3^3 + 3^4 + 3^5 + 2*3^6 + O(3^7))
sage: E.padic_regulator(5)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + 5^10 + 3*5^11 + 3*5^12 + 5^13 + 4*5^14 + 5^15 + 2*5^16
sage: E.padic_regulator(3, 5)
(3 + 2*3^2 + 3^3 + O(3^4), 3^2 + 2*3^3 + 3^4 + O(3^5))
```

A torsion point in both the good and supersingular cases:

```
sage: E = EllipticCurve('11a')
sage: P = E.torsion_subgroup().gens()[0]; P
(5 : 5 : 1)
sage: h = E.padic_height(19, 5)
sage: h(P)
0
sage: h = E.padic_height(5, 5)
sage: h(P)
0
```

The result is not dependend on the model for the curve: sage: E = EllipticCurve([0,0,0,0,2^12*17]) sage: Em = E.minimal_model() sage: P = E.gens()[0] sage: Pm = Em.gens()[0] sage: h = E.padic_height(7) sage: hm = Em.padic_height(7) sage: h(P) == hm(Pm) True

padic_height_pairing_matrix(*p, prec=20, height=None, check_hypotheses=True*)

Computes the cyclotomic p-adic height pairing matrix of this curve with respect to the basis self.gens() for the Mordell-Weil group for a given odd prime p of good ordinary reduction.

INPUT:

- p - prime = 5
- prec - answer will be returned modulo p^{prec}
- height - precomputed height function. If not supplied, this function will call padic_height to compute it.
- check_hypotheses - boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: The p-adic cyclotomic height pairing matrix of this curve to the given precision.

TODO: - remove restriction that curve must be in minimal weierstrass form. This is currently required for E.gens().

AUTHORS:

- David Harvey, Liang Xiao, Robert Bradshaw, Jennifer Balakrishnan: original implementation at the 2006 MSRI graduate workshop on modular forms
- David Harvey (2006-09-13): cleaned up and integrated into Sage, removed some redundant height computations

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.padic_height_pairing_matrix(5, 10)
[5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)]
```

A rank two example:

```
sage: e =EllipticCurve('389a')
sage: e._set_gens([e(-1, 1), e(1,0)])  # avoid platform dependent gens
sage: e.padic_height_pairing_matrix(5,10)
[                      3*5 + 2*5^2 + 5^4 + 5^5 + 5^7 + 4*5^9 + O(5^10) 5 + 4*5^2 + 5^3 + 2*5
[5 + 4*5^2 + 5^3 + 2*5^4 + 3*5^5 + 4*5^6 + 5^7 + 5^8 + 2*5^9 + O(5^10)
```

An anomalous rank 3 example:

```
sage: e = EllipticCurve("5077a")
sage: e._set_gens([e(-1,3), e(2,0), e(4,6)])
sage: e.padic_height_pairing_matrix(5,4)
[4 + 3*5 + 4*5^2 + 4*5^3 + O(5^4)       4 + 4*5^2 + 2*5^3 + O(5^4)       3*5 + 4*5^2 + 5^3 +
[      4 + 4*5^2 + 2*5^3 + O(5^4)   3 + 4*5 + 3*5^2 + 5^3 + O(5^4)                 2 + 4*5 +
[      3*5 + 4*5^2 + 5^3 + O(5^4)                 2 + 4*5 + O(5^4)     1 + 3*5 + 5^2 + 5^3 +
```

padic_height_via_multiply(*p, prec=20, E2=None, check_hypotheses=True*)

Computes the cyclotomic p-adic height.

The equation of the curve must be minimal at p.

INPUT:

- p - prime = 5 for which the curve has good ordinary reduction
- prec - integer = 2, desired precision of result
- E2 - precomputed value of E2. If not supplied, this function will call padic_E2 to compute it. The value supplied must be correct mod $p^(prec - 2)$ (or slightly higher in the anomalous case; see the code for details).
- check_hypotheses - boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: A function that accepts two parameters:

- a Q-rational point on the curve whose height should be computed
- optional boolean flag 'check': if False, it skips some input checking, and returns the p-adic height of that point to the desired precision.
- The normalization (sign and a factor 1/2 with respect to some other normalizations that appear in the literature) is chosen in such a way as to make the p-adic Birch Swinnerton-Dyer conjecture hold as stated in [Mazur-Tate-Teitelbaum].

AUTHORS:

- David Harvey (2008-01): based on the padic_height() function, using the algorithm of"Computing p-adic heights via point multiplication"

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: P = E.gens()[0]
sage: h = E.padic_height_via_multiply(5, 10)
sage: h(P)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

An anomalous case:

```
sage: h = E.padic_height_via_multiply(53, 10)
sage: h(P)
26*53^-1 + 30 + 20*53 + 47*53^2 + 10*53^3 + 32*53^4 + 9*53^5 + 22*53^6 + 35*53^7 + 30*53^8 +
```

Supply the value of E2 manually:

```
sage: E2 = E.padic_E2(5, 8)
sage: E2
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + O(5^8)
sage: h = E.padic_height_via_multiply(5, 10, E2=E2)
sage: h(P)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

Boundary case:

```
sage: E.padic_height_via_multiply(5, 3)(P)
5 + 5^2 + O(5^3)
```

Check that answers agree over a range of precisions:

```
sage: max_prec = 30    # make sure we get past p^2    # long time
sage: full = E.padic_height(5, max_prec)(P)           # long time
sage: for prec in range(2, max_prec):                 # long time
...       assert E.padic_height_via_multiply(5, prec)(P) == full   # long time
```

padic_lseries(*p, normalize='L_ratio', use_eclib=False*)

Return the p-adic L-series of self at p, which is an object whose approx method computes approximation to the true p-adic L-series to any desired precision.

INPUT:

- p - prime
- use_eclib - bool (default:False); whether or not to use John Cremona's eclib for the computation of modular symbols
- normalize - 'L_ratio' (default), 'period' or 'none'; this is describes the way the modular symbols are normalized. See modular_symbol for more details.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: L = E.padic_lseries(5); L
5-adic L-series of Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: type(L)
<class 'sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesOrdinary'>
```

We compute the 3-adic L-series of two curves of rank 0 and in each case verify the interpolation property for their leading coefficient (i.e., value at 0):

```
sage: e = EllipticCurve('11a')
sage: ms = e.modular_symbol()
```

```
sage: [ms(1/11), ms(1/3), ms(0), ms(oo)]
[0, -3/10, 1/5, 0]
sage: ms(0)
1/5
sage: L = e.padic_lseries(3)
sage: P = L.series(5)
sage: P(0)
2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + O(3^7)
sage: alpha = L.alpha(9); alpha
2 + 3^2 + 2*3^3 + 2*3^4 + 2*3^6 + 3^8 + O(3^9)
sage: R.<x> = QQ[]
sage: f = x^2 - e.ap(3)*x + 3
sage: f(alpha)
O(3^9)
sage: r = e.lseries().L_ratio(); r
1/5
sage: (1 - alpha^(-1))^2 * r
2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + 3^7 + O(3^9)
sage: P(0)
2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + O(3^7)
```

Next consider the curve 37b:

```
sage: e = EllipticCurve('37b')
sage: L = e.padic_lseries(3)
sage: P = L.series(5)
sage: alpha = L.alpha(9); alpha
1 + 2*3 + 3^2 + 2*3^5 + 2*3^7 + 3^8 + O(3^9)
sage: r = e.lseries().L_ratio(); r
1/3
sage: (1 - alpha^(-1))^2 * r
3 + 3^2 + 2*3^4 + 2*3^5 + 2*3^6 + 3^7 + O(3^9)
sage: P(0)
3 + 3^2 + 2*3^4 + 2*3^5 + O(3^6)
```

We can use eclib to compute the L-series:

```
sage: e = EllipticCurve('11a')
sage: L = e.padic_lseries(3,use_eclib=True)
sage: L.series(5,prec=10)
1 + 2*3^3 + 3^6 + O(3^7) + (2 + 2*3 + 3^2 + O(3^4))*T + (2 + 3 + 3^2 + 2*3^3 + O(3^4))*T^2 +
```

padic_regulator(*p, prec=20, height=None, check_hypotheses=True*)

Computes the cyclotomic p-adic regulator of this curve.

INPUT:

- p - prime = 5
- prec - answer will be returned modulo p^{prec}
- height - precomputed height function. If not supplied, this function will call padic_height to compute it.
- check_hypotheses - boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: The p-adic cyclotomic regulator of this curve, to the requested precision.

If the rank is 0, we output 1.

TODO: - remove restriction that curve must be in minimal weierstrass form. This is currently required for E.gens().

AUTHORS:

- Liang Xiao: original implementation at the 2006 MSRI graduate workshop on modular forms

- David Harvey (2006-09-13): cleaned up and integrated into Sage, removed some redundant height computations
- Chris Wuthrich (2007-05-22): added multiplicative and supersingular cases
- David Harvey (2007-09-20): fixed some precision loss that was occurring

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.padic_regulator(5, 10)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

An anomalous case:

```
sage: E.padic_regulator(53, 10)
26*53^-1 + 30 + 20*53 + 47*53^2 + 10*53^3 + 32*53^4 + 9*53^5 + 22*53^6 + 35*53^7 + 30*53^8 +
```

An anomalous case where the precision drops some:

```
sage: E = EllipticCurve("5077a")
sage: E.padic_regulator(5, 10)
5 + 5^2 + 4*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 4*5^7 + 2*5^8 + 5^9 + O(5^10)
```

Check that answers agree over a range of precisions:

```
sage: max_prec = 30     # make sure we get past p^2     # long time
sage: full = E.padic_regulator(5, max_prec)             # long time
sage: for prec in range(1, max_prec):                   # long time
...       assert E.padic_regulator(5, prec) == full     # long time
```

A case where the generator belongs to the formal group already (trac #3632):

```
sage: E = EllipticCurve([37,0])
sage: E.padic_regulator(5,10)
2*5^2 + 2*5^3 + 5^4 + 5^5 + 4*5^6 + 3*5^8 + 4*5^9 + O(5^10)
```

The result is not dependend on the model for the curve:

```
sage: E = EllipticCurve([0,0,0,0,2^12*17])
sage: Em = E.minimal_model()
sage: E.padic_regulator(7) == Em.padic_regulator(7)
True
```

padic_sigma(*p, N=20, E2=None, check=False, check_hypotheses=True*)

Computes the p-adic sigma function with respect to the standard invariant differential $dx/(2y+a_1x+a_3)$, as defined by Mazur and Tate, as a power series in the usual uniformiser t at the origin.

The equation of the curve must be minimal at p.

INPUT:

- `p` - prime = 5 for which the curve has good ordinary reduction
- `N` - integer = 1, indicates precision of result; see OUTPUT section for description
- `E2` - precomputed value of E2. If not supplied, this function will call padic_E2 to compute it. The value supplied must be correct mod p^{N-2}.
- `check` - boolean, whether to perform a consistency check (i.e. verify that the computed sigma satisfies the defining
- `differential equation` - note that this does NOT guarantee correctness of all the returned digits, but it comes pretty close :-))
- `check_hypotheses` - boolean, whether to check that this is a curve for which the p-adic sigma function makes sense

OUTPUT: A power series $t + \cdots$ with coefficients in $\mathbf{Z}_p$.

The output series will be truncated at $O(t^{N+1})$, and the coefficient of t^n for $n \geq 1$ will be correct to precision $O(p^{N-n+1})$.

In practice this means the following. If $t_0 = p^k u$, where u is a p-adic unit with at least N digits of precision, and $k \geq 1$, then the returned series may be used to compute $\sigma(t_0)$ correctly modulo p^{N+k} (i.e. with N correct p-adic digits).

ALGORITHM: Described in "Efficient Computation of p-adic Heights" (David Harvey), which is basically an optimised version of the algorithm from "p-adic Heights and Log Convergence" (Mazur, Stein, Tate).

Running time is soft-$O(N^2 \log p)$, plus whatever time is necessary to compute E_2.

AUTHORS:

- David Harvey (2006-09-12)
- David Harvey (2007-02): rewrote

EXAMPLES:

```
sage: EllipticCurve([-1, 1/4]).padic_sigma(5, 10)
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7 + O(5^8))*t^3 +
```

Run it with a consistency check:

```
sage: EllipticCurve("37a").padic_sigma(5, 10, check=True)
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7 + O(5^8))*t^3 +
```

Boundary cases:

```
sage: EllipticCurve([1, 1, 1, 1, 1]).padic_sigma(5, 1)
 (1 + O(5))*t + O(t^2)
sage: EllipticCurve([1, 1, 1, 1, 1]).padic_sigma(5, 2)
 (1 + O(5^2))*t + (3 + O(5))*t^2 + O(t^3)
```

Supply your very own value of E2:

```
sage: X = EllipticCurve("37a")
sage: my_E2 = X.padic_E2(5, 8)
sage: my_E2 = my_E2 + 5**5    # oops!!!
sage: X.padic_sigma(5, 10, E2=my_E2)
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 4*5^5 + 2*5^6 + 3*5^7 + O(5^8)
```

Check that sigma is "weight 1".

```
sage: f = EllipticCurve([-1, 3]).padic_sigma(5, 10)
sage: g = EllipticCurve([-1*(2**4), 3*(2**6)]).padic_sigma(5, 10)
sage: t = f.parent().gen()
sage: f(2*t)/2
(1 + O(5^10))*t + (4 + 3*5 + 3*5^2 + 3*5^3 + 4*5^4 + 4*5^5 + 3*5^6 + 5^7 + O(5^8))*t^3 + (3
sage: g
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (4 + 3*5 + 3*5^2 + 3*5^3 + 4*5^4 + 4*5^5 + 3*5^6 +
sage: f(2*t)/2 -g
O(t^11)
```

Test that it returns consistent results over a range of precision:

```
sage: max_N = 30   # get up to at least p^2          # long time
sage: E = EllipticCurve([1, 1, 1, 1, 1])             # long time
sage: p = 5                                          # long time
sage: E2 = E.padic_E2(5, max_N)                      # long time
sage: max_sigma = E.padic_sigma(p, max_N, E2=E2)     # long time
sage: for N in range(3, max_N):                      # long time
...       sigma = E.padic_sigma(p, N, E2=E2)         # long time
...       assert sigma == max_sigma
```

padic_sigma_truncated(*p, N=20, lamb=0, E2=None, check_hypotheses=True*)

Computes the p-adic sigma function with respect to the standard invariant differential $dx/(2y+a_1x+a_3)$, as defined by Mazur and Tate, as a power series in the usual uniformiser t at the origin.

The equation of the curve must be minimal at p.

This function differs from padic_sigma() in the precision profile of the returned power series; see OUTPUT below.

INPUT:

- `p` - prime = 5 for which the curve has good ordinary reduction
- `N` - integer = 2, indicates precision of result; see OUTPUT section for description
- `lamb` - integer = 0, see OUTPUT section for description
- `E2` - precomputed value of E2. If not supplied, this function will call padic_E2 to compute it. The value supplied must be correct mod p^{N-2}.
- `check_hypotheses` - boolean, whether to check that this is a curve for which the p-adic sigma function makes sense

OUTPUT: A power series $t + \cdots$ with coefficients in $\mathbf{Z}_p$.

The coefficient of t^j for $j \geq 1$ will be correct to precision $O(p^{N-2+(3-j)(lamb+1)})$.

ALGORITHM: Described in "Efficient Computation of p-adic Heights" (David Harvey, to appear in LMS JCM), which is basically an optimised version of the algorithm from "p-adic Heights and Log Convergence" (Mazur, Stein, Tate), and "Computing p-adic heights via point multiplication" (David Harvey, still draft form).

Running time is soft-$O(N^2 \lambda^{-1} \log p)$, plus whatever time is necessary to compute E_2.

AUTHOR:

- David Harvey (2008-01): wrote based on previous padic_sigma function

EXAMPLES:

```
sage: E = EllipticCurve([-1, 1/4])
sage: E.padic_sigma_truncated(5, 10)
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7 + O(5^8))*t^3 +
```

Note the precision of the t^3 coefficient depends only on N, not on lamb:

```
sage: E.padic_sigma_truncated(5, 10, lamb=2)
O(5^17) + (1 + O(5^14))*t + O(5^11)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7 + O(5^8))*t^3 +
```

Compare against plain padic_sigma() function over a dense range of N and lamb

```
sage: E = EllipticCurve([1, 2, 3, 4, 7])                                  # long time
sage: E2 = E.padic_E2(5, 50)                                              # long time
sage: for N in range(2, 10):                                              # long time
...       for lamb in range(10):                                          # long time
...           correct = E.padic_sigma(5, N + 3*lamb, E2=E2)               # long time
...           compare = E.padic_sigma_truncated(5, N=N, lamb=lamb, E2=E2)     # long time
...           assert compare == correct                                   # long time
```

pari_curve(*prec=None, factor=1*)

Return the PARI curve corresponding to this elliptic curve.

INPUT:

- `prec` - The precision of quantities calculated for the returned curve, in bits. If None, defaults to factor multiplied by the precision of the largest cached curve (or the default real precision if none yet computed).
- `factor` - The factor by which to increase the precision over the maximum previously computed precision. Only used if prec (which gives an explicit precision) is None.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])
sage: e = E.pari_curve()
sage: type(e)
```

```
<type 'sage.libs.pari.gen.gen'>
sage: e.type()
't_VEC'
sage: e.ellan(10)
[1, -2, -3, 2, -2, 6, -1, 0, 6, 4]

sage: E = EllipticCurve(RationalField(), ['1/3', '2/3'])
sage: e = E.pari_curve(prec = 100)
sage: E._pari_curve.has_key(100)
True
sage: e.type()
't_VEC'
sage: e[:5]
[0, 0, 0, 1/3, 2/3]
```

This shows that the bug uncovered by trac #3954 is fixed:

```
sage: E._pari_curve.has_key(100)
True

sage: E = EllipticCurve('37a1').pari_curve()
sage: E[14].python().prec()
64
sage: [a.precision() for a in E]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4, 4, 4, 4] # 32-bit
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 3, 3, 3, 3] # 64-bit
```

This shows that the bug uncovered by trac #4715 is fixed:

```
sage: Ep = EllipticCurve('903b3').pari_curve()
```

pari_mincurve(*prec=None, factor=1*)

Return the PARI curve corresponding to a minimal model for this elliptic curve.

INPUT:

- `prec` - The precision of quantities calculated for the returned curve, in bits. If None, defaults to factor multiplied by the precision of the largest cached curve (or the default real precision if none yet computed).
- `factor` - The factor by which to increase the precision over the maximum previously computed precision. Only used if prec (which gives an explicit precision) is None.

EXAMPLES:

```
sage: E = EllipticCurve(RationalField(), ['1/3', '2/3'])
sage: e = E.pari_mincurve()
sage: e[:5]
[0, 0, 0, 27, 486]
sage: E.conductor()
47232
sage: e.ellglobalred()
[47232, [1, 0, 0, 0], 2]
```

period_lattice(*embedding=None*)

Returns the period lattice of the elliptic curve.

INPUT:

- `embedding` - ignored (for compatibility with the period_lattice function for elliptic_curve_number_field)

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice()
Period lattice associated to Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

point_search(*height_limit, verbose=True*)

Search for points on a curve up to an input bound on the naive logarithmic height.

INPUT:

- `height_limit (float)` - bound on naive height (at most 21,
- `or mwrank overflows` - see below)
- `verbose (bool)` - (default: True)
  If True, report on each point as found together with linear relations between the points found and the saturation process.
  If False, just return the result.

OUTPUT: points (list) - list of independent points which generate the subgroup of the Mordell-Weil group generated by the points found and then p-saturated for p20.

Warning: height_limit is logarithmic, so increasing by 1 will cause the running time to increase by a factor of approximately 4.5 (=exp(1.5)). The limit of 21 is to prevent overflow, but in any case using height_limit=20 takes rather a long time!

IMPLEMENTATION: Uses Cremona's mwrank package. At the heart of this function is Cremona's port of Stoll's ratpoints program (version 1.4).

EXAMPLES:

```
sage: E=EllipticCurve('389a1')
sage: E.point_search(5, verbose=False)
[(0 : -1 : 1), (-1 : 1 : 1)]
```

Increasing the height_limit takes longer, but finds no more points:

```
sage: E.point_search(10, verbose=False)
[(0 : -1 : 1), (-1 : 1 : 1)]
```

In fact this curve has rank 2 so no more than 2 points will ever be output, but we are not using this fact.

```
sage: E.saturation(_)
([(0 : -1 : 1), (-1 : 1 : 1)], '1', 0.152460172772408)
```

What this shows is that if the rank is 2 then the points listed do generate the Mordell-Weil group (mod torsion). Finally,

```
sage: E.rank()
2
```

q_eigenform(*prec*)

Synonym for `self.q_expansion(prec)`.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.q_eigenform(10)
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + 6*q^6 - q^7 + 6*q^9 + O(q^10)
sage: E.q_eigenform(10) == E.q_expansion(10)
True
```

q_expansion(*prec*)

Return the q-expansion to precision prec of the newform attached to this elliptic curve.

INPUT:

- `prec` - an integer

OUTPUT:

a power series (in th evariable 'q')

Note: If you want the output to be a modular form and not just a q-expansion, use `modular_form()`.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.q_expansion(20)
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + 6*q^6 - q^7 + 6*q^9 + 4*q^10 - 5*q^11 - 6*q^12 - 2*q^13
```

quadratic_twist(*D*)

Return the global minimal model of the quadratic twist of this curve by D.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E7=E.quadratic_twist(7); E7
Elliptic Curve defined by y^2  = x^3 - 784*x + 5488 over Rational Field
sage: E7.conductor()
29008
sage: E7.quadratic_twist(7) == E
True
```

rank(*use_database=False, verbose=False, only_use_mwrank=True, algorithm='mwrank_shell', proof=None*)

Return the rank of this elliptic curve, assuming no conjectures.

If we fail to provably compute the rank, raises a RuntimeError exception.

INPUT:

- `use_database (bool)` - (default: False), if True, try to look up the regulator in the Cremona database.
- `verbose` - (default: None), if specified changes the verbosity of mwrank computations. algorithm -
- – `'mwrank_shell'` - call mwrank shell command
- – `'mwrank_lib'` - call mwrank c library
- `only_use_mwrank` - (default: True) if False try using analytic rank methods first.
- `proof` - bool or None (default: None, see proof.elliptic_curve or sage.structure.proof). Note that results obtained from databases are considered proof = True

OUTPUT:

- `rank (int)` - the rank of the elliptic curve.

IMPLEMENTATION: Uses L-functions, mwrank, and databases.

EXAMPLES:

```
sage: EllipticCurve('11a').rank()
0
sage: EllipticCurve('37a').rank()
1
sage: EllipticCurve('389a').rank()
2
sage: EllipticCurve('5077a').rank()
3
sage: EllipticCurve([1, -1, 0, -79, 289]).rank()    # long time.  This will use the default p
4
sage: EllipticCurve([0, 0, 1, -79, 342]).rank(proof=False)  # long time -- but under a minut
5
sage: EllipticCurve([0, 0, 1, -79, 342]).simon_two_descent()[0]  # much faster -- almost ins
5
```

Examples with denominators in defining equations:

```
sage: E = EllipticCurve( [0, 0, 0, 0, -675/4])
sage: E.rank()
0
sage: E = EllipticCurve( [0, 0, 1/2, 0, -1/5])
sage: E.rank()
1
sage: E.minimal_model().rank()
1
```

A large example where mwrank doesn't determine the result with certainty:

```
sage: EllipticCurve([1,0,0,0,37455]).rank(proof=False)
0
sage: EllipticCurve([1,0,0,0,37455]).rank(proof=True)
...
RuntimeError: Rank not provably correct.
```

real_components()

Returns 1 if there is 1 real component and 2 if there are 2.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.real_components ()
2
sage: E = EllipticCurve('37b')
sage: E.real_components ()
2
sage: E = EllipticCurve('11a')
sage: E.real_components ()
1
```

reducible_primes()

Returns a list of the primes p such that the mod p representation $\rho_{E,p}$ is reducible. For all other primes the representation is irreducible.

Note: This is *not* provably correct in general. See the documentation for `isogeny_class()`.

EXAMPLES:

```
sage: E = EllipticCurve('225a')
sage: E.reducible_primes()
[3]
```

reduction(p)

Return the reduction of the elliptic curve at a prime of good reduction.

Note: All is done in `self.change_ring(GF(p))`; all we do is check that p is prime and does not divide the discriminant.

INPUT:

- p - a (positive) prime number

OUTPUT: an elliptic curve over the finite field GF(p)

EXAMPLES:

```
sage: E = EllipticCurve('389a1')
sage: E.reduction(2)
Elliptic Curve defined by y^2 + y = x^3 + x^2 over Finite Field of size 2
sage: E.reduction(3)
Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of size 3
sage: E.reduction(5)
Elliptic Curve defined by y^2 + y = x^3 + x^2 + 3*x over Finite Field of size 5
sage: E.reduction(38)
```

```
...
AttributeError: p must be prime.
sage: E.reduction(389)
...
AttributeError: The curve must have good reduction at p.
```

regulator(*use_database=True, proof=None, precision=None*)

Returns the regulator of this curve, which must be defined over Q.

INPUT:

- `use_database` - bool (default: False), if True, try to look up the generators in the Cremona database.
- `proof` - bool or None (default: None, see proof.[tab] or sage.structure.proof). Note that results from databases are considered proof = True
- `precision` - int or None (default: None): the precision in bits of the result (default real precision if None)

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])
sage: E.regulator()               # long time (1 second)
0.0511114082399688
sage: EllipticCurve('11a').regulator()
1.00000000000000
sage: EllipticCurve('37a').regulator()
0.0511114082399688
sage: EllipticCurve('389a').regulator()
0.152460177943144
sage: EllipticCurve('5077a').regulator()
0.41714355875838...
sage: EllipticCurve([1, -1, 0, -79, 289]).regulator()  # long time (seconds)
1.50434488827528
sage: EllipticCurve([0, 0, 1, -79, 342]).regulator(proof=False)  # long time (seconds)
14.790527570131...
```

regulator_of_points(*points=, [], precision=None*)

Returns the regulator of the given points on this curve.

INPUT:

- `points` -(default: empty list) a list of points on this curve
- `precision` - int or None (default: None): the precision in bits of the result (default real precision if None)

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: P = E(0,0)
sage: Q = E(1,0)
sage: E.regulator_of_points([P,Q])
0.000000000000000
sage: 2*P==Q
True

sage: E = EllipticCurve('5077a1')
sage: points = [E.lift_x(x) for x in [-2,-7/4,1]]
sage: E.regulator_of_points(points)
0.417143558758384
sage: E.regulator_of_points(points,precision=100)
0.41714355875838396981711954462
```

```
sage: E = EllipticCurve('389a')
sage: E.regulator_of_points()
1.00000000000000
sage: points = [P,Q] = [E(-1,1),E(0,-1)]
sage: E.regulator_of_points(points)
0.152460177943144
sage: E.regulator_of_points(points, precision=100)
0.15246017794314375162432475705
sage: E.regulator_of_points(points, precision=200)
0.152460177943143751624324757049455823243727077486630817840280980046053225683562463604114816
sage: E.regulator_of_points(points, precision=300)
0.152460177943143751624324757049455823243727077486630817840280980046053225683562463604114816
```

root_number()

Returns the root number of this elliptic curve.

This is 1 if the order of vanishing of the L-function L(E,s) at 1 is even, and -1 if it is odd.

EXAMPLES:

```
sage: EllipticCurve('11a1').root_number()
1
sage: EllipticCurve('37a1').root_number()
-1
sage: EllipticCurve('389a1').root_number()
1
```

satisfies_heegner_hypothesis(*D*)

Returns True precisely when D is a fundamental discriminant that satisfies the Heegner hypothesis for this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.satisfies_heegner_hypothesis(-7)
True
sage: E.satisfies_heegner_hypothesis(-11)
False
```

saturation(*points, verbose=False, max_prime=0, odd_primes_only=False*)

Given a list of rational points on E, compute the saturation in E(Q) of the subgroup they generate.

INPUT:

- `points (list)` - list of points on E
- `verbose (bool)` - (default: False), if True, give verbose output
- `max_prime (int)` - (default: 0), saturation is performed for all primes up to max_prime. If max_prime==0, perform saturation at *all* primes, i.e., compute the true saturation.
- `odd_primes_only (bool)` - only do saturation at odd primes

OUTPUT:

- `saturation (list)` - points that form a basis for the saturation
- `index (int)` - the index of the group generated by points in their saturation
- `regulator (real with default precision)` - regulator of saturated points.

IMPLEMENTATION: Uses Cremona's mwrank package. With max_prime=0, we call mwrank with successively larger prime bounds until the full saturation is provably found. The results of saturation at the previous primes is stored in each case, so this should be reasonably fast.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: P=E(0,0)
sage: Q=5*P; Q
(1/4 : -5/8 : 1)
sage: E.saturation([Q])
([(0 : 0 : 1)], '5', 0.0511114075779915)
```

sea(*p, early_abort=False*)

Return the number of points on E over $\mathbf{F}_p$ computed using the SEA algorithm, as implemented in PARI by Christophe Doche and Sylvain Duquesne.

INPUT:

- `p` - a prime number
- `early_abort` - bool (default: False); if True an early abort technique is used and the computation is interrupted as soon as a small divisor of the order is detected.

Note: As of 2006-02-02 this function does not work on Microsoft Windows under Cygwin (though it works under VMWare of course).

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.sea(next_prime(10^30))
1000000000000001426441464441649
```

selmer_rank_bound()

Bound on the rank of the curve, computed using the 2-selmer group. This is the rank of the curve minus the rank of the 2-torsion, minus a number determined by whatever mwrank was able to determine related to Sha[2]. Thus in many cases, this is the actual rank of the curve.

EXAMPLE: The following is the curve 960D1, which has rank 0, but Sha of order 4.

```
sage: E = EllipticCurve([0, -1, 0, -900, -10098])
sage: E.selmer_rank_bound()
0
```

It gives 0 instead of 2, because it knows Sha is nontrivial. In contrast, for the curve 571A, also with rank 0 and Sha of order 4, we get a worse bound:

```
sage: E = EllipticCurve([0, -1, 1, -929, -10595])
sage: E.selmer_rank_bound()
2
sage: E.rank(only_use_mwrank=False)    # uses L-function
0
```

sha()

Return an object of class 'sage.schemes.elliptic_curves.sha_tate.Sha' attached to this elliptic curve.

This can be used in functions related to bounding the order of Sha (The Tate-Shafarevich group of the curve).

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: S=E.sha()
sage: S
Shafarevich-Tate group for the Elliptic Curve defined by y^2 + y = x^3 - x over Rational Fie
sage: S.bound_kolyvagin()
([2], 1)
```

silverman_height_bound()

Return the Silverman height bound. This is a positive real (floating point) number B such that for all rational points P on the curve,

$$h(P) \leq \hat{h}(P) + B$$

where h(P) is the logarithmic height of P and $\hat{h}(P)$ is the canonical height.

Note that the CPS_height_bound is often better (i.e. smaller) than the Silverman bound.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.silverman_height_bound()
4.8254007581809182
sage: E.CPS_height_bound()
0.16397076103046915
```

simon_two_descent (*verbose=0, lim1=5, lim3=50, limtriv=10, maxprob=20, limbigprime=30*)

Given a curve with no 2-torsion, computes (probably) the rank of the Mordell-Weil group, with certainty the rank of the 2-Selmer group, and a list of independent points on the curve.

INPUT:

- `verbose` - integer, 0,1,2,3; (default: 0), the verbosity level
- `lim1` - (default: 5) limite des points triviaux sur les quartiques
- `lim3` - (default: 50) limite des points sur les quartiques ELS
- `limtriv` - (default: 10) limite des points triviaux sur la courbe elliptique
- `maxprob` - (default: 20)
- `limbigprime` - (default: 30) to distinguish between small and large prime numbers. Use probabilistic tests for large primes. If 0, don't any probabilistic tests.

OUTPUT:

- `integer` - "probably" the rank of self
- `integer` - the 2-rank of the Selmer group
- `list` - list of independent points on the curve.

IMPLEMENTATION: Uses Denis Simon's GP/PARI scripts from http://www.math.unicaen.fr/~simon/

EXAMPLES: These computations use pseudo-random numbers, so we set the seed for reproducible testing.

We compute the ranks of the curves of lowest known conductor up to rank 8. Amazingly, each of these computations finishes almost instantly!

```
sage: E = EllipticCurve('11a1')
sage: set_random_seed(0)
sage: E.simon_two_descent()
(0, 0, [])
sage: E = EllipticCurve('37a1')
sage: set_random_seed(0)
sage: E.simon_two_descent()
(1, 1, [(0 : 0 : 1)])
sage: E = EllipticCurve('389a1')
sage: set_random_seed(0)
sage: E.simon_two_descent()
(2, 2, [(1 : 0 : 1), (-11/9 : -55/27 : 1)])
sage: E = EllipticCurve('5077a1')
sage: set_random_seed(0)
sage: E.simon_two_descent()
(3, 3, [(1 : 0 : 1), (2 : -1 : 1), (0 : 2 : 1)])
```

In this example Simon's program does not find any points, though it does correctly compute the rank of the 2-Selmer group.

```
sage: E = EllipticCurve([1, -1, 0, -751055859, -7922219731979])     # long (0.6 seconds)
sage: set_random_seed(0)
sage: E.simon_two_descent ()
(1, 1, [])
```

The rest of these entries were taken from Tom Womack's page http://tom.womack.net/maths/conductors.htm

```
sage: E = EllipticCurve([1, -1, 0, -79, 289])
sage: set_random_seed(0)
sage: E.simon_two_descent()
(4, 4, [(6 : -5 : 1), (4 : 3 : 1), (5 : -3 : 1), (8 : -15 : 1)])
sage: E = EllipticCurve([0, 0, 1, -79, 342])
sage: set_random_seed(0)
sage: E.simon_two_descent()
(5, 5, [(5 : 8 : 1), (10 : 23 : 1), (3 : 11 : 1), (4 : -10 : 1), (0 : 18 : 1)])
sage: E = EllipticCurve([1, 1, 0, -2582, 48720])
sage: set_random_seed(0)
sage: r, s, G = E.simon_two_descent(); r,s
(6, 6)
sage: E = EllipticCurve([0, 0, 0, -10012, 346900])
sage: set_random_seed(0)
sage: r, s, G = E.simon_two_descent(); r,s
(7, 7)
sage: E = EllipticCurve([0, 0, 1, -23737, 960366])
sage: set_random_seed(0)
sage: r, s, G = E.simon_two_descent(); r,s
(8, 8)
```

supersingular_primes(*B*)

Return a list of all supersingular primes for this elliptic curve up to and possibly including B.

EXAMPLES:

```
sage: e = EllipticCurve('11a')
sage: e.aplist(20)
[-2, -1, 1, -2, 1, 4, -2, 0]
sage: e.supersingular_primes(1000)
[2, 19, 29, 199, 569, 809]

sage: e = EllipticCurve('27a')
sage: e.aplist(20)
[0, 0, 0, -1, 0, 5, 0, -7]
sage: e.supersingular_primes(97)
[2, 5, 11, 17, 23, 29, 41, 47, 53, 59, 71, 83, 89]
sage: e.ordinary_primes(97)
[7, 13, 19, 31, 37, 43, 61, 67, 73, 79, 97]
sage: e.supersingular_primes(3)
[2]
sage: e.supersingular_primes(2)
[2]
sage: e.supersingular_primes(1)
[]
```

tamagawa_exponent(*p*)

The Tamagawa index of the elliptic curve at p.

This is the index of the component group $E(\mathbf{Q}_p)/E^0(\mathbf{Q}_p)$. It equals the Tamagawa number (as the component group is cyclic) except for types I_m^* (m even) when the group can be $C_2 times C_2$.

EXAMPLES:

```
sage: E = EllipticCurve('816a1')
sage: E.tamagawa_number(2)
4
sage: E.tamagawa_exponent(2)
2
```

```
sage: E.kodaira_symbol(2)
I2*

sage: E = EllipticCurve('200c4')
sage: E.kodaira_symbol(5)
I4*
sage: E.tamagawa_number(5)
4
sage: E.tamagawa_exponent(5)
2
```

See #4715:

```
sage: E=EllipticCurve('117a3')
sage: E.tamagawa_exponent(13)
4
```

tamagawa_number(p)

The Tamagawa number of the elliptic curve at p.

This is the order of the component group $E(\mathbf{Q}_p)/E^0(\mathbf{Q}_p)$.

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: E.tamagawa_number(11)
5
sage: E = EllipticCurve('37b')
sage: E.tamagawa_number(37)
3
```

tamagawa_number_old(p)

The Tamagawa number of the elliptic curve at p.

This is the order of the component group $E(\mathbf{Q}_p)/E^0(\mathbf{Q}_p)$.

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: E.tamagawa_number_old(11)
5
sage: E = EllipticCurve('37b')
sage: E.tamagawa_number_old(37)
3
```

tamagawa_numbers()

Return a list of all Tamagawa numbers for all prime divisors of the conductor (in order).

EXAMPLES:

```
sage: e = EllipticCurve('30a1')
sage: e.tamagawa_numbers()
[2, 3, 1]
sage: vector(e.tamagawa_numbers())
(2, 3, 1)
```

tamagawa_product()

Returns the product of the Tamagawa numbers.

EXAMPLES:

```
sage: E = EllipticCurve('54a')
sage: E.tamagawa_product ()
3
```

tate_curve(*p*)

Creates the Tate Curve over the p-adics associated to this elliptic curves.

This Tate curve a p-adic curve with split multiplicative reduction of the form $y^2 + xy = x^3 + s_4x + s_6$ which is isomorphic to the given curve over the algebraic closure of $\mathbf{Q}_p$. Its points over $\mathbf{Q}_p$ are isomorphic to $\mathbf{Q}_p^\times/q^{\mathbf{Z}}$ for a certain parameter $q \in \mathbf{Z}_p$.

INPUT:

p - a prime where the curve has multiplicative reduction.

EXAMPLES:

```
sage: e = EllipticCurve('130a1')
sage: e.tate_curve(2)
2-adic Tate curve associated to the Elliptic Curve defined by y^2 + x*y + y = x^3 - 33*x + 6
```

The input curve must have multiplicative reduction at the prime.

```
sage: e.tate_curve(3)
...
ValueError: The elliptic curve must have multiplicative reduction at 3
```

We compute with $p = 5$:

```
sage: T = e.tate_curve(5); T
5-adic Tate curve associated to the Elliptic Curve defined by y^2 + x*y + y = x^3 - 33*x + 6
```

We find the Tate parameter q:

```
sage: T.parameter(prec=5)
3*5^3 + 3*5^4 + 2*5^5 + 2*5^6 + 3*5^7 + O(5^8)
```

We compute the $\mathcal{L}$-invariant of the curve:

```
sage: T.L_invariant(prec=10)
5^3 + 4*5^4 + 2*5^5 + 2*5^6 + 2*5^7 + 3*5^8 + 5^9 + O(5^10)
```

three_selmer_rank(*algorithm='UseSUnits'*)

Return the 3-selmer rank of this elliptic curve, computed using Magma.

INPUT:

- `algorithm` - 'Heuristic' (which is usually much faster in large examples), 'FindCubeRoots', or 'UseSUnits' (default)

OUTPUT: nonnegative integer

EXAMPLES: A rank 0 curve:

```
sage: EllipticCurve('11a').three_selmer_rank()       # optional - magma
0
```

A rank 0 curve with rational 3-isogeny but no 3-torsion

```
sage: EllipticCurve('14a3').three_selmer_rank()      # optional - magma
0
```

A rank 0 curve with rational 3-torsion:

```
sage: EllipticCurve('14a1').three_selmer_rank()      # optional - magma
1
```

A rank 1 curve with rational 3-isogeny:

```
sage: EllipticCurve('91b').three_selmer_rank()       # optional - magma
2
```

A rank 0 curve with nontrivial 3-Sha. The Heuristic option makes this about twice as fast as without it.

```
sage: EllipticCurve('681b').three_selmer_rank(algorithm='Heuristic')    # long (10 seconds);
2
```

torsion_order()

Return the order of the torsion subgroup.

EXAMPLES:

```
sage: e = EllipticCurve('11a')
sage: e.torsion_order()
5
sage: type(e.torsion_order())
<type 'sage.rings.integer.Integer'>
sage: e = EllipticCurve([1,2,3,4,5])
sage: e.torsion_order()
1
sage: type(e.torsion_order())
<type 'sage.rings.integer.Integer'>
```

torsion_points(*algorithm='pari'*)

Returns the torsion points of this elliptic curve as a sorted list.

INPUT:

- `algorithm` - string:
  - "pari" - (default) use the pari library
  - "doud" - use Doud's algorithm
  - "lutz_nagell" - use the Lutz-Nagell theorem

OUTPUT: A list of all the torsion points on this elliptic curve.

EXAMPLES:

```
sage: EllipticCurve('11a').torsion_points()
[(0 : 1 : 0), (5 : -6 : 1), (5 : 5 : 1), (16 : -61 : 1), (16 : 60 : 1)]
sage: EllipticCurve('37b').torsion_points()
[(0 : 1 : 0), (8 : -19 : 1), (8 : 18 : 1)]

sage: E=EllipticCurve([-1386747,368636886])
sage: T=E.torsion_subgroup(); T
Torsion Subgroup isomorphic to Multiplicative Abelian Group isomorphic to C8 x C2 associated
sage: T == E.torsion_subgroup(algorithm="doud")
True
sage: T == E.torsion_subgroup(algorithm="lutz_nagell")
True
sage: E.torsion_points()
[(-1293 : 0 : 1),
(-933 : -29160 : 1),
(-933 : 29160 : 1),
(-285 : -27216 : 1),
(-285 : 27216 : 1),
(0 : 1 : 0),
(147 : -12960 : 1),
(147 : 12960 : 1),
(282 : 0 : 1),
(1011 : 0 : 1),
(1227 : -22680 : 1),
(1227 : 22680 : 1),
(2307 : -97200 : 1),
(2307 : 97200 : 1),
(8787 : -816480 : 1),
(8787 : 816480 : 1)]
```

torsion_subgroup(*algorithm='pari'*)

Returns the torsion subgroup of this elliptic curve.

INPUT:

- `algorithm` - string:
- `"pari"` - (default) use the pari library
- `"doud"` - use Doud's algorithm
- `"lutz_nagell"` - use the Lutz-Nagell theorem

OUTPUT: The EllipticCurveTorsionSubgroup instance associated to this elliptic curve.

Note: To see the torsion points as a list, use `torsion_points()`.

EXAMPLES:

```
sage: EllipticCurve('11a').torsion_subgroup()
Torsion Subgroup isomorphic to Multiplicative Abelian Group isomorphic to C5 associated to t
sage: EllipticCurve('37b').torsion_subgroup()
Torsion Subgroup isomorphic to Multiplicative Abelian Group isomorphic to C3 associated to t

sage: e = EllipticCurve([-1386747,368636886]);e
Elliptic Curve defined by y^2  = x^3 - 1386747*x + 368636886 over Rational Field
sage: G = e.torsion_subgroup(); G
Torsion Subgroup isomorphic to Multiplicative Abelian
Group isomorphic to C8 x C2 associated to the Elliptic
Curve defined by y^2 = x^3 - 1386747*x + 368636886 over
Rational Field
sage: G.0
(1227 : 22680 : 1)
sage: G.1
(282 : 0 : 1)
sage: list(G)
[1, P1, P0, P0*P1, P0^2, P0^2*P1, P0^3, P0^3*P1, P0^4, P0^4*P1, P0^5, P0^5*P1, P0^6, P0^6*P1
```

two_descent(*verbose=True, selmer_only=False, first_limit=20, second_limit=8, n_aux=-1, second_descent=1*)

Compute 2-descent data for this curve.

INPUT:

- `verbose` - (default: True) print what mwrank is doing. If False, **no output** is printed.
- `selmer_only` - (default: False) selmer_only switch
- `first_limit` - (default: 20) firstlim is bound on x+z second_limit- (default: 8) secondlim is bound on log max x,z , i.e. logarithmic
- `n_aux` - (default: -1) n_aux only relevant for general 2-descent when 2-torsion trivial; n_aux=-1 causes default to be used (depends on method)
- `second_descent` - (default: True) second_descent only relevant for descent via 2-isogeny

OUTPUT:

Nothing - nothing is returned (though much is printed unless verbose=False)

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.two_descent(verbose=False) # no output
```

two_descent_simon(*verbose=0, lim1=5, lim3=50, limtriv=10, maxprob=20, limbigprime=30*)

Given a curve with no 2-torsion, computes (probably) the rank of the Mordell-Weil group, with certainty the rank of the 2-Selmer group, and a list of independent points on the curve.

INPUT:

- `verbose` - integer, 0,1,2,3; (default: 0), the verbosity level
- `lim1` - (default: 5) limite des points triviaux sur les quartiques

- `lim3` - (default: 50) limite des points sur les quartiques ELS
- `limtriv` - (default: 10) limite des points triviaux sur la courbe elliptique
- `maxprob` - (default: 20)
- `limbigprime` - (default: 30) to distinguish between small and large prime numbers. Use probabilistic tests for large primes. If 0, don't any probabilistic tests.

OUTPUT:

- `integer` - "probably" the rank of self
- `integer` - the 2-rank of the Selmer group
- `list` - list of independent points on the curve.

IMPLEMENTATION: Uses Denis Simon's GP/PARI scripts from http://www.math.unicaen.fr/~simon/

EXAMPLES: These computations use pseudo-random numbers, so we set the seed for reproducible testing.

We compute the ranks of the curves of lowest known conductor up to rank 8. Amazingly, each of these computations finishes almost instantly!

```
sage: E = EllipticCurve('11a1')
sage: set_random_seed(0)
sage: E.simon_two_descent()
(0, 0, [])
sage: E = EllipticCurve('37a1')
sage: set_random_seed(0)
sage: E.simon_two_descent()
(1, 1, [(0 : 0 : 1)])
sage: E = EllipticCurve('389a1')
sage: set_random_seed(0)
sage: E.simon_two_descent()
(2, 2, [(1 : 0 : 1), (-11/9 : -55/27 : 1)])
sage: E = EllipticCurve('5077a1')
sage: set_random_seed(0)
sage: E.simon_two_descent()
(3, 3, [(1 : 0 : 1), (2 : -1 : 1), (0 : 2 : 1)])
```

In this example Simon's program does not find any points, though it does correctly compute the rank of the 2-Selmer group.

```
sage: E = EllipticCurve([1, -1, 0, -751055859, -7922219731979])     # long (0.6 seconds)
sage: set_random_seed(0)
sage: E.simon_two_descent ()
(1, 1, [])
```

The rest of these entries were taken from Tom Womack's page http://tom.womack.net/maths/conductors.htm

```
sage: E = EllipticCurve([1, -1, 0, -79, 289])
sage: set_random_seed(0)
sage: E.simon_two_descent()
(4, 4, [(6 : -5 : 1), (4 : 3 : 1), (5 : -3 : 1), (8 : -15 : 1)])
sage: E = EllipticCurve([0, 0, 1, -79, 342])
sage: set_random_seed(0)
sage: E.simon_two_descent()
(5, 5, [(5 : 8 : 1), (10 : 23 : 1), (3 : 11 : 1), (4 : -10 : 1), (0 : 18 : 1)])
sage: E = EllipticCurve([1, 1, 0, -2582, 48720])
sage: set_random_seed(0)
sage: r, s, G = E.simon_two_descent(); r,s
(6, 6)
sage: E = EllipticCurve([0, 0, 0, -10012, 346900])
sage: set_random_seed(0)
```

```
sage: r, s, G = E.simon_two_descent(); r,s
(7, 7)
sage: E = EllipticCurve([0, 0, 1, -23737, 960366])
sage: set_random_seed(0)
sage: r, s, G = E.simon_two_descent(); r,s
(8, 8)
```

two_torsion_rank()

Return the dimension of the 2-torsion subgroup of $E(\mathbf{Q})$.

This will be 0, 1 or 2.

Note: As a side-effect of calling this function, the full torsion subgroup of the curve is computed (if not already cached). A simpler implementation of this function would be possible (by counting the roots of the 2-division polynomial), but the full torsion subgroup computation is not expensive.

EXAMPLES:

```
sage: EllipticCurve('11a1').two_torsion_rank()
0
sage: EllipticCurve('14a1').two_torsion_rank()
1
sage: EllipticCurve('15a1').two_torsion_rank()
2
```

cremona_curves(*conductors*)

Return iterator over all known curves (in database) with conductor in the list of conductors.

EXAMPLES:

```
sage: [(E.label(), E.rank()) for E in cremona_curves(srange(35,40))]
[('35a1', 0),
('35a2', 0),
('35a3', 0),
('36a1', 0),
('36a2', 0),
('36a3', 0),
('36a4', 0),
('37a1', 1),
('37b1', 0),
('37b2', 0),
('37b3', 0),
('38a1', 0),
('38a2', 0),
('38a3', 0),
('38b1', 0),
('38b2', 0),
('39a1', 0),
('39a2', 0),
('39a3', 0),
('39a4', 0)]
```

cremona_optimal_curves(*conductors*)

Return iterator over all known optimal curves (in database) with conductor in the list of conductors.

EXAMPLES:

```
sage: [(E.label(), E.rank()) for E in cremona_optimal_curves(srange(35,40))]
[('35a1', 0),
('36a1', 0),
('37a1', 1),
```

```
('37b1', 0),
('38a1', 0),
('38b1', 0),
('39a1', 0)]
```

There is one case – 990h3 – when the optimal curve isn't labeled with a 1:

```
sage: [e.cremona_label() for e in cremona_optimal_curves([990])]
['990a1', '990b1', '990c1', '990d1', '990e1', '990f1', '990g1', '990h3', '990i1', '990j1', '990k
```

integral_points_with_bounded_mw_coeffs(*E, mw_base, N*)

Returns the set of integers x which are x-coordinates of points on the curve E which are linear combinations of the generators (basis and torsion points) with coefficients bounded by N.

38.8 Tables of elliptic curves of given rank

The default database of curves contains the following data:

Rank	Number of curves	Maximal conductor
0	30427	9999
1	31871	9999
2	2388	9999
3	836	119888
4	1	234446
5	1	19047851
6	1	5187563742
7	1	382623908456
8	1	457532830151317

AUTHOR: - William Stein (2007-10-07): initial version

See also the functions cremona_curves() and cremona_optimal_curves() which enable easy looping through the Cremona elliptic curve database.

class EllipticCurves()

rank(*rank, tors=0, n=10, labels=False*)

Return a list of at most n non-isogenous curves with given rank and torsion order.

INPUT:

- `rank` (int) – the desired rank
- `tors` (int, default 0) – the desired torsion order (ignored if 0)
- `n` (int, default 10) – the maximum number of curves returned.
- `labels` (bool, default False) – if True, return Cremona labels instead of curves.

OUTPUT:

(list) A list at most n of elliptic curves of required rank.

EXAMPLES:

```
sage: elliptic_curves.rank(n=5, rank=3, tors=2, labels=True)
['59450i1', '59450i2', '61376c1', '61376c2', '65481c1']

sage: elliptic_curves.rank(n=5, rank=0, tors=5, labels=True)
['11a1', '11a3', '38b1', '50b1', '50b2']
```

```
sage: elliptic_curves.rank(n=5, rank=1, tors=7, labels=True)
['574i1', '4730k1', '6378c1']

sage: e = elliptic_curves.rank(6)[0]; e.ainvs(), e.conductor()
([1, 1, 0, -2582, 48720], 5187563742)
sage: e = elliptic_curves.rank(7)[0]; e.ainvs(), e.conductor()
([0, 0, 0, -10012, 346900], 382623908456)
sage: e = elliptic_curves.rank(8)[0]; e.ainvs(), e.conductor()
([0, 0, 1, -23737, 960366], 457532830151317)
```

38.9 Elliptic curves over number fields.

EXAMPLES:

```
sage: k.<i> = NumberField(x^2+1)
sage: E = EllipticCurve([i,2])
sage: E.j_invariant()
-23328/365*i + 864/365
sage: E.simon_two_descent()
(1, 1, [(2*i : -2*i + 2 : 1)])
sage: P = E([2*i,-2*i+2])
sage: P+P
(15/32*i + 3/4 : 139/256*i + 339/256 : 1)
```

class **EllipticCurve_number_field**(*x*, *y=None*)

Elliptic curve over a number field.

EXAMPLES:

```
sage: K.<i>=NumberField(x^2+1)
sage: EllipticCurve([i, i - 1, i + 1, 24*i + 15, 14*i + 35])
Elliptic Curve defined by y^2 + i*x*y + (i+1)*y = x^3 + (i-1)*x^2 + (24*i+15)*x + (14*i+35) over
```

conductor()

Returns the conductor of this elliptic curve as a fractional ideal of the base field.

OUTPUT:

(fractional ideal) The conductor of the curve.

EXAMPLES:

```
sage: K.<i>=NumberField(x^2+1)
sage: EllipticCurve([i, i - 1, i + 1, 24*i + 15, 14*i + 35]).conductor()
Fractional ideal (21*i - 3)
sage: K.<a>=NumberField(x^2-x+3)
sage: EllipticCurve([1 + a , -1 + a , 1 + a , -11 + a , 5 -9*a  ]).conductor()
Fractional ideal (-6*a)
```

A not so well known curve with everywhere good reduction:

```
sage: K.<a>=NumberField(x^2-38)
sage: E=EllipticCurve([0,0,0, 217968148569327655682438l0*a - 134364590724198567128296995, 12
sage: E.conductor()
Fractional ideal (1)
```

global_integral_model()

Return a model of self which is integral at all primes.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2+1)
sage: E = EllipticCurve([i/5,i/5,i/5,i/5,i/5])
sage: P1,P2 = K.primes_above(5)
sage: E.global_integral_model()
Elliptic Curve defined by y^2 + (-i)*x*y + (-25*i)*y = x^3 + 5*i*x^2 + 125*i*x + 3125*i over
```

global_minimal_model(*proof=None*)

Returns a model of self that is integral, minimal at all primes.

Note: This is only implemented for class number 1. In general, such a model may or may not exist.

INPUT:

- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.

OUTPUT:

A global integral and minimal model.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2-38)
sage: E = EllipticCurve([0,0,0, 21796814856932765568243810*a - 13436459072419856712829695,
sage: E.global_minimal_model()
Elliptic Curve defined by y^2 + a*x*y + (a+1)*y = x^3 + (a+1)*x^2 + (3682585202005220468063
```

has_additive_reduction(P)

Return True if this elliptic curve has (bad) additive reduction at the prime P.

INPUT:

- `P` – a prime ideal of the base field of self, or a field element generating such an ideal.

OUTPUT:

(bool) True if the curve has additive reduction at P, else False.

EXAMPLES:

```
sage: E=EllipticCurve('27a1')
sage: [(p,E.has_additive_reduction(p)) for p in prime_range(15)]
[(2, False), (3, True), (5, False), (7, False), (11, False), (13, False)]

sage: K.<a>=NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.has_additive_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
(Fractional ideal (2*a + 1), True)]
```

has_bad_reduction(P)

Return True if this elliptic curve has bad reduction at the prime P.

INPUT:

- `P` – a prime ideal of the base field of self, or a field element generating such an ideal.

OUTPUT:

(bool) True if the curve has bad reduction at P, else False.

Note: This requires determining a local integral minimal model; we do not just check that the discriminant of the current model has valuation zero.

EXAMPLES:

```
sage: E=EllipticCurve('14a1')
sage: [(p,E.has_bad_reduction(p)) for p in prime_range(15)]
```

```
[(2, True), (3, False), (5, False), (7, True), (11, False), (13, False)]

sage: K.<a>=NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.has_bad_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
(Fractional ideal (2*a + 1), True)]
```

has_good_reduction(*P*)

Return True if this elliptic curve has good reduction at the prime P.

INPUT:

•P – a prime ideal of the base field of self, or a field element generating such an ideal.

OUTPUT:

(bool) – True if the curve has good reduction at P, else False.

Note: This requires determining a local integral minimal model; we do not just check that the discriminant of the current model has valuation zero.

EXAMPLES:

```
sage: E=EllipticCurve('14a1')
sage: [(p,E.has_good_reduction(p)) for p in prime_range(15)]
[(2, False), (3, True), (5, True), (7, False), (11, True), (13, True)]

sage: K.<a>=NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.has_good_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), True),
(Fractional ideal (2*a + 1), False)]
```

has_multiplicative_reduction(*P*)

Return True if this elliptic curve has (bad) multiplicative reduction at the prime P.

Note: See also has_split_multiplicative_reduction() and has_nonsplit_multiplicative_reduction().

INPUT:

•**P – a prime ideal of the base field of self, or a field** element generating such an ideal.

OUTPUT:

(bool) True if the curve has multiplicative reduction at P, else False.

EXAMPLES:

```
sage: E=EllipticCurve('14a1')
sage: [(p,E.has_multiplicative_reduction(p)) for p in prime_range(15)]
[(2, True), (3, False), (5, False), (7, True), (11, False), (13, False)]

sage: K.<a>=NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.has_multiplicative_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False), (Fractional ideal (2*a + 1), False)]
```

has_nonsplit_multiplicative_reduction(*P*)

Return True if this elliptic curve has (bad) non-split multiplicative reduction at the prime P.

INPUT:

•P – a prime ideal of the base field of self, or a field element generating such an ideal.

OUTPUT:

(bool) True if the curve has non-split multiplicative reduction at P, else False.

EXAMPLES:

```
sage: E=EllipticCurve('14a1')
sage: [(p,E.has_nonsplit_multiplicative_reduction(p)) for p in prime_range(15)]
[(2, True), (3, False), (5, False), (7, False), (11, False), (13, False)]

sage: K.<a>=NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.has_nonsplit_multiplicative_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False), (Fractional ideal (2*a + 1), False)]
```

has_split_multiplicative_reduction(*P*)

Return True if this elliptic curve has (bad) split multiplicative reduction at the prime P.

INPUT:

- P – a prime ideal of the base field of self, or a field element generating such an ideal.

OUTPUT:

(bool) True if the curve has split multiplicative reduction at P, else False.

EXAMPLES:

```
sage: E=EllipticCurve('14a1')
sage: [(p,E.has_split_multiplicative_reduction(p)) for p in prime_range(15)]
[(2, False), (3, False), (5, False), (7, True), (11, False), (13, False)]

sage: K.<a>=NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.has_split_multiplicative_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False), (Fractional ideal (2*a + 1), False)]
```

integral_model()

Return a model of self which is integral at all primes.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2+1)
sage: E = EllipticCurve([i/5,i/5,i/5,i/5,i/5])
sage: P1,P2 = K.primes_above(5)
sage: E.global_integral_model()
Elliptic Curve defined by y^2 + (-i)*x*y + (-25*i)*y = x^3 + 5*i*x^2 + 125*i*x + 3125*i over
```

is_global_integral_model()

Return true iff self is integral at all primes.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2+1)
sage: E = EllipticCurve([i/5,i/5,i/5,i/5,i/5])
sage: P1,P2 = K.primes_above(5)
sage: Emin = E.global_integral_model()
sage: Emin.is_global_integral_model()
True
```

is_local_integral_model(**P*)

Tests if self is integral at the prime ideal P, or at all the primes if P is a list or tuple.

INPUT:

- *P – a prime ideal, or a list or tuple of primes.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2+1)
sage: P1,P2 = K.primes_above(5)
sage: E = EllipticCurve([i/5,i/5,i/5,i/5,i/5])
sage: E.is_local_integral_model(P1,P2)
False
sage: Emin = E.local_integral_model(P1,P2)
sage: Emin.is_local_integral_model(P1,P2)
True
```

kodaira_symbol(*P, proof=None*)

Returns the Kodaira Symbol of this elliptic curve at the prime P.

INPUT:

- `P` – either None or a prime ideal of the base field of self.
- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.

OUTPUT:

(string) The Kodaira Symbol of the curve at P.

EXAMPLES:

```
sage: K.<a>=NumberField(x^2-5)
sage: E=EllipticCurve([20, 225, 750, 625*a + 6875, 31250*a + 46875])
sage: bad_primes = E.discriminant().support(); bad_primes
[Fractional ideal (7/2*a - 81/2),
Fractional ideal (a + 52),
Fractional ideal (-a),
Fractional ideal (2)]
sage: [E.kodaira_symbol(P) for P in bad_primes]
[I1, I1, I0, II]
sage: K.<a> = QuadraticField(-11)
sage: E = EllipticCurve('11a1').change_ring(K)
sage: [E.kodaira_symbol(P) for P in K(11).support()]
[I10]
```

local_data(*P=None, proof=None*)

Local data for this elliptic curve at the prime P.

INPUT:

- `P` – either None or a prime ideal of the base field of self.
- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.

OUTPUT:

If P is specified, returns the `EllipticCurveLocalData` object associated to the prime P for this curve. Otherwise, returns a list of such objects, one for each prime P in the support of the discriminant of this model.

Note: The model is not required to be integral on input.

For principal P, a generator is used as a uniformizer, and integrality or minimality at other primes is not affected. For non-principal P, the minimal model returned will preserve integrality at other primes, but not minimality.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2+1)
sage: E = EllipticCurve([1 + i, 0, 1, 0, 0])
```

```
sage: E.local_data()
[Local data at Fractional ideal (-3*i - 2):
Reduction type: bad split multiplicative
Local minimal model: Elliptic Curve defined by y^2 + (i+1)*x*y + y = x^3 over Number Field i
Minimal discriminant valuation: 2
Conductor exponent: 1
Kodaira Symbol: I2
Tamagawa Number: 2, Local data at Fractional ideal (2*i + 1):
Reduction type: bad non-split multiplicative
Local minimal model: Elliptic Curve defined by y^2 + (i+1)*x*y + y = x^3 over Number Field i
Minimal discriminant valuation: 1
Conductor exponent: 1
Kodaira Symbol: I1
Tamagawa Number: 1]
sage: E.local_data(K.ideal(3))
Local data at Fractional ideal (3):
Reduction type: good
Local minimal model: Elliptic Curve defined by y^2 + (i+1)*x*y + y = x^3 over Number Field i
Minimal discriminant valuation: 0
Conductor exponent: 0
Kodaira Symbol: I0
Tamagawa Number: 1
```

An example raised in #3897:

```
sage: E = EllipticCurve([1,1])
sage: E.local_data(3)
Local data at Principal ideal (3) of Integer Ring:
Reduction type: good
Local minimal model: Elliptic Curve defined by y^2 = x^3 + x + 1 over Rational Field
Minimal discriminant valuation: 0
Conductor exponent: 0
Kodaira Symbol: I0
Tamagawa Number: 1
```

local_information(*P=None, proof=None*)

code{local_information} has been renamed code{local_data} and is being deprecated.

local_integral_model(**P*)

Return a model of self which is integral at the prime ideal P.

Note: The integrality at other primes is not affected, even if P is non-principal.

INPUT:

- `*P` – a prime ideal, or a list or tuple of primes.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2+1)
sage: P1,P2 = K.primes_above(5)
sage: E = EllipticCurve([i/5,i/5,i/5,i/5,i/5])
sage: E.local_integral_model((P1,P2))
Elliptic Curve defined by y^2 + (-i)*x*y + (-25*i)*y = x^3 + 5*i*x^2 + 125*i*x + 3125*i over
```

local_minimal_model(*P, proof=None*)

Returns a model which is integral at all primes and minimal at P.

INPUT:

- `P` – either None or a prime ideal of the base field of self.
- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.

OUTPUT:

A model of the curve which is minimal (and integral) at P.

Note: The model is not required to be integral on input.

For principal P, a generator is used as a uniformizer, and integrality or minimality at other primes is not affected. For non-principal P, the minimal model returned will preserve integrality at other primes, but not minimality.

EXAMPLES:

```
sage: K.<a>=NumberField(x^2-5)
sage: E=EllipticCurve([20, 225, 750, 625*a + 6875, 31250*a + 46875])
sage: P=K.ideal(a)
sage: E.local_minimal_model(P).ainvs()
[0, 1, 0, a - 33, -2*a + 64]
```

period_lattice(*embedding*)

Returns the period lattice of the elliptic curve for the given embedding of its base field.

INPUT:

- `embedding` - an embedding of the base number field into $\mathbf{R}$ or $\mathbf{C}$.

Note: The precision of the embedding is ignored: we only use the given embedding to determine which embedding into `QQbar` to use. Once the lattice has been initialized, periods can be computed to arbitrary precision.

EXAMPLES:

First define a field with two real embeddings:

```
sage: K.<a> = NumberField(x^3-2)
sage: E=EllipticCurve([0,0,0,a,2])
sage: embs=K.embeddings(CC); len(embs)
3
```

For each embedding we have a different period lattice:

```
sage: E.period_lattice(embs[0])
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + a*x + 2 over Number Field
From: Number Field in a with defining polynomial x^3 - 2
To:   Algebraic Field
Defn: a |--> -0.6299605249474365? - 1.091123635971722?*I

sage: E.period_lattice(embs[1])
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + a*x + 2 over Number Field
From: Number Field in a with defining polynomial x^3 - 2
To:   Algebraic Field
Defn: a |--> -0.6299605249474365? + 1.091123635971722?*I

sage: E.period_lattice(embs[2])
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + a*x + 2 over Number Field
From: Number Field in a with defining polynomial x^3 - 2
To:   Algebraic Field
Defn: a |--> 1.259921049894873?
```

Although the original embeddings have only the default precision, we can obtain the basis with higher precision later:

```
sage: L=E.period_lattice(embs[0])
sage: L.basis()
(1.86405007647981 - 0.903761485143226*I, -0.149344633143919 - 2.06619546272945*I)

sage: L.basis(prec=100)
(1.8640500764798108425920506200 - 0.90376148514322594749786960975*I, -0.14934463314391922099
```

reduction(*place*)

Return the reduction of the elliptic curve at a place of good reduction.

INPUT:

- `place` – a prime ideal in the base field of the curve

OUTPUT:

An elliptic curve over a finite field, the residue field of the place.

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: EK = EllipticCurve([0,0,0,i,i+3])
sage: v = K.fractional_ideal(2*i+3)
sage: EK.reduction(v)
Elliptic Curve defined by y^2  = x^3 + 5*x + 8 over Residue field of Fractional ideal (2*i +
sage: EK.reduction(K.ideal(1+i))
...
AttributeError: The curve must have good reduction at the place.
sage: EK.reduction(K.ideal(2))
...
AttributeError: The ideal must be prime.
```

simon_two_descent(*verbose=0, lim1=5, lim3=50, limtriv=10, maxprob=20, limbigprime=30*)

Computes lower and upper bounds on the rank of the Mordell-Weil group, and a list of independent points.

INPUT:

- `verbose` – 0, 1, 2, or 3 (default: 0), the verbosity level
- `lim1` – (default: 5) limit on trivial points on quartics
- `lim3` – (default: 50) limit on points on ELS quartics
- `limtriv` – (default: 10) limit on trivial points on elliptic curve
- `maxprob` – (default: 20)
- `limbigprime` – (default: 30) to distinguish between small and large prime numbers. Use probabilistic tests for large primes. If 0, don't use probabilistic tests.

OUTPUT:

`(lower, upper, list)` where `lower` is a lower bound on the rank, `upper` is an upper bound (the 2-Selmer rank) and `list` is a list of independent points on the Weierstrass model. The length of `list` is equal to either `lower`, or `lower-1`, since when `lower` is less than `upper` and of different parity, the value of `lower` is increased by 1.

Note: For non-quadratic number fields, this code does return, but it takes a long time.

IMPLEMENTATION:

Uses Denis Simon's GP/PARI scripts from url{http://www.math.unicaen.fr/~simon/}.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 + 23, 'a')
sage: E = EllipticCurve(K, '37')
sage: E == loads(dumps(E))
True
sage: E.simon_two_descent()
(2, 2, [(-1 : 0 : 1), (1/2*a - 5/2 : -1/2*a - 13/2 : 1)])

sage: K.<a> = NumberField(x^2 + 7, 'a')
sage: E = EllipticCurve(K, [0,0,0,1,a]); E
Elliptic Curve defined by y^2  = x^3 + x + a over Number Field in a with defining polynomial
sage: v = E.simon_two_descent(verbose=1); v
courbe elliptique : Y^2 = x^3 + Mod(1, y^2 + 7)*x + Mod(y, y^2 + 7)
A = 0
```

```
B = Mod(1, y^2 + 7)
C = Mod(y, y^2 + 7)
LS2gen = [Mod(Mod(-5, y^2 + 7)*x^2 + Mod(-3*y, y^2 + 7)*x + Mod(8, y^2 + 7), x^3 + Mod(1, y^
#LS2gen = 2
 Recherche de points triviaux sur la courbe
points triviaux sur la courbe = [[1, 1, 0], [Mod(1/2*y + 3/2, y^2 + 7), Mod(-y - 2, y^2 + 7)
zc = Mod(Mod(-5, y^2 + 7)*x^2 + Mod(-3*y, y^2 + 7)*x + Mod(8, y^2 + 7), x^3 + Mod(1, y^2 + 7
 symbole de Hilbert (Mod(2, y^2 + 7),Mod(-5, y^2 + 7)) = -1
 zc = Mod(Mod(1, y^2 + 7)*x^2 + Mod(1/2*y - 1/2, y^2 + 7)*x + Mod(-1, y^2 + 7), x^3 + Mod(1,
 symbole de Hilbert (Mod(-2*y + 2, y^2 + 7),Mod(1, y^2 + 7)) = 0
 sol de Legendre = [1, 0, 1]~
 zc*z1^2 = Mod(Mod(2*y - 2, y^2 + 7)*x + Mod(2*y + 10, y^2 + 7), x^3 + Mod(1, y^2 + 7)*x + M
 quartique : (-1/2*y + 1/2)*Y^2 = x^4 + (-3*y - 15)*x^2 + (-8*y - 16)*x + (-11/2*y - 15/2)
 reduite: Y^2 = (-1/2*y + 1/2)*x^4 - 4*x^3 + (-3*y + 3)*x^2 + (2*y - 2)*x + (1/2*y + 3/2)
 non ELS en [2, [0, 1]~, 1, 1, [1, 1]~]
zc = Mod(Mod(1, y^2 + 7)*x^2 + Mod(1/2*y + 1/2, y^2 + 7)*x + Mod(-1, y^2 + 7), x^3 + Mod(1,
 vient du point trivial [Mod(1/2*y + 3/2, y^2 + 7), Mod(-y - 2, y^2 + 7), 1]
m1 = 1
m2 = 1
#S(E/K)[2]    = 2
#E(K)/2E(K)   = 2
#Ш(E/K)[2]    = 1
rang(E/K)     = 1
listpointsmwr = [[Mod(1/2*y + 3/2, y^2 + 7), Mod(-y - 2, y^2 + 7), 1]]
(1, 1, [(1/2*a + 3/2 : -a - 2 : 1)])
```

A curve with 2-torsion:

```
sage: K.<a> = NumberField(x^2 + 7, 'a')
sage: E = EllipticCurve(K, '15a')
sage: v = E.simon_two_descent(); v  # long time (about 10 seconds), points can vary
(1, 3, [...])
```

tamagawa_exponent(*P*, *proof=None*)

Returns the Tamagawa index of this elliptic curve at the prime P.

INPUT:

- `P` – either None or a prime ideal of the base field of self.
- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.

OUTPUT:

(positive integer) The Tamagawa index of the curve at P.

EXAMPLES:

```
sage: K.<a>=NumberField(x^2-5)
sage: E=EllipticCurve([20, 225, 750, 625*a + 6875, 31250*a + 46875])
sage: [E.tamagawa_exponent(P) for P in E.discriminant().support()]
[1, 1, 1, 1]
sage: K.<a> = QuadraticField(-11)
sage: E = EllipticCurve('11a1').change_ring(K)
sage: [E.tamagawa_exponent(P) for P in K(11).support()]
[10]
```

tamagawa_number(*P*, *proof=None*)

Returns the Tamagawa number of this elliptic curve at the prime P.

INPUT:

- `P` – either None or a prime ideal of the base field of self.

- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.

OUTPUT:

(positive integer) The Tamagawa number of the curve at P.

EXAMPLES:

```
sage: K.<a>=NumberField(x^2-5)
sage: E=EllipticCurve([20, 225, 750, 625*a + 6875, 31250*a + 46875])
sage: [E.tamagawa_number(P) for P in E.discriminant().support()]
[1, 1, 1, 1]
sage: K.<a> = QuadraticField(-11)
sage: E = EllipticCurve('11a1').change_ring(K)
sage: [E.tamagawa_number(P) for P in K(11).support()]
[10]
```

torsion_order()

Returns the order of the torsion subgroup of this elliptic curve.

OUTPUT:

(integer) the order of the torsion subgroup of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: K.<t> = NumberField(x^4 + x^3 + 11*x^2 + 41*x + 101)
sage: EK = E.base_extend(K)
sage: EK.torsion_order()
25

sage: E = EllipticCurve('15a1')
sage: K.<t> = NumberField(x^2 + 2*x + 10)
sage: EK = E.base_extend(K)
sage: EK.torsion_order()
16

sage: E = EllipticCurve('19a1')
sage: K.<t> = NumberField(x^9-3*x^8-4*x^7+16*x^6-3*x^5-21*x^4+5*x^3+7*x^2-7*x+1)
sage: EK = E.base_extend(K)
sage: EK.torsion_order()
9

sage: K.<i> = QuadraticField(-1)
sage: EK = EllipticCurve([0,0,0,i,i+3])
sage: EK.torsion_order()
1
```

torsion_points()

Returns a list of the torsion points of this elliptic curve.

OUTPUT:

(list) A sorted list of the torsion points.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.torsion_points()
[(0 : 1 : 0), (5 : -6 : 1), (5 : 5 : 1), (16 : -61 : 1), (16 : 60 : 1)]
sage: K.<t> = NumberField(x^4 + x^3 + 11*x^2 + 41*x + 101)
sage: EK = E.base_extend(K)
sage: EK.torsion_points()
```

```
[(t : 1/11*t^3 + 6/11*t^2 + 19/11*t + 48/11 : 1),
(1/11*t^3 - 5/11*t^2 + 19/11*t - 40/11 : -6/11*t^3 - 3/11*t^2 - 26/11*t - 321/11 : 1),
(1/11*t^3 - 5/11*t^2 + 19/11*t - 40/11 : 6/11*t^3 + 3/11*t^2 + 26/11*t + 310/11 : 1),
(t : -1/11*t^3 - 6/11*t^2 - 19/11*t - 59/11 : 1),
(16 : 60 : 1),
(-3/55*t^3 - 7/55*t^2 - 2/55*t - 133/55 : 6/55*t^3 + 3/55*t^2 + 25/11*t + 156/55 : 1),
(14/121*t^3 - 15/121*t^2 + 90/121*t + 232/121 : 16/121*t^3 - 69/121*t^2 + 293/121*t - 46/121
(-26/121*t^3 + 20/121*t^2 - 219/121*t - 995/121 : -15/121*t^3 - 156/121*t^2 + 232/121*t - 28
(10/121*t^3 + 49/121*t^2 + 168/121*t + 73/121 : -32/121*t^3 - 60/121*t^2 + 261/121*t + 686/1
(5 : 5 : 1),
(-9/121*t^3 - 21/121*t^2 - 127/121*t - 377/121 : -7/121*t^3 + 24/121*t^2 + 197/121*t + 16/12
(3/55*t^3 + 7/55*t^2 + 2/55*t + 78/55 : 7/55*t^3 - 24/55*t^2 + 9/11*t + 17/55 : 1),
(-5/121*t^3 + 36/121*t^2 - 84/121*t + 24/121 : -34/121*t^3 + 27/121*t^2 - 305/121*t - 829/12
(5/121*t^3 - 14/121*t^2 - 158/121*t - 453/121 : 49/121*t^3 + 129/121*t^2 + 315/121*t + 86/12
(5 : -6 : 1),
(5/121*t^3 - 14/121*t^2 - 158/121*t - 453/121 : -49/121*t^3 - 129/121*t^2 - 315/121*t - 207/
(-5/121*t^3 + 36/121*t^2 - 84/121*t + 24/121 : 34/121*t^3 - 27/121*t^2 + 305/121*t + 708/121
(3/55*t^3 + 7/55*t^2 + 2/55*t + 78/55 : -7/55*t^3 + 24/55*t^2 - 9/11*t - 72/55 : 1),
(-9/121*t^3 - 21/121*t^2 - 127/121*t - 377/121 : 7/121*t^3 - 24/121*t^2 - 197/121*t - 137/12
(16 : -61 : 1),
(10/121*t^3 + 49/121*t^2 + 168/121*t + 73/121 : 32/121*t^3 + 60/121*t^2 - 261/121*t - 807/12
(-26/121*t^3 + 20/121*t^2 - 219/121*t - 995/121 : 15/121*t^3 + 156/121*t^2 - 232/121*t + 276
(14/121*t^3 - 15/121*t^2 + 90/121*t + 232/121 : -16/121*t^3 + 69/121*t^2 - 293/121*t - 75/12
(-3/55*t^3 - 7/55*t^2 - 2/55*t - 133/55 : -6/55*t^3 - 3/55*t^2 - 25/11*t - 211/55 : 1),
(0 : 1 : 0)]
```

```
sage: E = EllipticCurve('15a1')
sage: K.<t> = NumberField(x^2 + 2*x + 10)
sage: EK = E.base_extend(K)
sage: EK.torsion_points()
[(t : t - 5 : 1),
(-1 : 0 : 1),
(t : -2*t + 4 : 1),
(8 : 18 : 1),
(1/2 : 5/4*t + 1/2 : 1),
(-2 : 3 : 1),
(-7 : 5*t + 8 : 1),
(3 : -2 : 1),
(-t - 2 : 2*t + 8 : 1),
(-13/4 : 9/8 : 1),
(-t - 2 : -t - 7 : 1),
(8 : -27 : 1),
(-7 : -5*t - 2 : 1),
(-2 : -2 : 1),
(1/2 : -5/4*t - 2 : 1),
(0 : 1 : 0)]
```

```
sage: K.<i> = QuadraticField(-1)
sage: EK = EllipticCurve(K,[0,0,0,0,-1])
sage: EK.torsion_points ()
[(-2 : -3*i : 1),
(0 : -i : 1),
(1 : 0 : 1),
(0 : i : 1),
(-2 : 3*i : 1),
(0 : 1 : 0)]
```

torsion_subgroup()

Returns the torsion subgroup of this elliptic curve.

OUTPUT:

(`EllipticCurveTorsionSubgroup`) The `EllipticCurveTorsionSubgroup` associated to this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: K.<t>=NumberField(x^4 + x^3 + 11*x^2 + 41*x + 101)
sage: EK=E.base_extend(K)
sage: tor = EK.torsion_subgroup()
sage: tor
Torsion Subgroup isomorphic to Multiplicative Abelian Group isomorphic to C5 x C5 associated
sage: tor.gens()
((16 : 60 : 1), (t : 1/11*t^3 + 6/11*t^2 + 19/11*t + 48/11 : 1))

sage: E = EllipticCurve('15a1')
sage: K.<t>=NumberField(x^2 + 2*x + 10)
sage: EK=E.base_extend(K)
sage: EK.torsion_subgroup()
Torsion Subgroup isomorphic to Multiplicative Abelian Group isomorphic to C4 x C4 associated

sage: E = EllipticCurve('19a1')
sage: K.<t>=NumberField(x^9-3*x^8-4*x^7+16*x^6-3*x^5-21*x^4+5*x^3+7*x^2-7*x+1)
sage: EK=E.base_extend(K)
sage: EK.torsion_subgroup()
Torsion Subgroup isomorphic to Multiplicative Abelian Group isomorphic to C9 associated to t

sage: K.<i> = QuadraticField(-1)
sage: EK = EllipticCurve([0,0,0,i,i+3])
sage: EK.torsion_subgroup ()
Torsion Subgroup isomorphic to Trivial Abelian Group associated to the Elliptic Curve define
```

38.10 Elliptic curves over finite fields

AUTHORS:

- William Stein (2005): Initial version
- Robert Bradshaw et al....
- John Cremona (2008-02): Point counting and group structure for non-prime fields, Frobenius endomorphism and order, elliptic logs

class **`EllipticCurve_finite_field`**(*x, y=None*)

Elliptic curve over a finite field.

`abelian_group`(*debug=False*)

Returns the abelian group structure of the group of points on this elliptic curve.

> **Warning:** The algorithm is definitely *not* intended for use with *large* finite fields! The factorization of the orders of elements must be feasible. Also, baby-step-giant-step methods are used which have space and time requirements which are $O(\sqrt{q})$.

Also, the algorithm uses random points on the curve and hence the generators are likely to differ from one run to another; but the group is cached so the generators will not change in any one run of Sage.

Note: This function applies to elliptic curves over arbitrary finite fields. The related function abelian_group_prime_field() uses the pari script, for prime fields only; it is now obsolete

INPUT:

•`debug` - (default: False): if True, print debugging messages

OUTPUT:

•an abelian group

•tuple of images of each of the generators of the abelian group as points on this curve

AUTHORS:

•John Cremona

EXAMPLES:

```
sage: E=EllipticCurve(GF(11),[2,5])
sage: E.abelian_group()
(Multiplicative Abelian Group isomorphic to C10, ...

sage: E=EllipticCurve(GF(41),[2,5])
sage: E.abelian_group()
(Multiplicative Abelian Group isomorphic to C22 x C2, ...

sage: F.<a>=GF(3^6,'a')
sage: E=EllipticCurve([a^4 + a^3 + 2*a^2 + 2*a, 2*a^5 + 2*a^3 + 2*a^2 + 1])
sage: E.abelian_group()
(Multiplicative Abelian Group isomorphic to C26 x C26, ...

sage: F.<a>=GF(101^3,'a')
sage: E=EllipticCurve([2*a^2 + 48*a + 27, 89*a^2 + 76*a + 24])
sage: E.abelian_group()
(Multiplicative Abelian Group isomorphic to C1031352, ...
```

The group can be trivial:

```
sage: E=EllipticCurve(GF(2),[0,0,1,1,1])
sage: E.abelian_group()
(Trivial Abelian Group, ())
```

Of course, there are plenty of points if we extend the field:

```
sage: E.cardinality(extension_degree=100)
1267650600228231653296516890625
```

This tests the patch for trac #3111, using 10 primes randomly selected:

```
sage: E = EllipticCurve('389a')
sage: for p in [5927, 2297, 1571, 1709, 3851, 127, 3253, 5783, 3499, 4817]:
...       G = E.change_ring(GF(p)).abelian_group()
sage: for p in prime_range(10000):            #long time (~20s)
...       if p != 389:
...           G=E.change_ring(GF(p)).abelian_group()
```

This tests that the bug reported in trac #3926 has been fixed:

```
sage: K.<i> = QuadraticField(-1)
sage: OK = K.ring_of_integers()
sage: P=K.factor(10007)[0][0]
sage: OKmodP = OK.residue_field(P)
sage: E = EllipticCurve([0,0,0,i,i+3])
sage: Emod = E.change_ring(OKmodP); Emod
Elliptic Curve defined by y^2  = x^3 + ibar*x + (ibar+3) over Residue field in ibar of Fract
```

```
sage: Emod.abelian_group() #random generators
(Multiplicative Abelian Group isomorphic to C50067594 x C2,
((3152*ibar + 7679 : 7330*ibar + 7913 : 1), (8466*ibar + 1770 : 0 : 1)))
```

cardinality(*algorithm='heuristic', extension_degree=1*)

Return the number of points on this elliptic curve over an extension field (default: the base field).

INPUT:

- `algorithm` - string (default: 'heuristic'), used only for point counting over prime fields
  - `'heuristic'` - use a heuristic to choose between pari, bsgs and sea.
  - `'pari'` - use the baby step giant step method as implemented in PARI via the C-library function ellap.
  - `'sea'` - use sea.gp as implemented in PARI by Christophe Doche and Sylvain Duquesne. ('sea' stands for 'Schoof-Elkies-Atkin'.)
  - `bsgs` - use the baby step giant step method as implemented in Sage, with the Cremona - Sutherland version of Mestre's trick.
  - `all` - (over prime fields only) compute cardinality with all of pari, sea and bsgs; return result if they agree or raise a RuntimeError if they do not.
- `early_abort` - bool (default: False); this is used only by sea. if True, stop early if a small factor of the order is found.
- `extension_degree` - int (default: 1); if the base field is $k = GF(p^n)$ and extension_degree=d, returns the cardinality of $E(GF(p^{nd}))$.

OUTPUT: an integer

The cardinality is cached.

Over prime fields, one of the above algorithms is used. Over non-prime fields, the serious point counting is done on a standard curve with the same j-invariant over the field GF(p)(j), then lifted to the base_field, and finally account is taken of twists.

For j=0 and j=1728 special formulas are used instead.

EXAMPLES:

```
sage: EllipticCurve(GF(4,'a'),[1,2,3,4,5]).cardinality()
8
sage: k.<a> = GF(3^3)
sage: l = [a^2 + 1, 2*a^2 + 2*a + 1, a^2 + a + 1, 2, 2*a]
sage: EllipticCurve(k,l).cardinality()
29

sage: l = [1, 1, 0, 2, 0]
sage: EllipticCurve(k,l).cardinality()
38
```

An even bigger extension (which we check against Magma):

```
sage: EllipticCurve(GF(3^100,'a'),[1,2,3,4,5]).cardinality()
515377520732011331036459693969645888996929981504
sage: magma.eval("Order(EllipticCurve([GF(3^100)|1,2,3,4,5]))")     # optional - magma
'515377520732011331036459693969645888996929981504'

sage: EllipticCurve(GF(10007),[1,2,3,4,5]).cardinality()
10076
sage: EllipticCurve(GF(10007),[1,2,3,4,5]).cardinality(algorithm='sea')
10076
sage: EllipticCurve(GF(10007),[1,2,3,4,5]).cardinality(algorithm='pari')
10076
sage: EllipticCurve(GF(next_prime(10**20)),[1,2,3,4,5]).cardinality(algorithm='sea')
100000000011093199520
```

The cardinality is cached:

```
sage: E = EllipticCurve(GF(3^100,'a'),[1,2,3,4,5])
sage: E.cardinality() is E.cardinality()
True
sage: E=EllipticCurve(GF(11^2,'a'),[3,3])
sage: E.cardinality()
128
sage: EllipticCurve(GF(11^100,'a'),[3,3]).cardinality()
137806123398222701841183371720896367762643312000384671846835266941791510341065565176497846500
```

cardinality_bsgs(*verbose=False*)

Return the cardinality of self over the base field. Will be called by user function cardinality only when necessary, i.e. when the j_invariant is not in the prime field.
ALGORITHM: A variant of "Mestre's trick" extended to all finite fields by Cremona and Sutherland, 2008.
Note:

1. The Mestre-Schoof-Cremona-Sutherland algorithm may fail for a small finite number of curves over F_q for q at most 49, so for $q < 50$ we use an exhaustive count.
2. Quadratic twists are not implemented in characteristic 2 when $j = 0 (= 1728)$; but this case is treated separately.

EXAMPLES:

```
sage: p=next_prime(10^3)
sage: E=EllipticCurve(GF(p),[3,4])
sage: E.cardinality_bsgs()
1020
sage: E=EllipticCurve(GF(3^4,'a'),[1,1])
sage: E.cardinality_bsgs()
64
sage: F.<a>=GF(101^3,'a')
sage: E=EllipticCurve([2*a^2 + 48*a + 27, 89*a^2 + 76*a + 24])
sage: E.cardinality_bsgs()
1031352
```

cardinality_exhaustive()

Return the cardinality of self over the base field. Simply adds up the number of points with each x-coordinate: only used for small field sizes!

EXAMPLES:

```
sage: p=next_prime(10^3)
sage: E=EllipticCurve(GF(p),[3,4])
sage: E.cardinality_exhaustive()
1020
sage: E=EllipticCurve(GF(3^4,'a'),[1,1])
sage: E.cardinality_exhaustive()
64
```

cardinality_pari()

Return the cardinality of self over the (prime) base field using pari.

The result is not cached.

EXAMPLES:

```
sage: p=next_prime(10^3)
sage: E=EllipticCurve(GF(p),[3,4])
sage: E.cardinality_pari()
1020
sage: K=GF(next_prime(10^6))
```

```
sage: E=EllipticCurve(K,[1,0,0,1,1])
sage: E.cardinality_pari()
999945
```

TESTS:

```
sage: K.<a>=GF(3^20)
sage: E=EllipticCurve(K,[1,0,0,1,a])
sage: E.cardinality_pari()
...
ValueError: cardinality_pari() only works over prime fields.
sage: E.cardinality()
3486794310
```

cardinality_sea(*early_abort=False*)

Return the cardinality of self over the (prime) base field using sea.

INPUT:

- `early_abort` - bool (default: False). if True, an early abort technique is used and the computation is interrupted as soon as a small divisor of the order is detected. The function then returns 0. This is useful for ruling out curves whose cardinality is divisible by a small prime.

The result is not cached.

EXAMPLES:

```
sage: p=next_prime(10^3)
sage: E=EllipticCurve(GF(p),[3,4])
sage: E.cardinality_sea()
1020
sage: K=GF(next_prime(10^6))
sage: E=EllipticCurve(K,[1,0,0,1,1])
sage: E.cardinality_sea()
999945
```

TESTS:

```
sage: K.<a>=GF(3^20)
sage: E=EllipticCurve(K,[1,0,0,1,a])
sage: E.cardinality_sea()
...
ValueError: cardinality_sea() only works over prime fields.
sage: E.cardinality()
3486794310
```

frobenius()

Return the frobenius of self as an element of a quadratic order

Note: This computes the curve cardinality, which may be time-consuming.

Frobenius is only determined up to conjugacy.

EXAMPLES:

```
sage: E=EllipticCurve(GF(11),[3,3])
sage: E.frobenius()
phi
sage: E.frobenius().minpoly()
x^2 - 4*x + 11
```

For some supersingular curves, Frobenius is in Z:

```
sage: E=EllipticCurve(GF(25,'a'),[0,0,0,0,1])
sage: E.frobenius()
-5
```

frobenius_order()

Return the quadratic order Z[phi] where phi is the Frobenius endomorphism of the elliptic curve

Note: This computes the curve cardinality, which may be time-consuming.

EXAMPLES:

```
sage: E=EllipticCurve(GF(11),[3,3])
sage: E.frobenius_order()
Order in Number Field in phi with defining polynomial x^2 - 4*x + 11
```

For some supersingular curves, Frobenius is in Z and the Frobenius order is Z:

```
sage: E=EllipticCurve(GF(25,'a'),[0,0,0,0,1])
sage: R=E.frobenius_order()
sage: R
Order in Number Field in phi with defining polynomial x + 5
sage: R.degree()
1
```

frobenius_polynomial()

Return the characteristic polynomial of Frobenius.

The Frobenius endomorphism of the elliptic curve has quadratic characteristic polynomial. In most cases this is irreducible and defines an imaginary quadratic order; for some supersingular curves, Frobenius is an integer a and the polynomial is $(x-a)^2$.

Note: This computes the curve cardinality, which may be time-consuming.

EXAMPLES:

```
sage: E=EllipticCurve(GF(11),[3,3])
sage: E.frobenius_polynomial()
x^2 - 4*x + 11
```

For some supersingular curves, Frobenius is in Z and the polynomial is a square:

```
sage: E=EllipticCurve(GF(25,'a'),[0,0,0,0,1])
sage: E.frobenius_polynomial().factor()
(x + 5)^2
```

gens()

Returns a tuple of length up to 2 of points which generate the abelian group of points on this elliptic curve. See abelian_group() for limitations.

The algorithm uses random points on the curve, and hence the generators are likely to differ from one run to another; but they are cached so will be consistent in any one run of Sage.

AUTHORS:

- John Cremona

EXAMPLES:

```
sage: E=EllipticCurve(GF(11),[2,5]) # random output
sage: E.gens()
((0 : 4 : 1),)
sage: EllipticCurve(GF(41),[2,5]).gens() # random output
((21 : 1 : 1), (8 : 0 : 1))
sage: F.<a>=GF(3^6,'a')
sage: E=EllipticCurve([a,a+1])
sage: pts=E.gens()
sage: len(pts)
1
sage: pts[0].order()==E.cardinality()
True
```

order(*algorithm='heuristic', extension_degree=1*)

Return the number of points on this elliptic curve over an extension field (default: the base field).

INPUT:

- `algorithm` - string (default: 'heuristic'), used only for point counting over prime fields
  - `'heuristic'` - use a heuristic to choose between pari, bsgs and sea.
  - `'pari'` - use the baby step giant step method as implemented in PARI via the C-library function ellap.
  - `'sea'` - use sea.gp as implemented in PARI by Christophe Doche and Sylvain Duquesne. ('sea' stands for 'Schoof-Elkies-Atkin'.)
  - `bsgs` - use the baby step giant step method as implemented in Sage, with the Cremona - Sutherland version of Mestre's trick.
  - `all` - (over prime fields only) compute cardinality with all of pari, sea and bsgs; return result if they agree or raise a RuntimeError if they do not.
- `early_abort` - bool (default: False); this is used only by sea. if True, stop early if a small factor of the order is found.
- `extension_degree` - int (default: 1); if the base field is $k = GF(p^n)$ and extension_degree=d, returns the cardinality of $E(GF(p^{nd}))$.

OUTPUT: an integer

The cardinality is cached.

Over prime fields, one of the above algorithms is used. Over non-prime fields, the serious point counting is done on a standard curve with the same j-invariant over the field GF(p)(j), then lifted to the base_field, and finally account is taken of twists.

For j=0 and j=1728 special formulas are used instead.

EXAMPLES:

```
sage: EllipticCurve(GF(4,'a'),[1,2,3,4,5]).cardinality()
8
sage: k.<a> = GF(3^3)
sage: l = [a^2 + 1, 2*a^2 + 2*a + 1, a^2 + a + 1, 2, 2*a]
sage: EllipticCurve(k,l).cardinality()
29

sage: l = [1, 1, 0, 2, 0]
sage: EllipticCurve(k,l).cardinality()
38
```

An even bigger extension (which we check against Magma):

```
sage: EllipticCurve(GF(3^100,'a'),[1,2,3,4,5]).cardinality()
515377520732011331036459693969645888996929981504
sage: magma.eval("Order(EllipticCurve([GF(3^100)|1,2,3,4,5]))")     # optional - magma
'515377520732011331036459693969645888996929981504'

sage: EllipticCurve(GF(10007),[1,2,3,4,5]).cardinality()
10076
sage: EllipticCurve(GF(10007),[1,2,3,4,5]).cardinality(algorithm='sea')
10076
sage: EllipticCurve(GF(10007),[1,2,3,4,5]).cardinality(algorithm='pari')
10076
sage: EllipticCurve(GF(next_prime(10**20)),[1,2,3,4,5]).cardinality(algorithm='sea')
100000000011093199520
```

The cardinality is cached:

```
sage: E = EllipticCurve(GF(3^100,'a'),[1,2,3,4,5])
sage: E.cardinality() is E.cardinality()
```

```
True
sage: E=EllipticCurve(GF(11^2,'a'),[3,3])
sage: E.cardinality()
128
sage: EllipticCurve(GF(11^100,'a'),[3,3]).cardinality()
137806123398222701841183371720896367762643312000384671846835266941791510341065565176497846500
```

plot(**args, **kwds*)

Draw a graph of this elliptic curve over a prime finite field.

INPUT:

- `*args, **kwds` - all other options are passed to the circle graphing primitive.

EXAMPLES:

```
sage: E = EllipticCurve(FiniteField(17), [0,1])
sage: P = plot(E, rgbcolor=(0,0,1))
```

points()

All the points on this elliptic curve. The list of points is cached so subsequent calls are free.

EXAMPLES:

```
sage: p = 5
sage: F = GF(p)
sage: E = EllipticCurve(F, [1, 3])
sage: a_sub_p = E.change_ring(QQ).ap(p); a_sub_p
2

sage: len(E.points())
4
sage: p + 1 - a_sub_p
4
sage: E.points()
[(0 : 1 : 0), (1 : 0 : 1), (4 : 1 : 1), (4 : 4 : 1)]

sage: K = GF(p**2,'a')
sage: E = E.change_ring(K)
sage: len(E.points())
32
sage: (p + 1)**2 - a_sub_p**2
32
sage: w = E.points(); w
[(0 : 1 : 0), (0 : 2*a + 4 : 1), (0 : 3*a + 1 : 1), (1 : 0 : 1), (2 : 2*a + 4 : 1), (2 : 3*a
```

Note that the returned list is an immutable sorted Sequence:

```
sage: w[0] = 9
...
ValueError: object is immutable; please change a copy instead.
```

random_element()

Returns a random point on this elliptic curve.

Returns the point at infinity with probability $1/(q+1)$ where the base field has cardinality q.

EXAMPLES:

```
sage: k = GF(next_prime(7^5))
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P
(751 : 6230 : 1)
sage: type(P)
```

```
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
sage: P in E
True

sage: k.<a> = GF(7^5)
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P
(a^4 + a + 5 : 6*a^4 + 3*a^3 + 2*a^2 + 4 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
sage: P in E
True

sage: k.<a> = GF(2^5)
sage: E = EllipticCurve(k,[a^2,a,1,a+1,1])
sage: P = E.random_element(); P
(a^4 : 0 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
sage: P in E
True
```

random_point()

Returns a random point on this elliptic curve.

Returns the point at infinity with probability $1/(q+1)$ where the base field has cardinality q.

EXAMPLES:

```
sage: k = GF(next_prime(7^5))
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P
(751 : 6230 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
sage: P in E
True

sage: k.<a> = GF(7^5)
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P
(a^4 + a + 5 : 6*a^4 + 3*a^3 + 2*a^2 + 4 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
sage: P in E
True

sage: k.<a> = GF(2^5)
sage: E = EllipticCurve(k,[a^2,a,1,a+1,1])
sage: P = E.random_element(); P
(a^4 : 0 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
sage: P in E
True
```

trace_of_frobenius()

Return the trace of Frobenius acting on this elliptic curve.

Note: This computes the curve cardinality, which may be time-consuming.

EXAMPLES:

```
sage: E=EllipticCurve(GF(101),[2,3])
sage: E.trace_of_frobenius()
6
sage: E=EllipticCurve(GF(11^5,'a'),[2,5])
sage: E.trace_of_frobenius()
802
```

The following shows that the issue from trac #2849 is fixed:

```
sage: E=EllipticCurve(GF(3^5,'a'),[-1,-1])
sage: E.trace_of_frobenius()
-27
```

38.11 Points on elliptic curves

The base class `EllipticCurvePoint_field`, derived from `AdditiveGroupElement`, provides support for points on elliptic curves defined over general fields. The derived classes `EllipticCurvePoint_number_field` and `EllipticCurvePoint_finite_field` provide further support for point on curves defined over number fields (including the rational field $\mathbf{Q}$) and over finite fields. Although there is no special class for points over $\mathbf{Q}$, there is currently greater functionality implemented over $\mathbf{Q}$ than over other number fields.

The class `EllipticCurvePoint`, which is based on `SchemeMorphism_projective_coordinates_ring`, currently has little extra functionality.

EXAMPLES:

An example over $\mathbf{Q}$:

```
sage: E = EllipticCurve('389a1')
sage: P = E(-1,1); P
(-1 : 1 : 1)
sage: Q = E(0,-1); Q
(0 : -1 : 1)
sage: P+Q
(4 : 8 : 1)
sage: P-Q
(1 : 0 : 1)
sage: 3*P-5*Q
(328/361 : -2800/6859 : 1)
```

An example over a number field:

```
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve(K,[1,0,0,0,-1])
sage: P = E(0,i); P
(0 : i : 1)
sage: P.order()
+Infinity
sage: 101*P-100*P==P
True
```

An example over a finite field:

```
sage: K.<a> = GF(101^3)
sage: E = EllipticCurve(K,[1,0,0,0,-1])
sage: P = E(40*a^2 + 69*a + 84 , 58*a^2 + 73*a + 45)
```

```
sage: P.order()
1032210
sage: E.cardinality()
1032210
```

Arithmetic with a point over an extension of a finite field:

```
sage: k.<a> = GF(5^2)
sage: E = EllipticCurve(k,[1,0]); E
Elliptic Curve defined by y^2 = x^3 + x over Finite Field in a of size 5^2
sage: P = E([a,2*a+4])
sage: 5*P
(2*a + 3 : 2*a : 1)
sage: P*5
(2*a + 3 : 2*a : 1)
sage: P + P + P + P + P
(2*a + 3 : 2*a : 1)
```

```
sage: F = Zmod(3)
sage: E = EllipticCurve(F,[1,0]);
sage: P = E([2,1])
sage: import sys
sage: n = sys.maxint
sage: P*(n+1)-P*n == P
True
```

AUTHORS:

- William Stein (2005) – Initial version
- Robert Bradshaw et al....
- **John Cremona (Feb 2008) – Point counting and group structure for** non-prime fields, Frobenius endomorphism and order, elliptic logs
- John Cremona (Aug 2008) – Introduced `EllipticCurvePoint_number_field` class
- Tobias Nagel, Michael Mardaus, John Cremona (Dec 2008) – p-adic elliptic logarithm over $\mathbf{Q}$
- David Hansen (Jan 2009) – Added `weil_pairing` function to `EllipticCurvePoint_finite_field` class

class `EllipticCurvePoint` (*X, v, check=True*)
A point on an elliptic curve.

class `EllipticCurvePoint_field` (*curve, v, check=True*)
A point on an elliptic curve over a field. The point has coordinates in the base field.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E([0,0])
(0 : 0 : 1)
sage: E(0,0)                # brackets are optional
(0 : 0 : 1)
sage: E([GF(5)(0), 0])      # entries are coerced
(0 : 0 : 1)
```

```
sage: E(0.000, 0)
(0 : 0 : 1)

sage: E(1,0,0)
...
TypeError: Coordinates [1, 0, 0] do not define a point on
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field


sage: E = EllipticCurve([0,0,1,-1,0])
sage: S = E(QQ); S
Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field

sage: K.<i>=NumberField(x^2+1)
sage: E=EllipticCurve(K,[0,1,0,-160,308])
sage: P=E(26,-120)
sage: Q=E(2+12*i,-36+48*i)
sage: P.order() == Q.order() == 4
True
sage: 2*P==2*Q
False


sage: K.<t>=FractionField(PolynomialRing(QQ,'t'))
sage: E=EllipticCurve([0,0,0,0,t^2])
sage: P=E(0,t)
sage: P,2*P,3*P
((0 : t : 1), (0 : -t : 1), (0 : 1 : 0))
```

TESTS:

```
sage: loads(S.dumps()) == S
True
sage: E = EllipticCurve('37a')
sage: P = E(0,0); P
(0 : 0 : 1)
sage: loads(P.dumps()) == P
True
sage: T = 100*P
sage: loads(T.dumps()) == T
True
```

Test pickling an elliptic curve that has known points on it:

```
sage: e = EllipticCurve([0, 0, 1, -1, 0]); g = e.gens(); loads(dumps(e)) == e
True
```

codomain()

Return the codomain of this point, which is the curve it is on. Synonymous with curve() which is perhaps more intuitive.

EXAMPLES:

```
sage: E=EllipticCurve(QQ,[1,1])
sage: P=E(0,1)
sage: P.domain()
Spectrum of Rational Field
sage: K.<a>=NumberField(x^2-3,'a')
sage: P=E.base_extend(K)(1,a)
sage: P.codomain()
```

```
Elliptic Curve defined by y^2 = x^3 + x + 1 over Number Field in a with defining polynomial
sage: P.codomain() == P.curve()
True
```

curve()

Return the curve that this point is on.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: P = E([-1,1])
sage: P.curve()
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
```

division_points(*m, poly_only=False*)

Return a list of all points Q such that $mQ = P$ where P = self.

Only points on the elliptic curve containing self and defined over the base field are included.

INPUT:

- m – a positive integer
- poly_only – bool (default: False); if True return polynomial whose roots give all possible x-coordinates of m-th roots of self.

OUTPUT:

(list) – a (possibly empty) list of solutions Q to $mQ = P$, where P = self.

EXAMPLES:

We find the five 5-torsion points on an elliptic curve:

```
sage: E = EllipticCurve('11a'); E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: P = E(0); P
(0 : 1 : 0)
sage: P.division_points(5)
[(0 : 1 : 0), (5 : -6 : 1), (5 : 5 : 1), (16 : -61 : 1), (16 : 60 : 1)]
```

Note above that 0 is included since [5]*0 = 0.

We create a curve of rank 1 with no torsion and do a consistency check:

```
sage: E = EllipticCurve('11a').quadratic_twist(-7)
sage: Q = E([44,-270])
sage: (4*Q).division_points(4)
[(44 : -270 : 1)]
```

We create a curve over a non-prime finite field with group of order 18:

```
sage: k.<a> = GF(25)
sage: E = EllipticCurve(k, [1,2+a,3,4*a,2])
sage: P = E([3,3*a+4])
sage: factor(E.order())
2 * 3^2
sage: P.order()
9
```

We find the 1-division points as a consistency check – there is just one, of course:

```
sage: P.division_points(1)
[(3 : 3*a + 4 : 1)]
```

The point P has order coprime to 2 but divisible by 3, so:

```
sage: P.division_points(2)
[(2*a + 1 : 3*a + 4 : 1), (3*a + 1 : a : 1)]
```

We check that each of the 2-division points works as claimed:

```
sage: [2*Q for Q in P.division_points(2)]
[(3 : 3*a + 4 : 1), (3 : 3*a + 4 : 1)]
```

Some other checks:

```
sage: P.division_points(3)
[]
sage: P.division_points(4)
[(0 : 3*a + 2 : 1), (1 : 0 : 1)]
sage: P.division_points(5)
[(1 : 1 : 1)]
```

An example over a number field (see trac #3383):

```
sage: E = EllipticCurve('19a1')
sage: K.<t> = NumberField(x^9-3*x^8-4*x^7+16*x^6-3*x^5-21*x^4+5*x^3+7*x^2-7*x+1)
sage: EK = E.base_extend(K)
sage: E(0).division_points(3)
[(0 : 1 : 0), (5 : -10 : 1), (5 : 9 : 1)]
sage: EK(0).division_points(3)
[(0 : 1 : 0), (5 : 9 : 1), (5 : -10 : 1)]
sage: E(0).division_points(9)
[(0 : 1 : 0), (5 : -10 : 1), (5 : 9 : 1)]
sage: EK(0).division_points(9)
[(0 : 1 : 0), (5 : 9 : 1), (5 : -10 : 1), (-150/121*t^8 + 414/121*t^7 + 1481/242*t^6 - 2382/
```

domain()

Return the domain of this point, which is $Spec(F)$ where F is the field of definition.

EXAMPLES:

```
sage: E=EllipticCurve(QQ,[1,1])
sage: P=E(0,1)
sage: P.domain()
Spectrum of Rational Field
sage: K.<a>=NumberField(x^2-3,'a')
sage: P=E.base_extend(K)(1,a)
sage: P.domain()
Spectrum of Number Field in a with defining polynomial x^2 - 3
```

has_finite_order()

Return True if this point has finite additive order as an element of the group of points on this curve.

For fields other than number fields and finite fields, this is NotImplemented unless self.is_zero().

EXAMPLES:

```
sage: K.<t>=FractionField(PolynomialRing(QQ,'t'))
sage: E=EllipticCurve([0,0,0,-t^2,0])
sage: P = E(0)
sage: P.has_finite_order()
True
sage: P=E(t,0)
sage: P.has_finite_order()
...
NotImplementedError: Computation of order of a point not implemented over general fields.
sage: (2*P).is_zero()
True
```

has_infinite_order()

Return True if this point has infinite additive order as an element of the group of points on this curve.

For fields other than number fields and finite fields, this is NotImplemented unless self.is_zero().

EXAMPLES:

```
sage: K.<t>=FractionField(PolynomialRing(QQ,'t'))
sage: E=EllipticCurve([0,0,0,-t^2,0])
sage: P = E(0)
sage: P.has_infinite_order()
False
sage: P=E(t,0)
sage: P.has_infinite_order()
...
NotImplementedError: Computation of order of a point not implemented over general fields.
sage: (2*P).is_zero()
True
```

is_divisible_by(*m*)

Return True if there exists a point Q defined over the same field as self such that mQ == self.

INPUT:

- m – a positive integer.

OUTPUT:

(bool) – True if there is a solution, else False.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: Q = 5*E(0,0); Q
(-2739/1444 : -77033/54872 : 1)
sage: Q.is_divisible_by(4)
False
sage: Q.is_divisible_by(5)
True
```

is_finite_order()

Return True if this point has finite additive order as an element of the group of points on this curve.

For fields other than number fields and finite fields, this is NotImplemented unless self.is_zero().

EXAMPLES:

```
sage: K.<t>=FractionField(PolynomialRing(QQ,'t'))
sage: E=EllipticCurve([0,0,0,-t^2,0])
sage: P = E(0)
sage: P.has_finite_order()
True
sage: P=E(t,0)
sage: P.has_finite_order()
...
NotImplementedError: Computation of order of a point not implemented over general fields.
sage: (2*P).is_zero()
True
```

order()

Return the order of this point on the elliptic curve.

If the point is zero, returns 1, otherwise raise a NotImplementedError.

For curves over number fields and finite fields, see below.

EXAMPLE:

```
sage: K.<t>=FractionField(PolynomialRing(QQ,'t'))
sage: E=EllipticCurve([0,0,0,-t^2,0])
sage: P=E(t,0)
sage: P.order()
...
```

```
NotImplementedError: Computation of order of a point not implemented over general fields.
sage: E(0).order() == 1
True
```

plot(***args*)

Plot this point on an elliptic curve.

INPUT:

- `**args` – all arguments get passed directly onto the point plotting function.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: P = E([-1,1])
sage: P.plot(pointsize=30, rgbcolor=(1,0,0))
```

scheme()

Return the scheme of this point, i.e., the curve it is on. This is synonymous with curve() which is perhaps more intuitive.

EXAMPLES:

```
sage: E=EllipticCurve(QQ,[1,1])
sage: P=E(0,1)
sage: P.scheme()
Elliptic Curve defined by y^2 = x^3 + x + 1 over Rational Field
sage: P.scheme() == P.curve()
True
sage: K.<a>=NumberField(x^2-3,'a')
sage: P=E.base_extend(K)(1,a)
sage: P.scheme()
Elliptic Curve defined by y^2 = x^3 + x + 1 over Number Field in a with defining polynomial
```

weil_pairing(*Q*, *n*)

Compute the Weil pairing of self and Q using Miller's algorithm.

INPUT:

- Q – a point on self.curve().
- n – an integer n such that $nP = nQ = (0 : 1 : 0)$ where P = self.

OUTPUT:

An n'th root of unity in the base field self.curve().base_field()

EXAMPLES:

```
sage: F.<a>=GF(2^5)
sage: E=EllipticCurve(F,[0,0,1,1,1])
sage: P = E(a^4 + 1, a^3)
sage: Fx.<b>=GF(2^(4*5))
sage: Ex=EllipticCurve(Fx,[0,0,1,1,1])
sage: phi=Hom(F,Fx)(F.gen().minpoly().roots(Fx)[0][0])
sage: Px=Ex(phi(P.xy()[0]),phi(P.xy()[1]))
sage: O = Ex(0)
sage: Qx = Ex(b^19 + b^18 + b^16 + b^12 + b^10 + b^9 + b^8 + b^5 + b^3 + 1, b^18 + b^13 + b^
sage: Px.weil_pairing(Qx,41) == b^19 + b^15 + b^9 + b^8 + b^6 + b^4 + b^3 + b^2 + 1
True
sage: Px.weil_pairing(17*Px,41) == Fx(1)
True
sage: Px.weil_pairing(O,41) == Fx(1)
True
```

An error is raised if either point is not n-torsion:

```
sage: Px.weil_pairing(O,40)
...
ValueError: points must both be n-torsion
```

A larger example (see trac #4964):

```
sage: P,Q = EllipticCurve(GF(19^4,'a'),[-1,0]).gens()
sage: P.order(), Q.order()
(360, 360)
sage: z = P.weil_pairing(Q,360)
sage: z.multiplicative_order()
360
```

An example over a number field:

```
sage: P,Q = EllipticCurve('11a1').change_ring(CyclotomicField(5)).torsion_subgroup().gens()
sage: (P.order(),Q.order())
(5, 5)
sage: P.weil_pairing(Q,5)
zeta5^2
sage: Q.weil_pairing(P,5)
zeta5^3
```

ALGORITHM:

Implemented using Proposition 8 in [Mil04]. The value 1 is returned for linearly dependent input points. This condition is caught via a DivisionByZeroError, since the use of a discrete logarithm test for linear dependence, is much to slow for large n.

REFERENCES:

[Mil04] Victor S. Miller, "The Weil pairing, and its efficient calculation", J. Cryptol., 17(4):235-261, 2004

AUTHOR:

- David Hansen (2009-01-25)

xy()

Return the x and y coordinates of this point, as a 2-tuple. If this is the point at infinity a ZeroDivisionError is raised.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: P = E([-1,1])
sage: P.xy()
(-1, 1)
sage: Q = E(0); Q
(0 : 1 : 0)
sage: Q.xy()
...
ZeroDivisionError: Rational division by zero
```

class EllipticCurvePoint_finite_field(*curve, v, check=True*)

Class for elliptic curve points over finite fields.

discrete_log(*Q, ord=None*)

Returns discrete log of Q with respect to P =self.

INPUT:

- Q (point) – another point on the same curve as self.
- ord (integer or None (default)) – the order of self.

OUTPUT:

(integer) – The discrete log of Q with respect to P, which is an integer m with $0 \leq m < o(P)$ such that $mP = Q$, if one exists. A ValueError is raised if there is no solution.

Note: The order of self is computed if not supplied.

AUTHOR:

•John Cremona. Adapted to use generic functions 2008-04-05.

EXAMPLE:

```
sage: F=GF(3^6,'a')
sage: a=F.gen()
sage: E= EllipticCurve([0,1,1,a,a])
sage: E.cardinality()
762
sage: A,G=E.abelian_group() ## set since this E is cyclic
sage: P=G[0]
sage: Q=400*P
sage: P.discrete_log(Q)
400
```

order()

Return the order of this point on the elliptic curve.

ALGORITHM:

Use generic functions from `sage.groups.generic`. If the group order is known, use `order_from_multiple()`, otherwise use `order_from_bounds()` with the Hasse bounds for the base field. In the latter case, we might find that we have a generator for the group, in which case it is cached.

We do not cause the group order to be calculated when not known, since this function is used in determining the group order via computation of several random points and their orders.

AUTHOR:

•John Cremona, 2008-02-10, adapted 2008-04-05 to use generic functions.

EXAMPLES:

```
sage: k.<a> = GF(5^5)
sage: E = EllipticCurve(k,[2,4]); E
Elliptic Curve defined by y^2 = x^3 + 2*x + 4 over Finite Field in a of size 5^5
sage: P = E(3*a^4 + 3*a , 2*a + 1 )
sage: P.order()
3227
sage: Q = E(0,2)
sage: Q.order()
7
```

In the next example, the cardinality of E will be computed (using SEA) and cached:

```
sage: p=next_prime(2^150)
sage: E=EllipticCurve(GF(p),[1,1])
sage: P=E(831623307675610677632782670796608848711856078, 42295786042873366706573292533588638
sage: P.order()
1427247692705959881058262545272474300628281448
sage: P.order()==E.cardinality()
True
```

class **EllipticCurvePoint_number_field**(*curve, v, check=True*)

A point on an elliptic curve over a number field.

Most of the functionality is derived from the parent class `EllipticCurvePoint_field`. In addition we have support for the order of a point, and heights (currently only implemented over $\mathbf{Q}$).

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E([0,0])
(0 : 0 : 1)
sage: E(0,0)                # brackets are optional
(0 : 0 : 1)
sage: E([GF(5)(0), 0])      # entries are coerced
(0 : 0 : 1)

sage: E(0.000, 0)
(0 : 0 : 1)

sage: E(1,0,0)
...
TypeError: Coordinates [1, 0, 0] do not define a point on
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field


sage: E = EllipticCurve([0,0,1,-1,0])
sage: S = E(QQ); S
Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

TESTS:

```
sage: loads(S.dumps()) == S
True
sage: P = E(0,0); P
(0 : 0 : 1)
sage: loads(P.dumps()) == P
True
sage: T = 100*P
sage: loads(T.dumps()) == T
True
```

Test pickling an elliptic curve that has known points on it:

```
sage: e = EllipticCurve([0, 0, 1, -1, 0]); g = e.gens(); loads(dumps(e)) == e
True
```

elliptic_logarithm(*embedding=None, precision=100, algorithm='pari'*)

Returns the elliptic logarithm of this elliptic curve point.

An embedding of the base field into $\mathbf{R}$ (with arbitrary precision) may be given; otherwise the first real embedding is used (with the specified precision), if there are any; otherwise a NotImplementedError is raised, since we have not yet implemented the complex elliptic logarithm.

INPUT:

- `embedding`: an embedding of the base field into RR
- `precision`: a positive integer (default 100) setting the number of bits of precision for the computation
- `algorithm`: either 'pari' (default) to use Pari's `ellpointtoz{}`, or 'sage' for a native implementation (currently not very accurate)

ALGORITHM:

See [Co2] Cohen H., A Course in Computational Algebraic Number Theory GTM 138, Springer 1996.

AUTHORS:

- Michael Mardaus (2008-07),
- Tobias Nagel (2008-07) – original version from [Co2].
- John Cremona (2008-07) – revision following eclib code.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: E.discriminant() > 0
True
sage: P = E([-1,1])
sage: P.is_on_identity_component ()
False
sage: P.elliptic_logarithm (precision=96)
0.47934825019021931612953301 01 + 0.985868850775824102211203849...*I
sage: Q=E([3,5])
sage: Q.is_on_identity_component()
True
sage: Q.elliptic_logarithm (precision=96)
1.9311282715425594424885852 20
```

An example with negative discriminant, and a torsion point:

```
sage: E = EllipticCurve('11a1')
sage: E.discriminant() < 0
True
sage: P = E([16,-61])
sage: P.elliptic_logarithm(precision=70)
0.25384186085591068434
sage: E.period_lattice().real_period(prec=70) / P.elliptic_logarithm(precision=70)
5.0000000000000000000
```

A larger example. The default algorithm uses Pari and makes sure the result has the requested precision:

```
sage: E = EllipticCurve([1, 0, 1, -85357462, 303528987048]) #18074g1
sage: P = E([4458713781401/835903744, -64466909836503771/24167649046528, 1])
sage: P.elliptic_logarithm()  # 100 bits
0.27656204014107061464076203097
```

However, the native algorithm 'sage' has trouble with precision in this example:

```
sage: P.elliptic_logarithm(algorithm='sage')  # 100 bits
0.2765620401410710087007...
```

This shows that the bug reported at #4901 has been fixed:

```
sage: E = EllipticCurve("4390c2")
sage: P = E(683762969925/44944,-565388972095220019/9528128)
sage: P.elliptic_logarithm()
0.00025638725886520225353198932529
sage: P.elliptic_logarithm(precision=64)
0.0002563872588652022 54
sage: P.elliptic_logarithm(precision=65)
0.00025638725886520225 35
sage: P.elliptic_logarithm(precision=128)
0.00025638725886520225353198932528666427412
sage: P.elliptic_logarithm(precision=129)
0.00025638725886520225353198932528666427412
sage: P.elliptic_logarithm(precision=256)
0.00025638725886520225353198932528666427411683880083463700150051421280096109363 73
sage: P.elliptic_logarithm(precision=257)
0.00025638725886520225353198932528666427411683880083463700150051421280096109363730
```

has_finite_order()

Return True iff this point has finite order on the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: P = E([0,0]); P
(0 : 0 : 1)
sage: P.has_finite_order()
False

sage: E = EllipticCurve([0,1])
sage: P = E([-1,0])
sage: P.has_finite_order()
True
```

has_good_reduction(*P=None*)

Returns True iff this point has good reduction modulo a prime.

INPUT:

- P – a prime of the base_field of the point's curve, or None (default)

OUTPUT:

(bool) If a prime P of the base field is specified, returns True iff the point has good reduction at P; otherwise, return true if the point has god reduction at all primes in the support of the discriminant of this model.

EXAMPLES:

```
sage: E = EllipticCurve('990e1')
sage: P = E.gen(0); P
(15 : 51 : 1)
sage: [E.has_good_reduction(p) for p in [2,3,5,7]]
[False, False, False, True]
sage: [P.has_good_reduction(p) for p in [2,3,5,7]]
[True, False, True, True]
sage: [E.tamagawa_exponent(p) for p in [2,3,5,7]]
[2, 2, 1, 1]
sage: [(2*P).has_good_reduction(p) for p in [2,3,5,7]]
[True, True, True, True]
sage: P.has_good_reduction()
False
sage: (2*P).has_good_reduction()
True
sage: (3*P).has_good_reduction()
False

sage: K.<i> = NumberField(x^2+1)
sage: E = EllipticCurve(K,[0,1,0,-160,308])
sage: P = E(26,-120)
sage: E.discriminant().support()
[Fractional ideal (i + 1),
Fractional ideal (-i - 2),
Fractional ideal (2*i + 1),
Fractional ideal (3)]
sage: [E.tamagawa_exponent(p) for p in E.discriminant().support()]
[1, 4, 4, 4]
sage: P.has_good_reduction()
False
sage: (2*P).has_good_reduction()
False
sage: (4*P).has_good_reduction()
True
```

has_infinite_order()

Return True iff this point has infinite order on the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: P = E([0,0]); P
(0 : 0 : 1)
sage: P.has_infinite_order()
True

sage: E = EllipticCurve([0,1])
sage: P = E([-1,0])
sage: P.has_infinite_order()
False
```

height (*precision=None*)

The Neron-Tate canonical height of the point.

Currently only implemented for curves defined over $\mathbf{Q}$ (using pari); and for points of finite order.

INPUT:

- `self` – a point on a curve over $\mathbf{Q}$
- `precision` – (int or None (default)): the precision in bits of the result (default real precision if None)

OUTPUT:

The rational number 0, or a nonzero real field element

EXAMPLES:

```
sage: E = EllipticCurve('11a'); E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: P = E([5,5]); P
(5 : 5 : 1)
sage: P.height()
0
sage: Q = 5*P
sage: Q.height()
0

sage: E = EllipticCurve('37a'); E
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: P = E([0,0])
sage: P.height()
0.0511114082399688
sage: P.order()
+Infinity
sage: E.regulator()       # slightly random output
0.051111408239968840

sage: E = EllipticCurve('4602a1'); E
Elliptic Curve defined by y^2 + x*y  = x^3 + x^2 - 37746035*x - 89296920339 over Rational Fi
sage: x = 77985922458974949246858229195945103471590
sage: y = 19575260230015313702261379022151675961965157108920263594545223
sage: d = 2254020761884782243
sage: E([ x / d^2,  y / d^3 ]).height()
86.7406561381275

sage: E = EllipticCurve([17, -60, -120, 0, 0]); E
Elliptic Curve defined by y^2 + 17*x*y - 120*y = x^3 - 60*x^2 over Rational Field
sage: E([30, -90]).height()
0
```

```
sage: E = EllipticCurve('389a1'); E
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
sage: [P,Q] = [E(-1,1),E(0,-1)]
sage: P.height(precision=100)
0.68666708330558658572355210295
sage: (3*Q).height(precision=100)/Q.height(precision=100)
9.0000000000000000000000000000
sage: _.parent()
Real Field with 100 bits of precision
```

An example to show that the bug at #5252 is fixed:

```
sage: E = EllipticCurve([1, -1, 1, -20637587012466263707737269 78, 32838647793306133075103747
sage: P = E([-30987785091199, 258909576181697016447])
sage: P.height()
25.8603170675462
sage: P.height(precision=100)
25.860317067546190743868840741
sage: P.height(precision=250)
25.860317067546190743868840740735110323098872903844416215577171041783572513
sage: P.height(precision=500)
25.860317067546190743868840740735110323098872903844416215577171041783572512955113057088981 32
```

Unfortunately, canonical height is not yet implemented in general:

```
sage: E = EllipticCurve('5077a1').change_ring(QuadraticField(-3,'a'))
sage: P = E([-2,3,1])
sage: P.height()
...
NotImplementedError: canonical height not yet implemented over general number fields.
```

is_on_identity_component(*embedding=None*)

Returns True iff this point is on the identity component of its curve with respect to a given (real or complex) embedding.

INPUT:

- `self` – a point on a curve over any ordered field (e.g. $\mathbf{Q}$)
- `embedding` – an embedding from the base_field of the point's curve into $\mathbf{R}$ or $\mathbf{C}$; if None (the default) it uses the first embedding of the base_field into $\mathbf{R}$ if any, else the first embedding into $\mathbf{C}$.

OUTPUT:

(bool) – True iff the point is on the identity component of the curve. (If the point is zero then the result is True.)

EXAMPLES:

For $K = \mathbf{Q}$ there is no need to specify an embedding:

```
sage: E=EllipticCurve('5077a1')
sage: [E.lift_x(x).is_on_identity_component() for x in range(-3,5)]
[False, False, False, False, False, True, True, True]
```

An example over a field with two real embeddings:

```
sage: L.<a> = QuadraticField(2)
sage: E=EllipticCurve(L,[0,1,0,a,a])
sage: P=E(-1,0)
sage: [P.is_on_identity_component(e) for e in L.embeddings(RR)]
[False, True]
```

We can check this as follows:

```
sage: [e(E.discriminant())>0 for e in L.embeddings(RR)]
[True, False]
sage: e = L.embeddings(RR)[0]
sage: E1 = EllipticCurve(RR,[e(ai) for ai in E.ainvs()])
sage: e1,e2,e3 = E1.two_division_polynomial().roots(RR,multiplicities=False)
sage: e1 < e2 < e3 and e(P[0]) < e3
True
```

order()

Return the order of this point on the elliptic curve.

If the point has infinite order, returns +Infinity. For curves defined over $\mathbf{Q}$, we call pari; over other number fields we implement the function here.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: P = E([0,0]); P
(0 : 0 : 1)
sage: P.order()
+Infinity

sage: E = EllipticCurve([0,1])
sage: P = E([-1,0])
sage: P.order()
2
```

padic_elliptic_logarithm(*p, absprec=20*)

Computes the p-adic elliptic logarithm of this point.

INPUT:

`p` - integer: a prime `absprec` - integer (default: 20): the initial p-adic absolute precision of the computation

OUTPUT:

The p-adic elliptic logarithm of self, with precision `absprec`.

AUTHORS:

- Tobias Nagel
- Michael Mardaus
- John Cremona

ALGORITHM:

For points in the formal group (i.e. not integral at p) we take the `log()` function from the formal groups module and evaluate it at $-x/y$. Otherwise we first multiply the point to get into the formal group, and divide the result afterwards.

TODO:

See comments at trac #4805. Currently the absolute precision of the result may be less than the given value of absprec, and error-handling is imperfect.

EXAMPLES:

```
sage: E = EllipticCurve([0,1,1,-2,0])
sage: E(0).padic_elliptic_logarithm(3)
0
sage: P = E(0,0)
sage: P.padic_elliptic_logarithm(3)
2 + 2*3 + 3^3 + 2*3^7 + 3^8 + 3^9 + 3^11 + 3^15 + 2*3^17 + 3^18 + O(3^19)
sage: P.padic_elliptic_logarithm(3).lift()
660257522
sage: P = E(-11/9,28/27)
```

```
sage: [(2*P).padic_elliptic_logarithm(p)/P.padic_elliptic_logarithm(p) for p in prime_range(
[2 + O(2^19), 2 + O(3^20), 2 + O(5^19), 2 + O(7^19), 2 + O(11^19), 2 + O(13^19), 2 + O(17^19
sage: [(3*P).padic_elliptic_logarithm(p)/P.padic_elliptic_logarithm(p) for p in prime_range(
[1 + 2 + O(2^19), 3 + 3^20 + O(3^21), 3 + O(5^19), 3 + O(7^19), 3 + O(11^19)]
sage: [(5*P).padic_elliptic_logarithm(p)/P.padic_elliptic_logarithm(p) for p in prime_range(
[1 + 2^2 + O(2^19), 2 + 3 + O(3^20), 5 + O(5^19), 5 + O(7^19), 5 + O(11^19)]
```

An example which arose during reviewing #4741:

```
sage: E = EllipticCurve('794a1')
sage: P = E(-1,2)
sage: P.padic_elliptic_logarithm(2) # default precision=20
2^4 + 2^5 + 2^6 + 2^8 + 2^9 + 2^13 + 2^14 + 2^15 + O(2^16)
sage: P.padic_elliptic_logarithm(2, absprec=30)
2^4 + 2^5 + 2^6 + 2^8 + 2^9 + 2^13 + 2^14 + 2^15 + 2^22 + 2^23 + 2^24 + O(2^26)
sage: P.padic_elliptic_logarithm(2, absprec=40)
2^4 + 2^5 + 2^6 + 2^8 + 2^9 + 2^13 + 2^14 + 2^15 + 2^22 + 2^23 + 2^24 + 2^28 + 2^29 + 2^31 +
```

38.12 Torsion subgroups of elliptic curves over number fields (including $\mathbb{Q}$).

AUTHORS:

- Nick Alexander: original implementation over **Q**
- Chris Wuthrich: original implementation over number fields
- **John Cremona: rewrote p-primary part to use division** polynomials, added some features, unified Number Field and **Q** code.

class **EllipticCurveTorsionSubgroup**(*E, algorithm=None*)

The torsion subgroup of an elliptic curve over a number field.

EXAMPLES:

Examples over **Q**:

```
sage: E = EllipticCurve([-4, 0]); E
Elliptic Curve defined by y^2  = x^3 - 4*x over Rational Field
sage: G = E.torsion_subgroup(); G
Torsion Subgroup isomorphic to Multiplicative Abelian Group isomorphic to C2 x C2 associated to
sage: G.order()
4
sage: G.gen(0)
(2 : 0 : 1)
sage: G.gen(1)
(0 : 0 : 1)
sage: G.ngens()
2
```

```
sage: E = EllipticCurve([17, -120, -60, 0, 0]); E
Elliptic Curve defined by y^2 + 17*x*y - 60*y = x^3 - 120*x^2 over Rational Field
sage: G = E.torsion_subgroup(); G
Torsion Subgroup isomorphic to Trivial Abelian Group associated to the Elliptic Curve defined by
sage: G.gens()
()
```

```
sage: e = EllipticCurve([0, 330761566545336520666609946884,0,\
347897536144342179642120321790729023127716119338758604800,\
114112815436927429551902303280680424778815462104985764887003237028585178135281666400000])
sage: e.torsion_order()
16
```

Constructing points from the torsion subgroup (which is an abstract abelian group):

```
sage: E = EllipticCurve('14a1')
sage: T = E.torsion_subgroup()
sage: [E(t) for t in T]
[(0 : 1 : 0),
(9 : 23 : 1),
(2 : 2 : 1),
(1 : -1 : 1),
(2 : -5 : 1),
(9 : -33 : 1)]
```

An example where the torsion subgroup is not cyclic:

```
sage: E = EllipticCurve([0,0,0,-49,0])
sage: T = E.torsion_subgroup()
sage: [E(t) for t in T]
[(0 : 1 : 0), (0 : 0 : 1), (7 : 0 : 1), (-7 : 0 : 1)]
```

An example where the torsion subgroup is trivial:

```
sage: E = EllipticCurve('37a1')
sage: T = E.torsion_subgroup()
sage: T
Torsion Subgroup isomorphic to Trivial Abelian Group associated to the Elliptic Curve defined by
sage: [E(t) for t in T]
[(0 : 1 : 0)]
```

Examples over other Number Fields:

```
sage: E=EllipticCurve('11a1')
sage: K.<i>=NumberField(x^2+1)
sage: EK=E.change_ring(K)
sage: from sage.schemes.elliptic_curves.ell_torsion import EllipticCurveTorsionSubgroup
sage: EllipticCurveTorsionSubgroup(EK)
Torsion Subgroup isomorphic to Multiplicative Abelian Group isomorphic to C5 associated to the E
```

Note: this class is normally constructed indirectly as follows:

```
sage: T = EK.torsion_subgroup(); T
Torsion Subgroup isomorphic to Multiplicative Abelian Group isomorphic to C5 associated to the E
sage: type(T)
<class 'sage.schemes.elliptic_curves.ell_torsion.EllipticCurveTorsionSubgroup'>
```

AUTHORS:

- Nick Alexamder - initial implementation over $\mathbf{Q}$.
- Chris Wuthrich - initial implementation over number fields.
- John Cremona - additional features and unification.

curve()
Return the curve of this torsion subgroup.
EXAMPLES:

```
sage: E=EllipticCurve('11a1')
sage: K.<i>=NumberField(x^2+1)
sage: EK=E.change_ring(K)
sage: T = EK.torsion_subgroup()
sage: T.curve() is EK
True
```

gen(*i=0*)
Return the i'th torsion generator.
EXAMPLES:

```
sage: E=EllipticCurve('11a1')
sage: K.<i>=NumberField(x^2+1)
sage: EK=E.change_ring(K)
sage: T = EK.torsion_subgroup()
sage: T.gen()
(16 : 60 : 1)
```

ngens()
Return the number of torsion generators.
EXAMPLES:

```
sage: E=EllipticCurve('11a1')
sage: K.<i>=NumberField(x^2+1)
sage: EK=E.change_ring(K)
sage: T = EK.torsion_subgroup()
sage: T.ngens()
1
```

points()
Return a list of all the points in this torsion subgroup. The list is cached.
EXAMPLES:

```
sage: K.<i>=NumberField(x^2 + 1)
sage: E = EllipticCurve(K,[0,0,0,1,0])
sage: tor = E.torsion_subgroup()
sage: tor.points()
[(i : 0 : 1), (0 : 0 : 1), (-i : 0 : 1), (0 : 1 : 0)]
```

38.13 Local data for elliptic curves over number fields (including $\mathbb{Q}$) at primes.

AUTHORS:

- John Cremona: First version 2008-09-21 (refactoring code from `ell_number_field.py` and `ell_rational_field.py`)

class EllipticCurveLocalData(*E, P, proof=None, algorithm='pari'*)
The class for the local reduction data of an elliptic curve.

Currently supported are elliptic curves defined over $\mathbf{Q}$, and elliptic curves defined over a number field, at an arbitrary prime or prime ideal.

bad_reduction_type()

Return the type of bad reduction of this reduction data.

OUTPUT:

(int or None):

- +1 for split multiplicative reduction
- -1 for non-split multiplicative reduction
- 0 for additive reduction
- None for good reduction

EXAMPLES:

```
sage: E=EllipticCurve('14a1')
sage: [(p,E.local_data(p).bad_reduction_type()) for p in prime_range(15)]
[(2, -1), (3, None), (5, None), (7, 1), (11, None), (13, None)]

sage: K.<a>=NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.local_data(p).bad_reduction_type()) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), None), (Fractional ideal (2*a + 1), 0)]
```

conductor_valuation()

Return the valuation of the conductor from this local reduction data.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import EllipticCurveLocalData
sage: E = EllipticCurve([0,0,0,0,64]); E
Elliptic Curve defined by y^2 = x^3 + 64 over Rational Field
sage: data = EllipticCurveLocalData(E,2)
sage: data.conductor_valuation()
2
```

has_additive_reduction()

Return True if there is additive reduction.

EXAMPLES:

```
sage: E = EllipticCurve('27a1')
sage: [(p,E.local_data(p).has_additive_reduction()) for p in prime_range(15)]
[(2, False), (3, True), (5, False), (7, False), (11, False), (13, False)]

sage: K.<a> = NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.local_data(p).has_additive_reduction()) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
(Fractional ideal (2*a + 1), True)]
```

has_bad_reduction()

Return True if there is bad reduction.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p,E.local_data(p).has_bad_reduction()) for p in prime_range(15)]
[(2, True), (3, False), (5, False), (7, True), (11, False), (13, False)]

sage: K.<a> = NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
```

```
sage: [(p,E.local_data(p).has_bad_reduction()) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
(Fractional ideal (2*a + 1), True)]
```

has_good_reduction()

Return True if there is good reduction.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p,E.local_data(p).has_good_reduction()) for p in prime_range(15)]
[(2, False), (3, True), (5, True), (7, False), (11, True), (13, True)]

sage: K.<a> = NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.local_data(p).has_good_reduction()) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), True),
(Fractional ideal (2*a + 1), False)]
```

has_multiplicative_reduction()

Return True if there is multiplicative reduction.

Note: See also `has_split_multiplicative_reduction()` and `has_nonsplit_multiplicative_reduction()`.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p,E.local_data(p).has_multiplicative_reduction()) for p in prime_range(15)]
[(2, True), (3, False), (5, False), (7, True), (11, False), (13, False)]

sage: K.<a> = NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.local_data(p).has_multiplicative_reduction()) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False), (Fractional ideal (2*a + 1), False)]
```

has_nonsplit_multiplicative_reduction()

Return True if there is non-split multiplicative reduction.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p,E.local_data(p).has_nonsplit_multiplicative_reduction()) for p in prime_range(15)]
[(2, True), (3, False), (5, False), (7, False), (11, False), (13, False)]

sage: K.<a> = NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.local_data(p).has_nonsplit_multiplicative_reduction()) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False), (Fractional ideal (2*a + 1), False)]
```

has_split_multiplicative_reduction()

Return True if there is split multiplicative reduction.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p,E.local_data(p).has_split_multiplicative_reduction()) for p in prime_range(15)]
[(2, False), (3, False), (5, False), (7, True), (11, False), (13, False)]
```

```
sage: K.<a> = NumberField(x^3-2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p,E.local_data(p).has_split_multiplicative_reduction()) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
(Fractional ideal (2*a + 1), False)]
```

kodaira_symbol()

Return the Kodaira symbol from this local reduction data.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import EllipticCurveLocalData
sage: E = EllipticCurve([0,0,0,0,64]); E
Elliptic Curve defined by y^2 = x^3 + 64 over Rational Field
sage: data = EllipticCurveLocalData(E,2)
sage: data.kodaira_symbol()
IV
```

minimal_model()

Return the (local) minimal model from this local reduction data.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import EllipticCurveLocalData
sage: E = EllipticCurve([0,0,0,0,64]); E
Elliptic Curve defined by y^2  = x^3 + 64 over Rational Field
sage: data = EllipticCurveLocalData(E,2)
sage: data.minimal_model()
Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
sage: data.minimal_model() == E.local_minimal_model(2)
True
```

prime()

Return the prime ideal associated with this local reduction data.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import EllipticCurveLocalData
sage: E = EllipticCurve([0,0,0,0,64]); E
Elliptic Curve defined by y^2 = x^3 + 64 over Rational Field
sage: data = EllipticCurveLocalData(E,2)
sage: data.prime()
Principal ideal (2) of Integer Ring
```

tamagawa_exponent()

Return the Tamagawa index from this local reduction data.

This is the exponent of $E(K_v)/E^0(K_v)$; in most cases it is the same as the Tamagawa index.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import EllipticCurveLocalData
sage: E = EllipticCurve('816a1')
sage: data = EllipticCurveLocalData(E,2)
sage: data.kodaira_symbol()
I2*
sage: data.tamagawa_number()
4
sage: data.tamagawa_exponent()
2

sage: E = EllipticCurve('200c4')
```

```
sage: data = EllipticCurveLocalData(E,5)
sage: data.kodaira_symbol()
I4*
sage: data.tamagawa_number()
4
sage: data.tamagawa_exponent()
2
```

tamagawa_number()

Return the Tamagawa number from this local reduction data.

This is the index $[E(K_v) : E^0(K_v)]$.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import EllipticCurveLocalData
sage: E = EllipticCurve([0,0,0,0,64]); E
Elliptic Curve defined by y^2 = x^3 + 64 over Rational Field
sage: data = EllipticCurveLocalData(E,2)
sage: data.tamagawa_number()
3
```

check_prime(*K*, *P*)

Function to check that P determines a prime of K, and return that ideal.

INPUT:

- K – a number field (including $\mathbf{Q}$).
- P – an element of K or a (fractional) ideal of K.

OUTPUT:

- If K is $\mathbf{Q}$: the prime integer equal to or which generates P.
- If K is not $\mathbf{Q}$‘: the prime ideal equal to or generated by P.

Note: If P is not a prime and does not generate a prime, a TypeError is raised.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import check_prime
sage: check_prime(QQ,3)
3
sage: check_prime(QQ,ZZ.ideal(31))
31
sage: K.<a>=NumberField(x^2-5)
sage: check_prime(K,a)
Fractional ideal (a)
sage: check_prime(K,a+1)
Fractional ideal (a + 1)
sage: [check_prime(K,P) for P in K.primes_above(31)]
[Fractional ideal (-5/2*a - 1/2), Fractional ideal (-5/2*a + 1/2)]
```

38.14 Kodaira symbols.

Kodaira symbols encode the type of reduction of an elliptic curve at a (finite) place.

The standard notation for Kodaira Symbols is as a string which is one of I_m, II, III, IV, I_m^*, II*, III*, IV*, where m denotes a non-negative integer. These have been encoded by single integers by different people. For convenience we give here the conversion table between strings, the eclib coding and the pari encoding.

Kodaira Symbol	Eclib coding	Pari Coding
I_0	0	1
I_0^*	1	-1
I_m $(m > 0)$	$10m$	$m + 4$
I_m^* $(m > 0)$	$10m + 1$	$-(m + 4)$
II	2	2
III	3	3
IV	4	4
II^*	7	-2
III^*	6	-3
IV^*	5	-4

AUTHORS:

- David Roe <roed@math.harvard.edu>
- John Cremona

KodairaSymbol(*symbol*)

Returns the specified Kodaira symbol.

INPUT:

- `symbol` (string or integer) – Either a string of the form "I0", "I1", ..., "In", "II", "III", "IV", "I0*", "I1*", ..., "In*", "II*", "III*", or "IV*", or an integer encoding a Kodaira symbol using Pari's conventions.

OUTPUT:

(KodairaSymbol) The corresponding Kodaira symbol.

EXAMPLES:

```
sage: KS = KodairaSymbol
sage: [KS(n) for n in range(1,10)]
[I0, II, III, IV, I1, I2, I3, I4, I5]
sage: [KS(-n) for n in range(1,10)]
[I0*, II*, III*, IV*, I1*, I2*, I3*, I4*, I5*]
sage: all([KS(str(KS(n)))==KS(n) for n in range(-10,10) if n!=0])
True
```

class **KodairaSymbol_class**(*symbol*)

Class to hold a Kodaira symbol of an elliptic curve over a p-adic local field.

Users should use the `KodairaSymbol()` function to construct Kodaira Symbols rather than use the class constructor directly.

38.15 Isomorphisms between Weierstrass models of elliptic curves

AUTHORS:

- Robert Bradshaw (2007): initial version
- John Cremona (Jan 2008): isomorphisms, automorphisms and twists in all characteristics

class **WeierstrassIsomorphism**(*E=None, urst=None, F=None*)

Class representing a Weierstrass isomorphism between two elliptic curves.

class baseWI (*u=1, r=0, s=0, t=0*)

This class implements the basic arithmetic of isomorphisms between Weierstrass models of elliptic curves. These are specified by lists of the form $[u, r, s, t]$ (with $u \neq 0$) which specifies a transformation $(x, y) \mapsto (x', y')$ where

$$(x, y) = (u^2x' + r, u^3y' + su^2x' + t).$$

INPUT:

- `u, r, s, t` (default (1,0,0,0)) – standard parameters of an isomorphism between Weierstrass models.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import *
sage: baseWI()
(1, 0, 0, 0)
sage: baseWI(2,3,4,5)
(2, 3, 4, 5)
sage: R.<u,r,s,t>=QQ[]; baseWI(u,r,s,t)
(u, r, s, t)
```

is_identity()

Returns True if this is the identity isomorphism.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import *
sage: w=baseWI(); w.is_identity()
True
sage: w=baseWI(2,3,4,5); w.is_identity()
False
```

tuple()

Returns the parameters u, r, s, t as a tuple.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import *
sage: u,r,s,t=baseWI(2,3,4,5).tuple()
sage: w=baseWI(2,3,4,5)
sage: u,r,s,t=w.tuple()
sage: u
2
```

isomorphisms (*E, F, JustOne=False*)

Returns one or all isomorphisms between two elliptic curves.

INPUT:

- `E, F` (EllipticCurve) – Two elliptic curves.
- `JustOne` (bool) If True, returns one isomorphism, or None if the curves are not isomorphic. If False, returns a (possibly empty) list of isomorphisms.

OUTPUT:

Either None, or a 4-tuple (u, r, s, t) representing an isomorphism, or a list of these.

Note: This function is not intended for users, who should use the interface provided by `ell_generic`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import *
sage: isomorphisms(EllipticCurve_from_j(0),EllipticCurve('27a3'))
[(-1, 0, 0, -1), (1, 0, 0, 0)]
sage: isomorphisms(EllipticCurve_from_j(0),EllipticCurve('27a3'),JustOne=True)
(1, 0, 0, 0)
sage: isomorphisms(EllipticCurve_from_j(0),EllipticCurve('27a1'))
[]
sage: isomorphisms(EllipticCurve_from_j(0),EllipticCurve('27a1'),JustOne=True)
```

38.16 Period lattices of elliptic curves and related functions.

Let E be an elliptic curve defined over a number field K (including $\mathbf{Q}$). We attach a period lattice (a discrete rank 2 subgroup of $\mathbf{C}$) to each embedding of K into $\mathbf{C}$.

In the case of real embeddings, the lattice is stable under complex conjugation and is called a real lattice. These have two types: rectangular, (the real curve has two connected components and positive discriminant) or non-rectangular (one connected component, negative discriminant).

The periods are computed to arbitrary precision using the AGM (Gauss's Arithmetic-Geometric Mean).

EXAMPLES:

```
sage: K.<a> = NumberField(x^3-2)
sage: E = EllipticCurve([0,1,0,a,a])
```

First we try a real embedding:

```
sage: emb = K.embeddings(RealField())[0]
sage: L = E.period_lattice(emb); L
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + x^2 + a*x + a over Number Field in
From: Number Field in a with defining polynomial x^3 - 2
To:   Algebraic Real Field
Defn: a |--> 1.259921049894873?
```

The first basis period is real:

```
sage: L.basis()
(3.81452977217855, 1.90726488608927 + 1.34047785962440*I)
sage: L.is_real()
True
```

For a basis ω_1, ω_2 normalised so that ω_1/ω_2 is in the fundamental region of the upper half-plane, use the function `normalised_basis()` instead:

```
sage: L.normalised_basis()
(1.90726488608927 - 1.34047785962440*I, -1.90726488608927 - 1.34047785962440*I)
```

Next a complex embedding:

```
sage: emb = K.embeddings(ComplexField())[0]
sage: L = E.period_lattice(emb); L
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + x^2 + a*x + a over Number Field in
From: Number Field in a with defining polynomial x^3 - 2
```

```
To:   Algebraic Field
Defn: a |--> -0.6299605249474365? - 1.091123635971722?*I
```

In this case, the basis ω_1, ω_2 is always normalised so that $\tau = \omega_1/\omega_2$ is in the fundamental region in the upper half plane:

```
sage: w1,w2 = L.basis(); w1,w2
(-1.37588604166076 - 2.58560946624443*I, -2.10339907847356 + 0.428378776460622*I)
sage: L.is_real()
False
sage: tau = w1/w2; tau
0.387694505032876 + 1.30821088214407*I
sage: L.normalised_basis()
(-1.37588604166076 - 2.58560946624443*I, -2.10339907847356 + 0.428378776460622*I)
```

AUTHORS:

- ?: initial version.
- John Cremona:
  - Adapted to handle real embeddings of number fields, September 2008.
  - Added basis_matrix function, November 2008
  - Added support for complex embeddings, May 2009.

class **PeriodLattice**(*base_ring, rank, degree, sparse=False*)

The class for the period lattice of an algebraic variety.

class **PeriodLattice_ell**(*E, embedding=None*)

The class for the period lattice of an elliptic curve.

Currently supported are elliptic curves defined over $\mathbf{Q}$, and elliptic curves defined over a number field with a real or complex embedding, where the lattice constructed depends on that embedding.

basis(*prec=None, algorithm='sage'*)

Return a basis for this period lattice as a 2-tuple.

INPUT:

- `prec` (default: `None`) – precision in bits (default precision if `None`).
- `algorithm` (string, default 'sage') – choice of implementation (for real ambeddings only) between 'sage' (native Sage implementation) or 'pari' (use the pari library: only available for real embeddings).

OUTPUT:

(tuple of Complex) (ω_1, ω_2) where the lattice is $\mathbf{Z}\omega_1 + \mathbf{Z}\omega_2$. If the lattice is real then ω_1 is real and positive, $\Im(\omega_2) > 0$ and $\Re(\omega_1/\omega_2)$ is either 0 (for rectangular lattices) or $\frac{1}{2}$ (for non-rectangular lattices). Otherwise, ω_1/ω_2 is in the fundamental region of the upper half-plane. If the latter normalisation is required for real lattices, use the function `normalised_basis()` instead.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().basis()
(2.99345864623196, 2.45138938198679*I)
```

This shows that the issue reported at trac #3954 is fixed:

```
sage: E = EllipticCurve('37a')
sage: b1 = E.period_lattice().basis(prec=30)
sage: b2 = E.period_lattice().basis(prec=30)
```

```
sage: b1 == b2
True
```

This shows that the issue reported at trac #4064 is fixed:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().basis(prec=30)[0].parent()
Real Field with 30 bits of precision
sage: E.period_lattice().basis(prec=100)[0].parent()
Real Field with 100 bits of precision

sage: K.<a> = NumberField(x^3-2)
sage: emb = K.embeddings(RealField())[0]
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(emb)
sage: L.basis(64)
(3.81452977217854509, 1.90726488608927254 + 1.34047785962440202*I)

sage: emb = K.embeddings(ComplexField())[0]
sage: L = E.period_lattice(emb)
sage: w1,w2 = L.basis(); w1,w2
(-1.37588604166076 - 2.58560946624443*I, -2.10339907847356 + 0.428378776460622*I)
sage: L.is_real()
False
sage: tau = w1/w2; tau
0.387694505032876 + 1.30821088214407*I
```

basis_matrix(*prec=None, normalised=False*)

Return the basis matrix of this period lattice.

INPUT:

- prec (int or None''(default)) - real precision in bits (default real precision if ''None).
- normalised (bool, default None) – if True and the embedding is real, use the normalised basis (see normalised_basis()) instead of the default.

OUTPUT:

A 2x2 real matrix whose rows are the lattice basis vectors, after identifying $\mathbf{C}$ with $\mathbf{R}^2$.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().basis_matrix()
[ 2.99345864623196 0.000000000000000]
[0.000000000000000  2.45138938198679]

sage: K.<a> = NumberField(x^3-2)
sage: emb = K.embeddings(RealField())[0]
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(emb)
sage: L.basis_matrix(64)
[ 3.81452977217854509 0.000000000000000000]
[ 1.90726488608927254  1.34047785962440202]
```

See #4388:

```
sage: L = EllipticCurve('11a1').period_lattice()
sage: L.basis_matrix()
[ 1.26920930427955 0.000000000000000]
[0.634604652139775  1.45881661693850]
sage: L.basis_matrix(normalised=True)
```

```
[0.634604652139775 -1.45881661693850]
[-1.26920930427955 0.000000000000000]

sage: L = EllipticCurve('389a1').period_lattice()
sage: L.basis_matrix()
[ 2.49021256085505 0.000000000000000]
[0.000000000000000  1.97173770155165]
sage: L.basis_matrix(normalised=True)
[ 2.49021256085505 0.000000000000000]
[0.000000000000000 -1.97173770155165]
```

complex_area(*prec=None*)

Return the area of a fundamental domain for the period lattice of the elliptic curve.

INPUT:

- prec (int or None''(default)) - real precision in bits (default real precision if ''None).

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().complex_area()
7.33813274078958

sage: K.<a> = NumberField(x^3-2)
sage: embs = K.embeddings(ComplexField())
sage: E = EllipticCurve([0,1,0,a,a])
sage: [E.period_lattice(emb).is_real() for emb in K.embeddings(CC)]
[False, False, True]
sage: [E.period_lattice(emb).complex_area() for emb in embs]
[6.02796894766694, 6.02796894766694, 5.11329270448346]
```

is_real()

Return True if this period lattice is real.

EXAMPLES:

```
sage: f = EllipticCurve('11a')
sage: f.period_lattice().is_real()
True

sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve(K,[0,0,0,i,2*i])
sage: emb = K.embeddings(ComplexField())[0]
sage: L = E.period_lattice(emb)
sage: L.is_real()
False

sage: K.<a> = NumberField(x^3-2)
sage: E = EllipticCurve([0,1,0,a,a])
sage: [E.period_lattice(emb).is_real() for emb in K.embeddings(CC)]
[False, False, True]
```

ALGORITHM:

The lattice is real if it is associated to a real embedding; such lattices are stable under conjugation.

is_rectangular()

Return True if this period lattice is rectangular.

Note: Only defined for real lattices; a RuntimeError is raised for non-real lattices.

EXAMPLES:

```
sage: f = EllipticCurve('11a')
sage: f.period_lattice().basis()
(1.26920930427955, 0.634604652139775 + 1.45881661693850*I)
sage: f.period_lattice().is_rectangular()
False

sage: f = EllipticCurve('37b')
sage: f.period_lattice().basis()
(1.08852159290423, 1.76761067023379*I)
sage: f.period_lattice().is_rectangular()
True
```

ALGORITHM:

The period lattice is rectangular precisely if the discriminant of the Weierstrass equation is positive, or equivalently if the number of real components is 2.

normalised_basis (*prec=None, algorithm='sage'*)

Return a normalised basis for this period lattice as a 2-tuple.

INPUT:

- `prec` (default: `None`) – precision in bits (default precision if `None`).
- `algorithm` (string, default 'sage') – choice of implementation (for real ambeddings only) between 'sage' (native Sage implementation) or 'pari' (use the pari library: only available for real embeddings).

OUTPUT:

(tuple of Complex) (ω_1, ω_2) where the lattice has the form $\mathbf{Z}\omega_1 + \mathbf{Z}\omega_2$. The basis is normalised so that ω_1/ω_2 is in the fundamental region of the upper half-plane. For an alternative normalisation for real lattices (with the first period real), use the function basis() instead.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().normalised_basis()
(2.99345864623196, -2.45138938198679*I)

sage: K.<a> = NumberField(x^3-2)
sage: emb = K.embeddings(RealField())[0]
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(emb)
sage: L.normalised_basis(64)
(1.90726488608927254 - 1.34047785962440202*I, -1.90726488608927254 - 1.34047785962440202*I)

sage: emb = K.embeddings(ComplexField())[0]
sage: L = E.period_lattice(emb)
sage: w1,w2 = L.normalised_basis(); w1,w2
(-1.37588604166076 - 2.58560946624443*I, -2.10339907847356 + 0.428378776460622*I)
sage: L.is_real()
False
sage: tau = w1/w2; tau
0.387694505032876 + 1.30821088214407*I
```

omega (*prec=None*)

Returns the real or complex volume of this period lattice.

INPUT:

- `prec` (int or `None``(default)) - real precision in bits (default real precision if ``None`)

OUTPUT:

(real) For real lattices, this is the real period times the number of connected components. For non-real lattices it is the complex area.

Note: If the curve is defined over **Q** and is given by a *minimal* Weierstrass equation, then this is the correct period in the BSD conjecture, i.e., it is the least real period * 2 when the period lattice is rectangular. More generally the product of this quantity over all embeddings appears in the generalised BSD formula.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().omega()
5.98691729246392
```

This is not a minimal model:

```
sage: E = EllipticCurve([0,-432*6^2])
sage: E.period_lattice().omega()
0.486109385710056
```

If you were to plug the above omega into the BSD conjecture, you would get nonsense. The following works though:

```
sage: F = E.minimal_model()
sage: F.period_lattice().omega()
0.972218771420113

sage: K.<a> = NumberField(x^3-2)
sage: emb = K.embeddings(RealField())[0]
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(emb)
sage: L.omega(64)
3.81452977217854509
```

A complex example (taken from J.E.Cremona and E.Whitley, *Periods of cusp forms and elliptic curves over imaginary quadratic fields*, Mathematics of Computation 62 No. 205 (1994), 407-429):

```
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0,1-i,i,-i,0])
sage: L = E.period_lattice(K.embeddings(CC)[0])
sage: L.omega()
8.80694160502647
```

real_period(*prec=None*)

Returns the real period of this period lattice.

INPUT:

- `prec` (int or `None` (default)) – real precision in bits (default real precision if `None`)

Note: Only defined for real lattices; a RuntimeError is raised for non-real lattices.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().real_period()
2.99345864623196

sage: K.<a> = NumberField(x^3-2)
sage: emb = K.embeddings(RealField())[0]
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(emb)
sage: L.real_period(64)
3.81452977217854509
```

sigma(*z, prec=None, flag=0*)

Returns the value of the Weierstrass sigma function for this elliptic curve period lattice.

INPUT:

- `z` – a complex number

- **prec (default: None) – real precision in bits** (default real precision if None).
- flag –
  0: (default) ???;
  1: computes an arbitrary determination of log(sigma(z))
  2, 3: same using the product expansion instead of theta series. ???

Note: The reason for the ???'s above, is that the PARI documentation for ellsigma is very vague. Also this is only implemented for curves defined over $\mathbf{Q}$.

TODO:

This function does not use any of the PeriodLattice functions and so should be moved to ell_rational_field.

EXAMPLES:

```
sage: EllipticCurve('389a1').period_lattice().sigma(CC(2,1))
2.60912163570108 - 0.200865080824587*I
```

normalise_periods(*w1, w2*)

Normalise the period basis (w_1, w_2) so that w_1/w_2 is in the fundamental region.

INPUT:

- w1, w2 (complex) – two complex numbers with non-real ratio

OUTPUT:

(tuple) $((\omega_1', \omega_2'), [a, b, c, d])$ where a, b, c, d are integers such that

- $ad - bc = \pm 1$;
- $(\omega_1', \omega_2') = (a\omega_1 + b\omega_2, c\omega_1 + d\omega_2)$;
- $\tau = \omega_1'/\omega_2'$ is in the upper half plane;
- $|\tau| \geq 1$ and $|\Re(\tau)| \leq \frac{1}{2}$.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.period_lattice import reduce_tau, normalise_periods
sage: w1 = CC(1.234, 3.456)
sage: w2 = CC(1.234, 3.456000001)
sage: w1/w2    # in lower half plane!
0.999999999743367 - 9.16334785827644e-11*I
sage: w1w2, abcd = normalise_periods(w1,w2)
sage: a,b,c,d = abcd
sage: w1w2 == (a*w1+b*w2, c*w1+d*w2)
True
sage: w1w2[0]/w1w2[1]
1.23400010389203e9*I
sage: a*d-b*c # note change of orientation
-1
```

reduce_tau(*tau*)

Transform a point in the upper half plane to the fundamental region.

INPUT:

- tau (complex) – a complex number with positive imaginary part

OUTPUT:

(tuple) $(\tau', [a, b, c, d])$ where a, b, c, d are integers such that

- $ad - bc = 1$;
- $\tau' = (a\tau + b)/(c\tau + d)$;

- $|\tau'| \geq 1$;
- $|\Re(\tau')| \leq \frac{1}{2}$.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.period_lattice import reduce_tau
sage: reduce_tau(CC(1.23,3.45))
(0.230000000000000 + 3.45000000000000*I, [1, -1, 0, 1])
sage: reduce_tau(CC(1.23,0.0345))
(-0.463960069171512 + 1.35591888067914*I, [-5, 6, 4, -5])
sage: reduce_tau(CC(1.23,0.0000345))
(0.130000000001761 + 2.89855072463768*I, [13, -16, 100, -123])
```

38.17 Formal groups of elliptic curves.

AUTHORS:

- William Stein: original implementations
- David Harvey: improved asymptotics of some methods
- Nick Alexander: separation from ell_generic.py, bugfixes and docstrings

class **EllipticCurveFormalGroup**(*E*)

The formal group associated to an elliptic curve.

curve()

The elliptic curve this formal group is associated to.

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: F = E.formal_group()
sage: F.curve()
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

differential(*prec=20*)

Returns the power series $f(t) = 1 + \cdots$ such that $f(t)dt$ is the usual invariant differential $dx/(2y + a_1x + a_3)$.

INPUT:

- `prec` - nonnegative integer (default 20), answer will be returned $O(t^{\text{prec}})$

OUTPUT: a power series with given precision

DETAILS: Return the formal series

$$f(t) = 1 + a_1 t + ({a_1}^2 + a_2)t^2 + \cdots$$

to precision $O(t^{prec})$ of page 113 of [Silverman AEC1].

The result is cached, and a cached version is returned if possible.

Warning: The resulting series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

EXAMPLES:

```
sage: EllipticCurve([-1, 1/4]).formal_group().differential(15)
 1 - 2*t^4 + 3/4*t^6 + 6*t^8 - 5*t^10 - 305/16*t^12 + 105/4*t^14 + O(t^15)
sage: EllipticCurve(Integers(53), [-1, 1/4]).formal_group().differential(15)
 1 + 51*t^4 + 14*t^6 + 6*t^8 + 48*t^10 + 24*t^12 + 13*t^14 + O(t^15)
```

AUTHOR:

•David Harvey (2006-09-10): factored out of log

group_law (*prec=10*)

The formal group law.

INPUT:

•`prec` - integer (default 10)

OUTPUT: a power series with given precision in ZZ[[ZZ[['t1']],'t2']]

DETAILS: Return the formal power series

$$F(t_1, t_2) = t_1 + t_2 - a_1 t_1 t_2 - \cdots$$

to precision $O(t^{prec})$ of page 115 of [Silverman AEC1].

The result is cached, and a cached version is returned if possible.

Warning: The resulting power series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

AUTHORS:

•Nick Alexander: minor fixes, docstring

EXAMPLES:

```
sage: e = EllipticCurve([1, 2])
sage: F = e.formal_group().group_law(5); F
 t1 + O(t1^5) + (1 - 2*t1^4 + O(t1^5))*t2 + (-4*t1^3 + O(t1^5))*t2^2 + (-4*t1^2 - 30*t1^4 +
sage: i = e.formal_group().inverse(5)
sage: Fx = F.base_extend(F.base_ring().base_extend(i.parent()))
sage: Fx (i.parent().gen()) (i)
 O(t^5)
```

Let's ensure caching with changed precision is working:

```
sage: e.formal_group().group_law(4)
 t1 + O(t1^4) + (1 + O(t1^4))*t2 + (-4*t1^3 + O(t1^4))*t2^2 + (-4*t1^2 + O(t1^4))*t2^3 + O(t
```

inverse (*prec=20*)

The formal group inverse law i(t), which satisfies F(t, i(t)) = 0.

INPUT:

•`prec` - integer (default 20)

OUTPUT: a power series with given precision

DETAILS: Return the formal power series

$$i(t) = -t + a_1 t^2 + \cdots$$

to precision $O(t^{prec})$ of page 114 of [Silverman AEC1].

The result is cached, and a cached version is returned if possible.

Warning: The resulting power series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

EXAMPLES:

```
sage: e = EllipticCurve([1, 2])
sage: F = e.formal_group().group_law(5)
sage: i = e.formal_group().inverse(5)
sage: Fx = F.base_extend(F.base_ring().base_extend(i.parent()))
sage: Fx (i) (i.parent().gen())
O(t^5)
```

log (*prec=20*)

Returns the power series $f(t) = t + \cdots$ which is an isomorphism to the additive formal group.

Generally this only makes sense in characteristic zero, although the terms before t^p may work in characteristic p.

INPUT:

- `prec` - nonnegative integer (default 20)

OUTPUT: a power series with given precision

EXAMPLES:

```
sage: EllipticCurve([-1, 1/4]).formal_group().log(15)
 t - 2/5*t^5 + 3/28*t^7 + 2/3*t^9 - 5/11*t^11 - 305/208*t^13 + O(t^15)
```

AUTHORS:

- David Harvey (2006-09-10): rewrote to use differential

mult_by_n (*n, prec=10*)

The formal 'multiplication by n' endomorphism $[n]$.

INPUT:

- `prec` - integer (default 10)

OUTPUT: a power series with given precision

DETAILS: Return the formal power series

$$[n](t) = nt + \cdots$$

to precision $O(t^{prec})$ of Proposition 2.3 of [Silverman AEC1].

Warning: The resulting power series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

AUTHORS:

- Nick Alexander: minor fixes, docstring
- David Harvey (2007-03): faster algorithm for char 0 field case

EXAMPLES:

```
sage: e = EllipticCurve([1, 2, 3, 4, 6])
sage: e.formal_group().mult_by_n(0, 5)
 O(t^5)
sage: e.formal_group().mult_by_n(1, 5)
 t + O(t^5)
```

We verify an identity of low degree:

```
sage: none = e.formal_group().mult_by_n(-1, 5)
sage: two = e.formal_group().mult_by_n(2, 5)
sage: ntwo = e.formal_group().mult_by_n(-2, 5)
sage: ntwo - none(two)
 O(t^5)
sage: ntwo - two(none)
 O(t^5)
```

It's quite fast:

```
sage: E = EllipticCurve("37a"); F = E.formal_group()
sage: F.mult_by_n(100, 20)
100*t - 49999950*t^4 + 3999999960*t^5 + 14285614285800*t^7 - 2999989920000150*t^8 + 13333332
```

sigma (*prec=10*)

EXAMPLE:

```
sage: E = EllipticCurve('14a')
sage: F = E.formal_group()
sage: F.sigma(5)
t + 1/2*t^2 + (1/2*c + 1/3)*t^3 + (3/4*c + 3/4)*t^4 + O(t^5)
```

w (*prec=20*)

The formal group power series w.

INPUT:

•`prec` - integer (default 20)

OUTPUT: a power series with given precision

DETAILS: Return the formal power series

$$w(t) = t^3 + a_1 t^4 + (a_2 + a_1^2)t^5 + \cdots$$

to precision $O(t^{prec})$ of Proposition IV.1.1 of [Silverman AEC1]. This is the formal expansion of $w = -1/y$ about the formal parameter $t = -x/y$ at
$infty$.

The result is cached, and a cached version is returned if possible.

Warning: The resulting power series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

ALGORITHM: Uses Newton's method to solve the elliptic curve equation at the origin. Complexity is roughly $O(M(n))$ where n is the precision and $M(n)$ is the time required to multiply polynomials of length n over the coefficient ring of E.

AUTHOR:

•David Harvey (2006-09-09): modified to use Newton's method instead of a recurrence formula.

EXAMPLES:

```
sage: e = EllipticCurve([0, 0, 1, -1, 0])
sage: e.formal_group().w(10)
 t^3 + t^6 - t^7 + 2*t^9 + O(t^10)
```

Check that caching works:

```
sage: e = EllipticCurve([3, 2, -4, -2, 5])
sage: e.formal_group().w(20)
 t^3 + 3*t^4 + 11*t^5 + 35*t^6 + 101*t^7 + 237*t^8 + 312*t^9 - 949*t^10 - 10389*t^11 - 57087
sage: e.formal_group().w(7)
 t^3 + 3*t^4 + 11*t^5 + 35*t^6 + O(t^7)
sage: e.formal_group().w(35)
 t^3 + 3*t^4 + 11*t^5 + 35*t^6 + 101*t^7 + 237*t^8 + 312*t^9 - 949*t^10 - 10389*t^11 - 57087
```

x (*prec=20*)

Return the formal series $x(t) = t/w(t)$ in terms of the local parameter $t = -x/y$ at infinity.

INPUT:

•`prec` - integer (default 20)

OUTPUT: a Laurent series with given precision
DETAILS: Return the formal series

$$x(t) = t^{-2} - a_1 t^{-1} - a_2 - a_3 t - \cdots$$

to precision $O(t^{prec})$ of page 113 of [Silverman AEC1].

Warning: The resulting series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

EXAMPLES:

```
sage: EllipticCurve([0, 0, 1, -1, 0]).formal_group().x(10)
 t^-2 - t + t^2 - t^4 + 2*t^5 - t^6 - 2*t^7 + 6*t^8 - 6*t^9 + O(t^10)
```

y (*prec=20*)
Return the formal series $y(t) = -1/w(t)$ in terms of the local parameter $t = -x/y$ at infinity.
INPUT:

•`prec` - integer (default 20)

OUTPUT: a Laurent series with given precision
DETAILS: Return the formal series

$$y(t) = -t^{-3} + a_1 t^{-2} + a_2 t + a_3 + \cdots$$

to precision $O(t^{prec})$ of page 113 of [Silverman AEC1].
The result is cached, and a cached version is returned if possible.

Warning: The resulting series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

EXAMPLES:

```
sage: EllipticCurve([0, 0, 1, -1, 0]).formal_group().y(10)
 -t^-3 + 1 - t + t^3 - 2*t^4 + t^5 + 2*t^6 - 6*t^7 + 6*t^8 + 3*t^9 + O(t^10)
```

38.18 Tate's parametrisation of p-adic curves with multiplicative reduction

Let E be an elliptic curve defined over the p-adic numbers $\mathbf{Q}_p$. Suppose that E has multiplicative reduction, i.e. that the j-invariant of E has negative valuation, say n. Then there exists a parameter q in $\mathbf{Z}_p$ of valuation n such that the points of E defined over the algebraic closure $\bar{\mathbf{Q}}_p$ are in bijection with $\bar{\mathbf{Q}}_p^\times / q^{\mathbf{Z}}$. More precisely there exists the series $s_4(q)$ and $s_6(q)$ such that the $y^2 + xy = x^3 + s_4(q)x + s_6(q)$ curve is isomorphic to E over $\bar{\mathbf{Q}}_p$ (or over $\mathbf{Q}_p$ if the reduction is *split* multiplicative). There is p-adic analytic map from $\bar{\mathbf{Q}}_p^\times$ to this curve with kernel $q^{\mathbf{Z}}$. Points of good reduction correspond to points of valuation 0 in $\bar{\mathbf{Q}}_p^\times$. See chapter V of [Sil2] for more details.

REFERENCES :

- **[Sil2] Silverman Joseph, Advanced Topics in the Arithmetic of Elliptic Curves,** GTM 151, Springer 1994.

AUTHORS:

- chris wuthrich (23/05/2007): first version
- William Stein (2007-05-29): added some examples; editing.

- chris wuthrich (04/09): reformated docstrings.

class TateCurve (*E, p*)

Tate's p-adic uniformisation of an elliptic curve with multiplicative reduction.

Note: Some of the methods of this Tate curve only work when the reduction is split multiplicative over $\mathbf{Q}_p$.

EXAMPLES:

```
sage: e = EllipticCurve('130a1')
sage: eq = e.tate_curve(5); eq
5-adic Tate curve associated to the Elliptic Curve defined by y^2 + x*y + y = x^3 - 33*x + 68 ov
sage: eq == loads(dumps(eq))
True
```

REFERENCES :

•[Sil2] Silverman Joseph, Advanced Topics in the Arithmetic of Elliptic Curves, GTM 151, Springer 1994.

E2 (*prec=20*)

Returns the value of the p-adic Eisenstein series of weight 2 evaluated on the elliptic curve having split multiplicative reduction.

INPUT:

•`prec` - the p-adic precision, default is 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.E2(prec=10)
4 + 2*5^2 + 2*5^3 + 5^4 + 2*5^5 + 5^7 + 5^8 + 2*5^9 + O(5^10)
```

L_invariant (*prec=20*)

Returns the *mysterious* $\mathcal{L}$-invariant associated to an elliptic curve with split multiplicative reduction. One instance where this constant appears is in the exceptional case of the p-adic Birch and Swinnerton-Dyer conjecture as formulated in [MTT]. See [Col] for a detailed discussion.

INPUT:

•`prec` - the p-adic precision, default is 20.

REFERENCES:

•[MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.

•[Col] Pierre Colmez, Invariant $\mathcal{L}$ et derivees de valeurs propores de Frobenius, preprint, 2004.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.L_invariant(prec=10)
5^3 + 4*5^4 + 2*5^5 + 2*5^6 + 2*5^7 + 3*5^8 + 5^9 + O(5^10)
```

curve (*prec=20*)

Returns the p-adic elliptic curve of the form $y^2 + xy = x^3 + s_4x + s_6$. This curve with split multiplicative reduction is isomorphic to the given curve over the algebraic closure of $\mathbf{Q}_p$.

INPUT:

•`prec` - the p-adic precision, default is 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.curve(prec=5)
Elliptic Curve defined by y^2 + (1+O(5^5))*x*y  = x^3 +
(2*5^4+5^5+2*5^6+5^7+3*5^8+O(5^9))*x + (2*5^3+5^4+2*5^5+5^7+O(5^8)) over 5-adic
Field with capped relative precision 5
```

is_split()

Returns True if the given elliptic curve has split multiplicative reduction.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.is_split()
True

sage: eq = EllipticCurve('37a1').tate_curve(37)
sage: eq.is_split()
False
```

lift(*P, prec=20*)

Given a point P in the formal group of the elliptic curve E with split multiplicative reduction, this produces an element u in $\mathbf{Q}_p^\times$ mapped to the point P by the Tate parametrisation. The algorithm return the unique such element in $1 + p\mathbf{Z}_p$.

INPUT:

- `P` - a point on the elliptic curve.
- `prec` - the p-adic precision, default is 20.

EXAMPLES:

```
sage: e = EllipticCurve('130a1')
sage: eq = e.tate_curve(5)
sage: P = e([-6,10])
sage: l = eq.lift(12*P, prec=10); l
1 + 4*5 + 5^3 + 5^4 + 4*5^5 + 5^6 + 5^7 + 4*5^8 + 5^9 + O(5^10)
```

Now we map the lift l back and check that it is indeed right.:

```
sage: eq.parametrisation_onto_original_curve(l)
(4*5^-2 + 2*5^-1 + 4*5 + 3*5^3 + 5^4 + 2*5^5 + 4*5^6 + O(5^7) : 2*5^-3 + 5^-1 + 4 + 4*5 + 5^
sage: e5 = e.change_ring(Qp(5,9))
sage: e5(12*P)
(4*5^-2 + 2*5^-1 + 4*5 + 3*5^3 + 5^4 + 2*5^5 + 4*5^6 + O(5^7) : 2*5^-3 + 5^-1 + 4 + 4*5 + 5^
```

original_curve()

Returns the elliptic curve the Tate curve was constructed from.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.original_curve()
Elliptic Curve defined by y^2 + x*y + y = x^3 - 33*x + 68 over Rational Field
```

padic_height(*prec=20*)

Returns the canonical p-adic height function on the original curve.

INPUT:

- `prec` - the p-adic precision, default is 20.

OUTPUT:

- A function that can be evaluated on rational points of E.

EXAMPLES:

```
sage: e = EllipticCurve('130a1')
sage: eq = e.tate_curve(5)
sage: h = eq.padic_height(prec=10)
sage: P=e.gens()[0]
sage: h(P)
2*5^-1 + 1 + 2*5 + 2*5^2 + 3*5^3 + 3*5^6 + 5^7 + O(5^8)
```

Check that it is a quadratic function:

```
sage: h(3*P)-3^2*h(P)
O(5^8)
```

padic_regulator (*prec=20*)

Computes the canonical p-adic regulator on the extended Mordell-Weil group as in [MTT] (with the correction of [Wer] and sign convention in [SW].) The p-adic Birch and Swinnerton-Dyer conjecture predicts that this value appears in the formula for the leading term of the p-adic L-function.

INPUT:

•`prec` - the p-adic precision, default is 20.

REFERENCES:

•[MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.

•[Wer] Annette Werner, Local heights on abelian varieties and rigid analytic unifomization, Doc. Math. 3 (1998), 301-319.

•[SW] William Stein and Christian Wuthrich, Computations About Tate-Shafarevich Groups using Iwasawa theory, preprint 2009.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.padic_regulator()
2*5^-1 + 1 + 2*5 + 2*5^2 + 3*5^3 + 3*5^6 + 5^7 + 3*5^9 + 3*5^10 + 3*5^12 + 4*5^13 + 3*5^15 +
```

parameter (*prec=20*)

Returns the Tate parameter q such that the curve is isomorphic over the algebraic closure of $\mathbf{Q}_p$ to the curve $\mathbf{Q}_p^\times/q^{\mathbf{Z}}$.

INPUT:

•`prec` - the p-adic precision, default is 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.parameter(prec=5)
3*5^3 + 3*5^4 + 2*5^5 + 2*5^6 + 3*5^7 + O(5^8)
```

parametrisation_onto_original_curve (*u, prec=20*)

Given an element u in $\mathbf{Q}_p^\times$, this computes its image on the original curve under the p-adic uniformisation of E.

INPUT:

•`u` - a non-zero p-adic number.

•`prec` - the p-adic precision, default is 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.parametrisation_onto_original_curve(1+5+5^2+O(5^10))
(4*5^-2 + 4*5^-1 + 4 + 2*5^3 + 3*5^4 + 2*5^6 + O(5^7) :
3*5^-3 + 5^-2 + 4*5^-1 + 1 + 4*5 + 5^2 + 3*5^5 + O(5^6) : 1 + O(5^20))
```

Here is how one gets a 4-torsion point on E over $\mathbf{Q}_5$:

```
sage: R = Qp(5,10)
sage: i = R(-1).sqrt()
sage: T = eq.parametrisation_onto_original_curve(i); T
(2 + 3*5 + 4*5^2 + 2*5^3 + 5^4 + 4*5^5 + 2*5^7 + 5^8 + 5^9 + O(5^10) :
3*5 + 5^2 + 5^4 + 3*5^5 + 3*5^7 + 2*5^8 + 4*5^9 + O(5^10) : 1 + O(5^20))
sage: 4*T
(0 : 1 + O(5^20) : 0)
```

parametrisation_onto_tate_curve(*u, prec=20*)

Given an element u in $\mathbf{Q}_p^\times$, this computes its image on the Tate curve under the p-adic uniformisation of E.

INPUT:

- `u` - a non-zero p-adic number.
- `prec` - the p-adic precision, default is 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.parametrisation_onto_tate_curve(1+5+5^2+O(5^10))
(5^-2 + 4*5^-1 + 1 + 2*5 + 3*5^2 + 2*5^5 + 3*5^6 + O(5^7) :
4*5^-3 + 2*5^-1 + 4 + 2*5 + 3*5^4 + 2*5^5 + O(5^6) : 1 + O(5^20))
```

prime()

Returns the residual characteristic p.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.original_curve()
Elliptic Curve defined by y^2 + x*y + y = x^3 - 33*x + 68 over Rational Field
sage: eq.prime()
5
```

38.19 Computation of Frobenius matrix on Monsky-Washnitzer cohomology.

The most interesting functions to be exported here are matrix_of_frobenius() and adjusted_prec().

Currently this code is limited to the case $p \geq 5$ (no $GF(p^n)$ for $n > 1$), and only handles the elliptic curve case (not more general hyperelliptic curves).

REFERENCES:

- Kedlaya, K., "Counting points on hyperelliptic curves using Monsky-Washnitzer cohomology", J. Ramanujan Math. Soc. 16 (2001) no 4, 323-338
- Edixhoven, B., "Point counting after Kedlaya", EIDMA-Stieltjes graduate course, Lieden (lecture notes?).

AUTHORS:

- David Harvey and Robert Bradshaw: initial code developed at the 2006 MSRI graduate workshop, working with Jennifer Balakrishnan and Liang Xiao
- David Harvey (2006-08): cleaned up, rewrote some chunks, lots more documentation, added Newton iteration method, added more complete 'trace trick', integrated better into Sage.
- David Harvey (2007-02): added algorithm with sqrt(p) complexity (removed in May 2007 due to better C++ implementation)
- Robert Bradshaw (2007-03): keep track of exact form in reduction algorithms
- Robert Bradshaw (2007-04): generalization to hyperelliptic curves

class MonskyWashnitzerDifferential(*parent, val=0, offset=0*)

Represents an element of the form F dx/2y

coeff()
This is a one-dimensional module over the base ring, generated by dx/2y. Return A where $Adx/2y = self$.

coeffs(*R=None*)

coleman_integral(*P, Q*)

extract_pow_y(*k*)
Really the power of y in A where self = A dx/2y.

integrate(*P, Q*)

max_pow_y()
Really the maximum power of y in A where self = A dx/2y.

min_pow_y()
Really the minimum power of y in A where self = A dx/2y.

reduce()
Use homology relations to find a and f such that $self = a + df$ where a is given in terms of the $x^i dx/2y$.

reduce_fast(*even_degree_only=False*)
Use homology relations to find a and f such that $self = a + df$ where a is given in terms of the $x^i dx/2y$.
EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^3-4*x+4)
sage: x, y = E.monsky_washnitzer_gens()
sage: x.diff().reduce_fast()
(x, (0, 0))
sage: y.diff().reduce_fast()
(y*1, (0, 0))
sage: (y^-1).diff().reduce_fast()
((y^-1)*1, (0, 0))
sage: (y^-11).diff().reduce_fast()
((y^-11)*1, (0, 0))
sage: (x*y^2).diff().reduce_fast()
(y^2*x, (0, 0))
```

reduce_neg_y()
Use homology relations to eliminate negative powers of y.

reduce_neg_y_fast(*even_degree_only=False*)
Use homology relations to eliminate negative powers of y.
EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5-3*x+1)
sage: x, y = E.monsky_washnitzer_gens()
sage: (y^-1).diff().reduce_neg_y_fast()
((y^-1)*1, 0 dx/2y)
sage: (y^-5*x^2+y^-1*x).diff().reduce_neg_y_fast()
((y^-1)*x + (y^-5)*x^2, 0 dx/2y)
```

It leaves non-negative powers of y alone:

```
sage: y.diff()
((-3)*1 + 5*x^4) dx/2y
sage: y.diff().reduce_neg_y_fast()
(0, ((-3)*1 + 5*x^4) dx/2y)
```

reduce_neg_y_faster(*even_degree_only=False*)
Use homology relations to eliminate negative powers of y.

reduce_pos_y()
Use homology relations to eliminate positive powers of y.

reduce_pos_y_fast(*even_degree_only=False*)
Use homology relations to eliminate positive powers of y.
EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^3-4*x+4)
sage: x, y = E.monsky_washnitzer_gens()
sage: y.diff().reduce_pos_y_fast()
(y*1, 0 dx/2y)
sage: (y^2).diff().reduce_pos_y_fast()
(y^2*1, 0 dx/2y)
sage: (y^2*x).diff().reduce_pos_y_fast()
(y^2*x, 0 dx/2y)
sage: (y^92*x).diff().reduce_pos_y_fast()
(y^92*x, 0 dx/2y)
sage: w = (y^3 + x).diff()
sage: w += w.parent()(x)
sage: w.reduce_pos_y_fast()
(y^3*1 + x, x dx/2y)
```

MonskyWashnitzerDifferentialRing(*base_ring*)

class **MonskyWashnitzerDifferentialRing_class**(*base_ring*)

Q()

base_extend(*R*)

change_ring(*R*)

degree()

frob_Q(*p*)

frob_basis_elements(*prec, p*)

frob_invariant_differential(*prec, p*)

$$F_p(dx/y) = px^{p-1}y(F_py)^{-1}dx/y = px^{p-1}y^{1-p}(1+pEy^{-2p})^{-1/2}dx/y = px^{p-1}y^{1-p}(F_pQy^{-p})^{-1/2}dx/y$$

Use Newton's method to calculate the square root.

helper_matrix()
We use this to solve for the linear combination of x^iy^j needed to clear all terms with y^{j-1}.

invariant_differential()

x_to_p(*p*)

class **SpecialCubicQuotientRing**(*Q, laurent_series=False*)
Specialised class for representing the quotient ring $R[x,T]/(T-x^3-ax-b)$, where R is an arbitrary commutative base ring (in which 2 and 3 are invertible), a and b are elements of that ring.

Polynomials are represented internally in the form $p_0+p_1x+p_2x^2$ where the p_i are polynomials in T. Multiplication of polynomials always reduces high powers of x (i.e. beyond x^2) to powers of T.

Hopefully this ring is faster than a general quotient ring because it uses the special structure of this ring to speed multiplication (which is the dominant operation in the frobenius matrix calculation). I haven't actually tested this theory though...

TODO: - Eventually we will want to run this in characteristic 3, so we need to: (a) Allow Q(x) to contain an x^2 term, and (b) Remove the requirement that 3 be invertible. Currently this is used in the Toom-Cook algorithm to speed multiplication.

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: R
SpecialCubicQuotientRing over Ring of integers modulo 125 with polynomial T = x^3 + 124*x + 94
```

Get generators:

```
sage: x, T = R.gens()
sage: x
(0) + (1)*x + (0)*x^2
sage: T
(T) + (0)*x + (0)*x^2
```

Coercions:

```
sage: R(7)
(7) + (0)*x + (0)*x^2
```

Create elements directly from polynomials:

```
sage: A, z = R.poly_ring().objgen()
sage: A
Univariate Polynomial Ring in T over Ring of integers modulo 125
sage: R.create_element(z^2, z+1, 3)
(T^2) + (T + 1)*x + (3)*x^2
```

Some arithmetic:

```
sage: x^3
(T + 31) + (1)*x + (0)*x^2
sage: 3 * x**15 * T**2 + x - T
(3*T^7 + 90*T^6 + 110*T^5 + 20*T^4 + 58*T^3 + 26*T^2 + 124*T) + (15*T^6 + 110*T^5 + 35*T^4 + 63*
```

Retrieve coefficients (output is zero-padded):

```
sage: x^10
(3*T^2 + 61*T + 8) + (T^3 + 93*T^2 + 12*T + 40)*x + (3*T^2 + 61*T + 9)*x^2
sage: (x^10).coeffs()
[[8, 61, 3, 0], [40, 12, 93, 1], [9, 61, 3, 0]]
```

TODO: write an example checking multiplication of these polynomials against Sage's ordinary quotient ring arithmetic. I can't seem to get the quotient ring stuff happening right now...

create_element (*p0, p1, p2, check=True*)

Creates the element $p_0 + p_1 * x + p_2 * x^2$, where pi's are polynomials in T.

INPUT:

- `p0, p1, p2` - coefficients; must be coercable into poly_ring()
- `check` - bool (default True): whether to carry out coercion

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: A, z = R.poly_ring().objgen()
sage: R.create_element(z^2, z+1, 3)
(T^2) + (T + 1)*x + (3)*x^2
```

gens()
Return a list [x, T] where x and T are the generators of the ring (as element *of this ring*).
Note: I have no idea if this is compatible with the usual Sage 'gens' interface.
EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: x, T = R.gens()
sage: x
(0) + (1)*x + (0)*x^2
sage: T
(T) + (0)*x + (0)*x^2
```

poly_ring()
Return the underlying polynomial ring in T.
EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: R.poly_ring()
Univariate Polynomial Ring in T over Ring of integers modulo 125
```

class **SpecialCubicQuotientRingElement**(*parent, p0, p1, p2, check=True*)
An element of a SpecialCubicQuotientRing.

coeffs()
Returns list of three lists of coefficients, corresponding to the x^0, x^1, x^2 coefficients. The lists are zero padded to the same length. The list entries belong to the base ring.
EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: p = R.create_element(t, t^2 - 2, 3)
sage: p.coeffs()
[[0, 1, 0], [123, 0, 1], [3, 0, 0]]
```

scalar_multiply(*scalar*)
Multiplies this element by a scalar, i.e. just multiply each coefficient of x^j by the scalar.
INPUT:

- `scalar` - either an element of base_ring, or an element of poly_ring.

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: x, T = R.gens()
sage: f = R.create_element(2, t, t^2 - 3)
sage: f
(2) + (T)*x + (T^2 + 122)*x^2
sage: f.scalar_multiply(2)
(4) + (2*T)*x + (2*T^2 + 119)*x^2
sage: f.scalar_multiply(t)
(2*T) + (T^2)*x + (T^3 + 122*T)*x^2
```

shift(*n*)
Returns this element multiplied by T^n.
EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: f = R.create_element(2, t, t^2 - 3)
sage: f
(2) + (T)*x + (T^2 + 122)*x^2
sage: f.shift(1)
(2*T) + (T^2)*x + (T^3 + 122*T)*x^2
sage: f.shift(2)
(2*T^2) + (T^3)*x + (T^4 + 122*T^2)*x^2
```

square()

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: x, T = R.gens()

sage: f = R.create_element(1 + 2*t + 3*t^2, 4 + 7*t + 9*t^2, 3 + 5*t + 11*t^2)
sage: f.square()
(73*T^5 + 16*T^4 + 38*T^3 + 39*T^2 + 70*T + 120) + (121*T^5 + 113*T^4 + 73*T^3 + 8*T^2 + 51*
```

class SpecialHyperellipticQuotientElement(*parent, val=0, offset=0, check=True*)

change_ring(*R*)

coeffs(*R=None*)

diff()

extract_pow_y(*k*)

max_pow_y()

min_pow_y()

truncate_neg(*n*)

SpecialHyperellipticQuotientRing(**args*)

class SpecialHyperellipticQuotientRing_class(*Q, R=None, invert_y=True*)

Q()

base_extend(*R*)

change_ring(*R*)

curve()

degree()

gens()

is_field()

monomial(*i, j, b=None*)

Returns $by^j x^i$, computed quickly.

monomial_diff_coeffs(*i, j*)

The key here is that the formula for $d(x^i y^j)$ is messy in terms of i, but varies nicely with j.

$$d(x^i y^j) = y^{j-1}(2ix^{i-1}y^2 + j(A_i(x) + B_i(x)y^2))\frac{dx}{2y}$$

Where A, B have degree at most $n-1$ for each i. Pre-compute A_i, B_i for each i the "hard" way, and the rest are easy.

monomial_diff_coeffs_matrices()

monsky_washnitzer()

prime()

x()

y()

adjusted_prec(*p, prec*)

Computes how much precision is required in matrix_of_frobenius to get an answer correct to prec p-adic digits.

The issue is that the algorithm used in matrix_of_frobenius sometimes performs divisions by p, so precision is lost during the algorithm.

The estimate returned by this function is based on Kedlaya's result (Lemmas 2 and 3 of "Counting Points on Hyperelliptic Curves..."), which implies that if we start with M p-adic digits, the total precision loss is at most $1 + \lfloor \log_p(2M-3) \rfloor$ p-adic digits. (This estimate is somewhat less than the amount you would expect by naively counting the number of divisions by p.)

INPUT:

- `p` - a prime = 5
- `prec` - integer, desired output precision, = 1

OUTPUT: adjusted precision (usually slightly more than prec)

frobenius_expansion_by_newton(*Q, p, M*)

Computes the action of Frobenius on dx/y and on xdx/y, using Newton's method (as suggested in Kedlaya's paper).

(This function does *not* yet use the cohomology relations - that happens afterwards in the "reduction" step.)

More specifically, it finds F_0 and F_1 in the quotient ring $R[x,T]/(T - Q(x))$, such that

$$F(dx/y) = T^{-r} F0 dx/y, \text{ and } F(xdx/y) = T^{-r} F1 dx/y$$

where

$$r = ((2M-3)p - 1)/2.$$

(Here T is $y^2 = z^{-2}$, and R is the coefficient ring of Q.)

F_0 and F_1 are computed in the SpecialCubicQuotientRing associated to Q, so all powers of x^j for $j \geq 3$ are reduced to powers of T.

INPUT:

- `Q` - cubic polynomial of the form $Q(x) = x^3 + ax + b$, whose coefficient ring is a $Z/(p^M)Z$-algebra
- `p` - residue characteristic of the p-adic field
- `M` - p-adic precision of the coefficient ring (this will be used to determine the number of Newton iterations)

OUTPUT:

- `F0, F1` - elements of SpecialCubicQuotientRing(Q), as described above
- `r` - non-negative integer, as described above

frobenius_expansion_by_series(*Q, p, M*)

Computes the action of Frobenius on dx/y and on x dx/y, using a series expansion.

(This function computes the same thing as frobenius_expansion_by_newton(), using a different method. Theoretically the Newton method should be asymptotically faster, when the precision gets large. However, in practice, this functions seems to be marginally faster for moderate precision, so I'm keeping it here until I figure out exactly why it's faster.)

(This function does *not* yet use the cohomology relations - that happens afterwards in the "reduction" step.)

More specifically, it finds F0 and F1 in the quotient ring $R[x,T]/(T - Q(x))$, such that $F(dx/y) = T^{-r}F0dx/y$, and $F(xdx/y) = T^{-r}F1dx/y$ where $r = ((2M-3)p-1)/2$. (Here T is $y^2 = z^{-2}$, and R is the coefficient ring of Q.)

F_0 and F_1 are computed in the SpecialCubicQuotientRing associated to Q, so all powers of x^j for $j \geq 3$ are reduced to powers of T.

It uses the sum

$$F0 = \sum_{k=0}^{M-2} \binom{-1/2}{k} px^{p-1}E^k T^{(M-2-k)p}$$

and

$$F1 = x^p F0,$$
$$where`E = Q(x^p) - Q(x)^p`.$$

INPUT:

- `Q` - cubic polynomial of the form $Q(x) = x^3 + ax + b$, whose coefficient ring is a $\mathbf{Z}/(p^M)\mathbf{Z}$ -algebra
- `p` - residue characteristic of the p-adic field
- `M` - p-adic precision of the coefficient ring (this will be used to determine the number of terms in the series)

OUTPUT:

- `F0, F1` - elements of SpecialCubicQuotientRing(Q), as described above
- `r` - non-negative integer, as described above

helper_matrix(*Q*)

Computes the (constant) matrix used to calculate the linear combinations of the $d(x^i y^j)$ needed to eliminate the negative powers of y in the cohomology (i.e. in reduce_negative()).

INPUT:

- `Q` - cubic polynomial

lift(*x*)

Tries to call x.lift(), presumably from the p-adics to ZZ.

If this fails, it assumes the input is a power series, and tries to lift it to a power series over QQ.

This function is just a very kludgy solution to the problem of trying to make the reduction code (below) work over both Zp and Zp[[t]].

matrix_of_frobenius(*Q, p, M, trace=None, compute_exact_forms=False*)

Computes the matrix of Frobenius on Monsky-Washnitzer cohomology, with respect to the basis $(dx/y, xdx/y)$.

INPUT:

- `Q` - cubic polynomial $Q(x) = x^3 + ax + b$ defining an elliptic curve E by $y^2 = Q(x)$. The coefficient ring of Q should be a $\mathbf{Z}/(p^M)\mathbf{Z}$-algebra in which the matrix of frobenius will be constructed.
- `p` - prime = 5 for which E has good reduction
- `M` - integer = 2; p -adic precision of the coefficient ring
- `trace` - (optional) the trace of the matrix, if known in advance. This is easy to compute because it's just the a_p of the curve. If the trace is supplied, matrix_of_frobenius will use it to speed the computation (i.e. we know the determinant is p, so we have two conditions, so really only column of the matrix needs to be computed. It's actually a little more complicated than that, but that's the basic idea.) If trace=None,

then both columns will be computed independently, and you can get a strong indication of correctness by verifying the trace afterwards.

Warning: THE RESULT WILL NOT NECESSARILY BE CORRECT TO M p-ADIC DIGITS. If you want prec digits of precision, you need to use the function adjusted_prec(), and then you need to reduce the answer mod p^{prec} at the end.

OUTPUT: 2x2 matrix of frobenius on Monsky-Washnitzer cohomology, with entries in the coefficient ring of Q.

EXAMPLES: A simple example:

```
sage: p = 5
sage: prec = 3
sage: M = monsky_washnitzer.adjusted_prec(p, prec)
sage: M
5
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4), p, M)
sage: A
[3090  187]
[2945  408]
```

But the result is only accurate to prec digits:

```
sage: B = A.change_ring(Integers(p**prec))
sage: B
[90 62]
[70 33]
```

Check trace (123 = -2 mod 125) and determinant:

```
sage: B.det()
5
sage: B.trace()
123
sage: EllipticCurve([-1, 1/4]).ap(5)
-2
```

Try using the trace to speed up the calculation:

```
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4),
...                                             p, M, -2)
sage: A
[2715  187]
[1445  408]
```

Hmmm... it looks different, but that's because the trace of our first answer was only -2 modulo 5^3, not -2 modulo 5^5. So the right answer is:

```
sage: A.change_ring(Integers(p**prec))
     [90 62]
     [70 33]
```

Check it works with only one digit of precision:

```
sage: p = 5
sage: prec = 1
sage: M = monsky_washnitzer.adjusted_prec(p, prec)
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4), p, M)
sage: A.change_ring(Integers(p))
[0 2]
[0 3]
```

Here's an example that's particularly badly conditioned for using the trace trick:

```
sage: p = 11
sage: prec = 3
sage: M = monsky_washnitzer.adjusted_prec(p, prec)
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 + 7*x + 8, p, M)
sage: A.change_ring(Integers(p**prec))
[1144  176]
[ 847  185]
```

The problem here is that the top-right entry is divisible by 11, and the bottom-left entry is divisible by 11^2. So when you apply the trace trick, neither $F(dx/y)$ nor $F(xdx/y)$ is enough to compute the whole matrix to the desired precision, even if you try increasing the target precision by one. Nevertheless, `matrix_of_frobenius` knows how to get the right answer by evaluating $F((x+1)dx/y)$ instead:

```
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 + 7*x + 8, p, M, -2)
sage: A.change_ring(Integers(p**prec))
[1144  176]
[ 847  185]
```

The running time is about `O(p*prec**2)` (times some logarithmic factors), so it's feasible to run on fairly large primes, or precision (or both?!?!):

```
sage: p = 10007
sage: prec = 2
sage: M = monsky_washnitzer.adjusted_prec(p, prec)
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(            # long time
...                               x^3 - x + R(1/4), p, M)   # long time
sage: B = A.change_ring(Integers(p**prec)); B               # long time
[74311982 57996908]
[95877067 25828133]
sage: B.det()                                               # long time
10007
sage: B.trace()                                             # long time
66
sage: EllipticCurve([-1, 1/4]).ap(10007)                    # long time
66
```

```
sage: p = 5
sage: prec = 300
sage: M = monsky_washnitzer.adjusted_prec(p, prec)
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(            # long time
...                               x^3 - x + R(1/4), p, M)   # long time
sage: B = A.change_ring(Integers(p**prec))                  # long time
sage: B.det()                                               # long time
```

```
5
sage: -B.trace()                                                    # long time
2
sage: EllipticCurve([-1, 1/4]).ap(5)                                # long time
-2
```

Let's check consistency of the results for a range of precisions:

```
sage: p = 5
sage: max_prec = 60
sage: M = monsky_washnitzer.adjusted_prec(p, max_prec)
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4), p, M)          # long time
sage: A = A.change_ring(Integers(p**max_prec))                      # long time
sage: result = []                                                   # long time
sage: for prec in range(1, max_prec):                               # long time
...       M = monsky_washnitzer.adjusted_prec(p, prec)              # long time
...       R.<x> = PolynomialRing(Integers(p^M),'x')                 # long time
...       B = monsky_washnitzer.matrix_of_frobenius(                # long time
...                           x^3 - x + R(1/4), p, M)               # long time
...       B = B.change_ring(Integers(p**prec))                      # long time
...       result.append(B == A.change_ring(                         # long time
...                                   Integers(p**prec)))           # long time
sage: result == [True] * (max_prec - 1)                             # long time
True
```

The remaining examples discuss what happens when you take the coefficient ring to be a power series ring; i.e. in effect you're looking at a family of curves.

The code does in fact work...

```
sage: p = 11
sage: prec = 3
sage: M = monsky_washnitzer.adjusted_prec(p, prec)
sage: S.<t> = PowerSeriesRing(Integers(p**M), default_prec=4)
sage: a = 7 + t + 3*t^2
sage: b = 8 - 6*t + 17*t^2
sage: R.<x> = PolynomialRing(S)
sage: Q = x**3 + a*x + b
sage: A = monsky_washnitzer.matrix_of_frobenius(Q, p, M)     # long time
sage: B = A.change_ring(PowerSeriesRing(Integers(p**prec), 't', default_prec=4))        # long t
sage: B                                                             # long time
[1144 + 264*t + 841*t^2 + 1025*t^3 + O(t^4)  176 + 1052*t + 216*t^2 + 523*t^3 + O(t^4)]
[   847 + 668*t + 81*t^2 + 424*t^3 + O(t^4)   185 + 341*t + 171*t^2 + 642*t^3 + O(t^4)]
```

The trace trick should work for power series rings too, even in the badly- conditioned case. Unfortunately I don't know how to compute the trace in advance, so I'm not sure exactly how this would help. Also, I suspect the running time will be dominated by the expansion, so the trace trick won't really speed things up anyway. Another problem is that the determinant is not always p:

```
sage: B.det()                                                       # long time
11 + 484*t^2 + 451*t^3 + O(t^4)
```

However, it appears that the determinant always has the property that if you substitute t - 11t, you do get the constant series p (mod p**prec). Similarly for the trace. And since the parameter only really makes sense when it's divisible by p anyway, perhaps this isn't a problem after all.

matrix_of_frobenius_hyperelliptic(*Q, p=None, prec=None, M=None*)

reduce_all(*Q, p, coeffs, offset, compute_exact_form=False*)

Applies cohomology relations to reduce all terms to a linear combination of dx/y and xdx/y.

INPUT:

- `Q` - cubic polynomial
- `coeffs` - list of length 3 lists. The i^{th} list [a, b, c] represents $y^{2(i-offset)}(a + bx + cx^2)dx/y$.
- `offset` - nonnegative integer

OUTPUT:

- `A, B` - pair such that the input differential is cohomologous to (A + Bx) dx/y.

Note: The algorithm operates in-place, so the data in coeffs is destroyed.

EXAMPLE:

```
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)
sage: coeffs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_all(Q, 5, coeffs, 1)
 (21, 106)
```

reduce_negative(*Q, p, coeffs, offset, exact_form=None*)

Applies cohomology relations to incorporate negative powers of y into the y^0 term.

INPUT:

- `p` - prime
- `Q` - cubic polynomial
- `coeffs` - list of length 3 lists. The i^{th} list [a, b, c] represents $y^{2(i-offset)}(a + bx + cx^2)dx/y$.
- `offset` - nonnegative integer

OUTPUT: The reduction is performed in-place. The output is placed in coeffs[offset]. Note that coeffs[i] will be meaningless for i offset after this function is finished.

EXAMPLE:

```
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)
sage: coeffs = [[10, 15, 20], [1, 2, 3], [4, 5, 6], [7, 8, 9]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_negative(Q, 5, coeffs, 3)
sage: coeffs[3]
 [28, 52, 9]


sage: R.<x> = Integers(7^3)['x']
sage: Q = x^3 - x + R(1/4)
sage: coeffs = [[7, 14, 21], [1, 2, 3], [4, 5, 6], [7, 8, 9]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_negative(Q, 7, coeffs, 3)
sage: coeffs[3]
 [245, 332, 9]
```

reduce_positive(*Q, p, coeffs, offset, exact_form=None*)

Applies cohomology relations to incorporate positive powers of y into the y^0 term.

INPUT:

- `Q` - cubic polynomial
- `coeffs` - list of length 3 lists. The i^{th} list [a, b, c] represents $y^{2(i-offset)}(a+bx+cx^2)dx/y$.
- `offset` - nonnegative integer

OUTPUT: The reduction is performed in-place. The output is placed in coeffs[offset]. Note that coeffs[i] will be meaningless for i offset after this function is finished.

EXAMPLE:

```
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)

sage: coeffs = [[1, 2, 3], [10, 15, 20]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_positive(Q, 5, coeffs, 0)
sage: coeffs[0]
 [16, 102, 88]

sage: coeffs = [[9, 8, 7], [10, 15, 20]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_positive(Q, 5, coeffs, 0)
sage: coeffs[0]
 [24, 108, 92]
```

reduce_zero(*Q, coeffs, offset, exact_form=None*)

Applies cohomology relation to incorporate x^2y^0 term into x^0y^0 and x^1y^0 terms.

INPUT:

- `Q` - cubic polynomial
- `coeffs` - list of length 3 lists. The i^{th} list [a, b, c] represents $y^{2(i-offset)}(a+bx+cx^2)dx/y$.
- `offset` - nonnegative integer

OUTPUT: The reduction is performed in-place. The output is placed in coeffs[offset]. This method completely ignores coeffs[i] for i != offset.

EXAMPLE:

```
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)
sage: coeffs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_zero(Q, coeffs, 1)
sage: coeffs[1]
 [6, 5, 0]
```

transpose_list(*input*)

INPUT:

- `input` - a list of lists, each list of the same length

OUTPUT:

- `output` - a list of lists such that output[i][j] = input[j][i]

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.monsky_washnitzer import transpose_list
sage: L = [[1, 2], [3, 4], [5, 6]]
sage: transpose_list(L)
[[1, 3, 5], [2, 4, 6]]
```

38.20 p-adic L-functions of elliptic curves

To an elliptic curve E over the rational numbers and a prime p, one can associate a p-adic L-function; at least if E does not have additive reduction at p. This function is defined by interpolation of L-values of E at twists. Through the main conjecture of Iwasawa theory it should also be equal to a characteristic series of a certain Selmer group.

If E is ordinary, then it is an element of $\mathbf{Z}_p[\![T]\!]$ and according to the p-adic version of the Birch and Swinnerton-Dyer conjecture [MTT], the order of vanishing at $T = 0$ is just the rank of $E(\mathbf{Q})$ or this rank plus one if the reduction at p is split multiplicative.

If E is supersingular, the series will have coefficients in a quadratic extension of $\mathbf{Z}_p$. We have also implemented the p-adic L-series as formulated by Perrin-Riou [BP], which has coefficients in the Dieudonne $D_pE = H^1_{dR}(E/\mathbf{Q}_p)$ module of E. There is a different description by Pollack [Po] which is not available here.

See [SW] for more details.

REFERENCES:

- [MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.
- [BP] Dominique Bernardi and Bernadette Perrin-Riou, Variante p-adique de la conjecture de Birch et Swinnerton-Dyer (le cas supersingulier), C. R. Acad. Sci. Paris, Ser I. Math, 317 (1993), no 3, 227-232.
- [Po] Robert Pollack, On the p-adic L-function of a modular form at supersingular prime, Duke Math. J. 118 (2003), no 3, 523-558.
- [SW] William Stein and Christian Wuthrich, Computations About Tate-Shafarevich Groups using Iwasawa theory, preprint 2009.

AUTHORS:

- William Stein (2007-01-01): first version
- chris wuthrich (22/05/2007): changed minor issues and added supersingular things
- chris wuthrich (11/2008): added quadratic_twists

class pAdicLseries (*E, p, use_eclib=False, normalize='L_ratio'*)

The p-adic L-series of an elliptic curve.

EXAMPLES: An ordinary example:

```
sage: e = EllipticCurve('389a')
sage: L = e.padic_lseries(5)
sage: L.series(0)
...
ValueError: n (=0) must be a positive integer
sage: L.series(1)
O(T^1)
sage: L.series(2)
```

```
O(5^4) + O(5)*T + (4 + O(5))*T^2 + (2 + O(5))*T^3 + (3 + O(5))*T^4 + O(T^5)
sage: L.series(3, prec=10)
O(5^5) + O(5^2)*T + (4 + 4*5 + O(5^2))*T^2 + (2 + 4*5 + O(5^2))*T^3 + (3 + O(5^2))*T^4 + (1 + O(
sage: L.series(2,quadratic_twist=-3)
2 + 4*5 + 4*5^2 + O(5^4) + O(5)*T + (1 + O(5))*T^2 + (4 + O(5))*T^3 + O(5)*T^4 + O(T^5)
```

A prime p such that E[p] is reducible:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.series(1)
5 + O(5^2) + O(T)
sage: L.series(2)
5 + 4*5^2 + O(5^3) + O(5^0)*T + O(5^0)*T^2 + O(5^0)*T^3 + O(5^0)*T^4 + O(T^5)
sage: L.series(3)
5 + 4*5^2 + 4*5^3 + O(5^4) + O(5)*T + O(5)*T^2 + O(5)*T^3 + O(5)*T^4 + O(T^5)
```

the load-dumps test:

```
sage: lp = EllipticCurve('11a').padic_lseries(5)
sage: lp == loads(dumps(lp))
True
```

alpha (*prec=20*)

Return a p-adic root α of the polynomial $x^2 - a_p x + p$ with $ord_p(\alpha) < 1$. In the ordinary case this is just the unit root.

INPUT: - `prec` - positive integer, the p-adic precision of the root.

EXAMPLES: Consider the elliptic curve 37a:

```
sage: E = EllipticCurve('37a')
```

An ordinary prime:

```
sage: L = E.padic_lseries(5)
sage: alpha = L.alpha(10); alpha
3 + 2*5 + 4*5^2 + 2*5^3 + 5^4 + 4*5^5 + 2*5^7 + 5^8 + 5^9 + O(5^10)
sage: alpha^2 - E.ap(5)*alpha + 5
O(5^10)
```

A supersingular prime:

```
sage: L = E.padic_lseries(3)
sage: alpha = L.alpha(10); alpha
(1 + O(3^10))*alpha
sage: alpha^2 - E.ap(3)*alpha + 3
(O(3^10))*alpha^2 + (O(3^11))*alpha + (O(3^11))
```

A reducible prime:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.alpha(5)
1 + 4*5 + 3*5^2 + 2*5^3 + 4*5^4 + O(5^5)
```

elliptic_curve()

Return the elliptic curve to which this p-adic L-series is associated.

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.elliptic_curve()
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
```

measure (*a, n, prec, quadratic_twist=1*)

Return the measure on $\mathbf{Z}_p^\times$ defined by

$$\mu_{E,\alpha}^+(a+p^n\mathbf{Z}_p) = \frac{1}{\alpha^n}\left[\frac{a}{p^n}\right]^+ - \frac{1}{\alpha^{n+1}}\left[\frac{a}{p^{n-1}}\right]^+$$

where $[\cdot]^+$ is the modular symbol. This is used to define this p-adic L-function (at least when the reduction is good).

The optional argument `quadratic_twist` replaces E by the twist in the above formula, but the twisted modular symbol is computed using a sum over modular symbols of E rather then finding the modular symbols for the twist.

Note that the normalisation is not correct at this stage: use `_quotient_of periods` and `_quotient_of periods_to_twist` to correct.

Note also that this function does not check if the condition on the `quadratic_twist=D` is satisfied. So the result will only be correct if for each prime ℓ dividing D, we have $ord_\ell(N) <= ord_\ell(D)$, where N is the conductor of the curve.

INPUT:

- `a` - an integer
- `n` - a non-negative integer
- `prec` - an integer
- `quadratic_twist` (default = 1) - a fundamental discriminant of a quadratic field, should be coprime to the conductor of E

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: L = E.padic_lseries(5)
sage: L.measure(1,2, prec=9)
2 + 3*5 + 4*5^3 + 2*5^4 + 3*5^5 + 3*5^6 + 4*5^7 + 4*5^8 + O(5^9)
sage: L.measure(1,2, quadratic_twist=8,prec=15)
O(5^15)
sage: L.measure(1,2, quadratic_twist=-4,prec=15)
4 + 4*5 + 4*5^2 + 3*5^3 + 2*5^4 + 5^5 + 3*5^6 + 5^8 + 2*5^9 + 3*5^12 + 2*5^13 + 4*5^14 + O(5

sage: E = EllipticCurve('11a1')
sage: a = E.quadratic_twist(-3).padic_lseries(5).measure(1,2,prec=15)
sage: b = E.padic_lseries(5).measure(1,2, quadratic_twist=-3,prec=15)
sage: a == b/E.padic_lseries(5)._quotient_of_periods_to_twist(-3)
True
```

modular_symbol (*r, sign=1, quadratic_twist=1*)

Return the modular symbol evaluated at r. This is used to compute this p-adic L-series.

Note that the normalisation is not correct at this stage: use `_quotient_of periods_to_twist` to correct.

Note also that this function does not check if the condition on the quadratic_twist=D is satisfied. So the result will only be correct if for each prime ℓ dividing D, we have $ord_{\ell(N)} <= ord_\ell(D)$, where N is the conductor of the curve.

INPUT:

- `r` - a cusp given as either a rational number or oo
- `sign` - +1 (default) or -1 (only implemented without twists)
- `quadratic_twist` - a fundamental discriminant of a quardratic field or +1 (default)

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: lp = E.padic_lseries(5)
sage: [lp.modular_symbol(r) for r in [0,1/5,oo,1/11]]
```

```
[1/5, 6/5, 0, 0]
sage: [lp.modular_symbol(r,sign=-1) for r in [0,1/3,oo,1/7]]
[0, 1, 0, -1]
sage: [lp.modular_symbol(r,quadratic_twist=-20) for r in [0,1/5,oo,1/11]]
[2, 2, 0, 1]

sage: lpt = E.quadratic_twist(-3).padic_lseries(5)
sage: et = E.padic_lseries(5)._quotient_of_periods_to_twist(-3)
sage: lpt.modular_symbol(0) == lp.modular_symbol(0,quadratic_twist=-3)/et
True
```

order_of_vanishing()

Return the order of vanishing of this p-adic L-series.

The output of this function is provably correct, due to a theorem of Kato [Ka]. This function will terminate if and only if the Mazur-Tate-Teitelbaum analogue [MTT] of the BSD conjecture about the rank of the curve is true and the subgroup of elements of p-power order in the Shafarevich-Tate group of this curve is finite. I.e. if this function terminates (with no errors!), then you may conclude that the p-adic BSD rank conjecture is true and that the p-part of Sha is finite.

NOTE: currently p must be a prime of good ordinary reduction.

REFERENCES:

- [MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.
- [Ka] Kayuza Kato, p-adic Hodge theory and values of zeta functions of modular forms, Cohomologies p-adiques et applications arithmetiques III, Asterisque vol 295, SMF, Paris, 2004.

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(3)
sage: L.order_of_vanishing()
0
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.order_of_vanishing()
0
sage: L = EllipticCurve('37a').padic_lseries(5)
sage: L.order_of_vanishing()
1
sage: L = EllipticCurve('43a').padic_lseries(3)
sage: L.order_of_vanishing()
1
sage: L = EllipticCurve('37b').padic_lseries(3)
sage: L.order_of_vanishing()
0
```

We verify that Sha(E)(p) is finite for p=3,5,7 for the first curve of rank 2:

```
sage: e = EllipticCurve('389a')
sage: for p in primes(3,10):
...       print p, e.padic_lseries(p).order_of_vanishing()
3 2
5 2
7 2
```

prime()

Returns the prime p as in 'p-adic L-function'.

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.prime()
5
```

teichmuller(*prec*)

Return Teichmuller lifts to the given precision.

INPUT:

•`prec` - a positive integer.

OUTPUT:

•a list of p-adic numbers, the cached Teichmuller lifts

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(7)
sage: L.teichmuller(1)
[0, 1, 2, 3, 4, 5, 6]
sage: L.teichmuller(2)
[0, 1, 30, 31, 18, 19, 48]
```

class pAdicLseriesOrdinary(*E, p, use_eclib=False, normalize='L_ratio'*)

is_ordinary()

Return True if the elliptic that this L-function is attached to is ordinary.

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.is_ordinary()
True
```

is_supersingular()

Return True if the elliptic that this L function is attached to is supersingular.

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.is_supersingular()
False
```

power_series(*n=2, quadratic_twist=1, prec=5*)

Returns the n-th approximation to the p-adic L-series as a power series in T (corresponding to $\gamma - 1$ with $\gamma = 1 + p$ as a generator of $1 + p\mathbf{Z}_p$). Each coefficient is a p-adic number whose precision is provably correct.

Here the normalization of the p-adic L-series is chosen such that $L_p(E, 1) = (1 - 1/\alpha)^2 L(E, 1)/\Omega_E$ where α is the unit root of the characteristic polynomial of Frobenius on $T_p E$ and Ω_E is the Neron period of E.

INPUT:

•`n` - (default: 2) a positive integer

•`quadratic_twist` - (default: +1) a fundamental discriminant of a qudratic field, coprime to the conductor of the curve

•`prec` - (default: 5) maximal number of terms of the series to compute; to compute as many as possible just give a very large number for `prec`; the result will still be correct.

ALIAS: power_series is identical to series.

EXAMPLES: We compute some p-adic L-functions associated to the elliptic curve 11a:

```
sage: E = EllipticCurve('11a')
sage: p = 3
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(3)
2 + 3 + 3^2 + 2*3^3 + O(3^5) + (1 + 3 + O(3^2))*T + (1 + 2*3 + O(3^2))*T^2 + O(3)*T^3 + (2*3
```

Another example at a prime of bad reduction, where the p-adic L-function has an extra 0 (compared to the non p-adic L-function):

```
sage: E = EllipticCurve('11a')
sage: p = 11
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(2)
O(11^4) + (10 + O(11))*T + (6 + O(11))*T^2 + (2 + O(11))*T^3 + (5 + O(11))*T^4 + O(T^5)
```

We compute a p-adic L-function that vanishes to order 2:

```
sage: E = EllipticCurve('389a')
sage: p = 3
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(1)
O(T^1)
sage: L.series(2)
O(3^4) + O(3)*T + (2 + O(3))*T^2 + O(T^3)
sage: L.series(3)
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + (1 + 3 + O(3^2))*T^4 + O(T^5)
```

Checks if the precision can be changed (trac 5846):

```
sage: L.series(3,prec=4)
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + O(T^4)
sage: L.series(3,prec=6)
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + (1 + 3 + O(3^2))*T^4 + (1 + O(
```

Rather than computing the p-adic L-function for the curve '15523a1', one can compute it as a quadratic_twist:

```
sage: E = EllipticCurve('43a1')
sage: lp = E.padic_lseries(3)
sage: lp.series(2,quadratic_twist=-19)
2 + 2*3 + 2*3^2 + O(3^4) + (1 + O(3))*T + (1 + O(3))*T^2 + O(T^3)
sage: E.quadratic_twist(-19).label()    #optional --- conductor is greater than 10000
'15523a1'
```

This proves that the rank of '15523a1' is zero, even if `mwrank` can not determine this.

series (*n=2, quadratic_twist=1, prec=5*)

Returns the n-th approximation to the p-adic L-series as a power series in T (corresponding to $\gamma - 1$ with $\gamma = 1 + p$ as a generator of $1 + p\mathbf{Z}_p$). Each coefficient is a p-adic number whose precision is provably correct.

Here the normalization of the p-adic L-series is chosen such that $L_p(E,1) = (1 - 1/\alpha)^2 L(E,1)/\Omega_E$ where α is the unit root of the characteristic polynomial of Frobenius on T_pE and Ω_E is the Neron period of E.

INPUT:

- `n` - (default: 2) a positive integer
- `quadratic_twist` - (default: +1) a fundamental discriminant of a qudratic field, coprime to the conductor of the curve
- `prec` - (default: 5) maximal number of terms of the series to compute; to compute as many as possible just give a very large number for `prec`; the result will still be correct.

ALIAS: power_series is identical to series.

EXAMPLES: We compute some p-adic L-functions associated to the elliptic curve 11a:

```
sage: E = EllipticCurve('11a')
sage: p = 3
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(3)
2 + 3 + 3^2 + 2*3^3 + O(3^5) + (1 + 3 + O(3^2))*T + (1 + 2*3 + O(3^2))*T^2 + O(3)*T^3 + (2*3
```

Another example at a prime of bad reduction, where the p-adic L-function has an extra 0 (compared to the non p-adic L-function):

```
sage: E = EllipticCurve('11a')
sage: p = 11
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(2)
O(11^4) + (10 + O(11))*T + (6 + O(11))*T^2 + (2 + O(11))*T^3 + (5 + O(11))*T^4 + O(T^5)
```

We compute a p-adic L-function that vanishes to order 2:

```
sage: E = EllipticCurve('389a')
sage: p = 3
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(1)
O(T^1)
sage: L.series(2)
O(3^4) + O(3)*T + (2 + O(3))*T^2 + O(T^3)
sage: L.series(3)
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + (1 + 3 + O(3^2))*T^4 + O(T^5)
```

Checks if the precision can be changed (trac 5846):

```
sage: L.series(3,prec=4)
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + O(T^4)
sage: L.series(3,prec=6)
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + (1 + 3 + O(3^2))*T^4 + (1 + O(
```

Rather than computing the p-adic L-function for the curve '15523a1', one can compute it as a quadratic_twist:

```
sage: E = EllipticCurve('43a1')
sage: lp = E.padic_lseries(3)
sage: lp.series(2,quadratic_twist=-19)
2 + 2*3 + 2*3^2 + O(3^4) + (1 + O(3))*T + (1 + O(3))*T^2 + O(T^3)
sage: E.quadratic_twist(-19).label()    #optional --- conductor is greater than 10000
'15523a1'
```

This proves that the rank of '15523a1' is zero, even if `mwrank` can not determine this.

class pAdicLseriesSupersingular(*E, p, use_eclib=False, normalize='L_ratio'*)

Dp_valued_height(*prec=20*)

Returns the canonical p-adic height with values in the Dieudonne module $D_p(E)$. It is defined to be

$$h_\eta \cdot \omega - h_\omega \cdot \eta$$

where h_η is made out of the sigma function of Bernardi and h_ω is log_E^2. The answer `v` is given as `v[1]*omega + v[2]*eta`. The coordinates of `v` are dependent of the Weierstrass equation.

EXAMPLES:

```
sage: E = EllipticCurve('53a')
sage: L = E.padic_lseries(5)
sage: h = L.Dp_valued_height(7)
sage: h(E.gens()[0])
(3*5 + 5^2 + 2*5^3 + 3*5^4 + 4*5^5 + 5^6 + 5^7 + O(5^8), 5^2 + 4*5^4 + 2*5^7 + 3*5^8 + O(5^9
```

Dp_valued_regulator (*prec=20, v1=0, v2=0*)

Returns the canonical p-adic regulator with values in the Dieudonne module $D_p(E)$ as defined by Perrin-Riou using the p-adic height with values in $D_p(E)$. The result is written in the basis ω, $\varphi(\omega)$, and hence the coordinates of the result are independent of the chosen Weierstrass equation.

NOTE: The definition here is corrected with repect to Perrin-Riou's article [PR]. See [SW].

REFERENCES:

- [PR] Perrin Riou, Arithmetique des courbes elliptiques a reduction supersinguliere en p, Experiment. Math. 12 (2003), no. 2, 155-186.
- [SW] William Stein and Christian Wuthrich, Computations About Tate-Shafarevich Groups using Iwasawa theory, preprint 2009.

EXAMPLES:

```
sage: E = EllipticCurve('43a')
sage: L = E.padic_lseries(7)
sage: L.Dp_valued_regulator(7)
(5*7 + 6*7^2 + 4*7^3 + 4*7^4 + 7^5 + 4*7^7 + O(7^8), 4*7^2 + 2*7^3 + 3*7^4 + 7^5 + 6*7^6 + 4
```

Dp_valued_series (*n=3, quadratic_twist=1, prec=5*)

Returns a vector of two components which are p-adic power series. The answer v is such that

$$(1-\varphi)^{-2}\cdot L_p(E,T) = \mathtt{v[1]}\cdot\omega + \mathtt{v[2]}\cdot\varphi(\omega)$$

as an element of the Dieudonne module $D_p(E) = H^1_{dR}(E/\mathbf{Q}_p)$ where ω is the invariant differential and φ is the Frobenius on $D_p(E)$. According to the p-adic Birch and Swinnerton-Dyer conjecture [BP] this function has a zero of order rank of $E(\mathbf{Q})$ and it's leading term is contains the order of the Tate-Shafarevich group, the Tamagawa numbers, the order of the torsion subgroup and the D_p-valued p-adic regulator.

INPUT:

- `n` - (default: 3) a positive integer
- `prec` - (default: 5) a positive integer

REFERENCE:

- [BP] Dominique Bernardi and Bernadette Perrin-Riou, Variante p-adique de la conjecture de Birch et Swinnerton-Dyer (le cas supersingulier), C. R. Acad. Sci. Paris, Ser I. Math, 317 (1993), no 3, 227-232.

EXAMPLES:

```
sage: E = EllipticCurve('14a')
sage: L = E.padic_lseries(5)
sage: L.Dp_valued_series(4)
(1 + 4*5 + 4*5^3 + O(5^4) + (4 + O(5))*T + (1 + O(5))*T^2 + (4 + O(5))*T^3 + (2 + O(5))*T^4
```

bernardi_sigma_function (*prec=20*)

Return the p-adic sigma function of Bernardi in terms of $z = log(t)$. This is the same as `padic_sigma` with `E2 = 0`.

EXAMPLES:

```
sage: E = EllipticCurve('14a')
sage: L = E.padic_lseries(5)
sage: L.bernardi_sigma_function(5) # Todo: some sort of consistency check!?
z + 1/24*z^3 + 29/384*z^5 - 8399/322560*z^7 - 291743/92897280*z^9 - 4364831/5225472*z^10 + 2
```

frobenius (*prec=20, method='mw'*)

This returns a geometric Frobenius φ on the Diedonne module $D_p(E)$ with respect to the basis ω, the invariant differential, and $\eta = x\omega$. It satisfies $\varphi^2 - a_p/p\,\varphi + 1/p = 0$.

INPUT:

- `prec` - (default: 20) a positive integer
- `method` - either 'mw' (default) for Monsky-Washintzer or 'approx' for the method described by Bernardi and Perrin-Riou (much slower)

EXAMPLES:

```
sage: E = EllipticCurve('14a')
sage: L = E.padic_lseries(5)
sage: phi = L.frobenius(5)
sage: phi
[                   2 + 5^2 + 5^4 + O(5^5)     3*5^-1 + 3 + 5 + 4*5^2 + 5^3 + O(5^4)]
[      3 + 3*5^2 + 4*5^3 + 3*5^4 + O(5^5) 3 + 4*5 + 3*5^2 + 4*5^3 + 3*5^4 + O(5^5)]
sage: -phi^2
[5^-1 + O(5^4)         O(5^4)]
[       O(5^5) 5^-1 + O(5^4)]
```

is_ordinary()

is_supersingular()

power_series (*n=3, quadratic_twist=1, prec=5*)

Return the n-th approximation to the p-adic L-series as a power series in T (corresponding to $\gamma - 1$ with $\gamma = 1 + p$ as a generator of $1 + p\mathbf{Z}_p$). Each coefficient is an element of a quadratic extension of the p-adic number whose precision is probably correct.

Here the normalization of the p-adic L-series is chosen such that $L_p(E,1) = (1 - 1/\alpha)^2 L(E,1)/\Omega_E$ where α is the unit root of the characteristic polynomial of Frobenius on T_pE and Ω_E is the Neron period of E.

INPUT:

- `n` - (default: 3) a positive integer
- `prec` - (default: 5) maxima number of terms of the series to compute; to compute as many as possible just give a very large number for prec; the result will still be correct.

ALIAS: power_series is identical to series.

EXAMPLES: A superingular example, where we must compute to higher precision to see anything:

```
sage: e = EllipticCurve('37a')
sage: L = e.padic_lseries(3); L
3-adic L-series of Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: L.series(2)
O(T^3)
sage: L.series(4)          # takes a long time (several seconds)
(O(3))*alpha + (O(3^2)) + ((O(3^-1))*alpha + (2*3^-1 + O(3^0)))*T + ((O(3^-1))*alpha + (2*3^
sage: L.alpha(2).parent()
Univariate Quotient Polynomial Ring in alpha over 3-adic Field with capped
relative precision 2 with modulus (1 + O(3^2))*x^2 + (3 + O(3^3))*x + (3 + O(3^3))
```

series (*n=3, quadratic_twist=1, prec=5*)

Return the n-th approximation to the p-adic L-series as a power series in T (corresponding to $\gamma - 1$ with $\gamma = 1 + p$ as a generator of $1 + p\mathbf{Z}_p$). Each coefficient is an element of a quadratic extension of the p-adic number whose precision is probably correct.

Here the normalization of the p-adic L-series is chosen such that $L_p(E,1) = (1 - 1/\alpha)^2 L(E,1)/\Omega_E$ where α is the unit root of the characteristic polynomial of Frobenius on T_pE and Ω_E is the Neron period of E.

INPUT:

- n - (default: 3) a positive integer
- prec - (default: 5) maxima number of terms of the series to compute; to compute as many as possible just give a very large number for prec; the result will still be correct.

ALIAS: power_series is identical to series.

EXAMPLES: A superingular example, where we must compute to higher precision to see anything:

```
sage: e = EllipticCurve('37a')
sage: L = e.padic_lseries(3); L
3-adic L-series of Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: L.series(2)
O(T^3)
sage: L.series(4)          # takes a long time (several seconds)
(O(3))*alpha + (O(3^2)) + ((O(3^-1))*alpha + (2*3^-1 + O(3^0)))*T + ((O(3^-1))*alpha + (2*3^
sage: L.alpha(2).parent()
Univariate Quotient Polynomial Ring in alpha over 3-adic Field with capped
relative precision 2 with modulus (1 + O(3^2))*x^2 + (3 + O(3^3))*x + (3 + O(3^3))
```

38.21 Modular symbols

To an elliptic curves E over the rational numbers one can associate a space - or better two spaces - of modular symbols of level N, equal to the conductor of E; because E is known to be modular.

There are two implementations of modular symbols, one within sage and the other as part of Cremona's eclib. One can choose here which one is used.

The normalisation of our modular symbols attached to E can be chosen, too. For instance one can make it depended on E rather than on its isogeny class. This is useful for p-adic L-functions.

For more details on modular symbols consult the following

REFERENCES:

- [MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.
- [Cre] John Cremona, Algorithms for modular elliptic curves, Cambridge University Press, 1997.
- [SW] William Stein and Christian Wuthrich, Computations About Tate-Shafarevich Groups using Iwasawa theory, preprint 2009.

AUTHORS:

- William Stein (2007): first version
- Chris Wuthrich (2008): add scaling and reference to eclib

class ModularSymbol()

A modular symbol attached to an elliptic curve, which is the map $\mathbf{Q} \to \mathbf{Q}$ obtained by sending r to the normalized symmetrized (or anti-symmetrized) integral from r to ∞.

This is as defined in [MTT], but normalized to depend on the curve and not only its isogeny class as in [SW].

See the documentation of E.modular_symbol() in Elliptic curves over the rational numbers for help.

REFERENCES:

- [MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.

- [SW] William Stein and Christian Wuthrich, Computations About Tate-Shafarevich Groups using Iwasawa theory, preprint 2009.

base_ring()

Return the base ring for this modular symbol.

EXAMPLES:

```
sage: m = EllipticCurve('11a1').modular_symbol()
sage: m.base_ring()
Rational Field
```

elliptic_curve()

Return the elliptic curve of this modular symbol.

EXAMPLES:

```
sage: m = EllipticCurve('11a1').modular_symbol()
sage: m.elliptic_curve()
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
```

sign()

Return the sign of this elliptic curve modular symbol.

EXAMPLES:

```
sage: m = EllipticCurve('11a1').modular_symbol()
sage: m.sign()
1
sage: m = EllipticCurve('11a1').modular_symbol(sign=-1)
sage: m.sign()
-1
```

class ModularSymbolECLIB(*E, sign, normalize='L_ratio'*)

class ModularSymbolSage(*E, sign, normalize='L_ratio'*)

modular_symbol_space(*E, sign, base_ring, bound=None*)

Creates the space of modular symbols of a given sign over a give base_ring, attached to the isogeny class of elliptic curves.

INPUT:

- `E` - an elliptic curve over $\mathbf{Q}$
- `sign` - integer, -1, 0, or 1
- `base_ring` - ring
- `bound` - (default: None) maximum number of Hecke operators to use to cut out modular symbols factor. If None, use enough to provably get the correct answer.

OUTPUT: a space of modular symbols

EXAMPLES:

```
sage: import sage.schemes.elliptic_curves.ell_modular_symbols
sage: E=EllipticCurve('11a1')
sage: M=sage.schemes.elliptic_curves.ell_modular_symbols.modular_symbol_space(E,-1,GF(37))
sage: M
Modular Symbols space of dimension 1 for Gamma_0(11) of weight 2 with sign -1 over Finite Field
```

38.22 Tate-Shafarevich group

If E is an elliptic curve over a global field K, the Shafarevich-Tate group is the subgroup of elements in $H^1(K, E)$ which map to zero under every global-to-local restriction map $H^1(K, E) \to H^1(K_v, E)$, one for each place v of K. It is known to be a torsion group and the m-torsion is finite for all $m > 1$. It is conjectured to be a finite.

AUTHORS:

- who started this ?
- chris wuthrich (04/09) - reformat docstrings.

class Sha (E)

The Shafarevich-Tate group associated to an elliptic curve.

If E is an elliptic curve over a global field K, the Shafarevich-Tate group is the subgroup of elements in $H^1(K, E)$ which map to zero under every global-to-local restriction map $H^1(K, E) \to H^1(K_v, E)$, one for each place v of K.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: E.sha()
Shafarevich-Tate group for the Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational
```

an (*use_database=False*)

Returns the Birch and Swinnerton-Dyer conjectural order of Sha as a provably corret integer, unless the analytic rank is > 1, in which case this function returns a numerical value.

INPUT: `use_database` – bool (default: False); if True, try to use any databases installed to lookup the analytic order of Sha, if possible. The order of Sha is computed if it can't be looked up.

This result is proved correct if the order of vanishing is 0 and the Manin constant is <= 2.

If the optional parameter `use_database` is True (default: False), this function returns the analytic order of Sha as listed in Cremona's tables, if this curve appears in Cremona's tables.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])    # 11A  = X_0(11)
sage: E.sha().an()
1
sage: E = EllipticCurve([0, -1, 1, 0, 0])        # X_1(11)
sage: E.sha().an()
1

sage: EllipticCurve('14a4').sha().an()
1
sage: EllipticCurve('14a4').sha().an(use_database=True)    # will be faster if you have large
1
```

The smallest conductor curve with nontrivial Sha:

```
sage: E = EllipticCurve([1,1,1,-352,-2689])      # 66b3
sage: E.sha().an()
4
```

The four optimal quotients with nontrivial Sha and conductor <= 1000:

```
sage: E = EllipticCurve([0, -1, 1, -929, -10595])        # 571A
sage: E.sha().an()
4
sage: E = EllipticCurve([1, 1, 0, -1154, -15345])        # 681B
```

```
sage: E.sha().an()
9
sage: E = EllipticCurve([0, -1, 0, -900, -10098])        # 960D
sage: E.sha().an()
4
sage: E = EllipticCurve([0, 1, 0, -20, -42])             # 960N
sage: E.sha().an()
4
```

The smallest conductor curve of rank > 1:

```
sage: E = EllipticCurve([0, 1, 1, -2, 0])        # 389A (rank 2)
sage: E.sha().an()
1.00000000000000
```

The following are examples that require computation of the Mordell-Weil group and regulator:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])                      # 37A  (rank 1)
sage: E.sha().an()
1

sage: E = EllipticCurve("1610f3")
sage: E.sha().an()
4
```

In this case the input curve is not minimal, and if this function didn't transform it to be minimal, it would give nonsense:

```
sage: E = EllipticCurve([0,-432*6^2])
sage: E.sha().an()
1
```

an_numerical (*prec=None, use_database=True, proof=None*)

Return the numerical analytic order of Sha, which is a floating point number in all cases.

INPUT:

- `prec` - integer (default: 53) bits precision – used for the L-series computation, period, regulator, etc.
- `use_database` - whether the rank and generators should be looked up in the database if possible. Default is True
- `proof` - bool or None (default: None, see proof.[tab] or sage.structure.proof) proof option passed onto regulator and rank computation.

Note: See also the an() command, which will return a provably correct integer when the rank is 0 or 1.

Warning: If the curve's generators are not known, computing them may be very time-consuming. Also, computation of the L-series derivative will be time-consuming for large rank and large conductor, and the computation time for this may increase substantially at greater precision. However, use of very low precision less than about 10 can cause the underlying pari library functions to fail.

EXAMPLES:

```
sage: EllipticCurve('11a').sha().an_numerical()
1.00000000000000
sage: EllipticCurve('37a').sha().an_numerical() # long time
1.00000000000000
sage: EllipticCurve('389a').sha().an_numerical() # long time
1.00000000000000
sage: EllipticCurve('66b3').sha().an_numerical()
4.00000000000000
sage: EllipticCurve('5077a').sha().an_numerical() # long time
1.00000000000000
```

A rank 4 curve:

```
sage: EllipticCurve([1, -1, 0, -79, 289]).sha().an_numerical()    # long time
1.00000000000000
```

A rank 5 curve:

```
sage: EllipticCurve([0, 0, 1, -79, 342]).sha().an_numerical(prec=10, proof=False) # long tim
1.0
```

See trac #1115

```
sage: sha=EllipticCurve('37a1').sha()
sage: [sha.an_numerical(prec) for prec in xrange(40,100,10)] # long time
[1.0000000000,
1.0000000000000,
1.0000000000000000,
1.00000000000000000000,
1.000000000000000000000000,
1.0000000000000000000000000000]
```

an_padic(*p, prec=0, use_twists=True*)

Returns the conjectural order of Sha(E), according to the p-adic analogue of the Birch and Swinnerton-Dyer conjecture as formulated in [MTT] and [BP].

REFERENCES:

- [MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.
- [BP] Dominique Bernardi and Bernadette Perrin-Riou, Variante p-adique de la conjecture de Birch et Swinnerton-Dyer (le cas supersingulier), C. R. Acad. Sci. Paris, Ser I. Math, 317 (1993), no 3, 227-232.
- [SW] William Stein and Christian Wuthrich, Computations About Tate-Shafarevich Groups using Iwasawa theory, preprint 2009.

INPUT:

- `p` - a prime > 3
- `prec` (optional) - the precision used in the computation of the p-adic L-Series
- `use_twists` (default = True) - If true the algorithm may change use a quadratic twist with minimal conductor to do the modular symbol computations rather than using the modular symbols of the curve itself. If False it forces the computation using the modular symbols of the curve itself.

OUTPUT: p-adic number - that conjecturally equals $Sha(E)(p)$.

If prec is set to zero (default) then the precision is set so that at least the first p-adic digit of conjectural $Sha(E)(p)$ is determined.

EXAMPLES: Good ordinary examples:

```
sage: EllipticCurve('11a1').sha().an_padic(5)  #rank 0
1 + O(5^2)
sage: EllipticCurve('43a1').sha().an_padic(5)  #rank 1
1 + O(5)
sage: EllipticCurve('389a1').sha().an_padic(5,4) #rank 2   (long time)
1 + O(5^3)
sage: EllipticCurve('858k2').sha().an_padic(7)  #rank 0, non trivial sha  (long time)
7^2 + O(7^3)
sage: EllipticCurve('300b2').sha().an_padic(3)  # an example with 9 elements in sha
3^2 + O(3^3)
sage: EllipticCurve('300b2').sha().an_padic(7)
2 + 7 + O(7^4)
```

Exceptional cases:

```
sage: EllipticCurve('11a1').sha().an_padic(11) #rank 0
1 + O(11)
```

The output has the correct sign

```
sage: EllipticCurve('123a1').sha().an_padic(41) #rank 1    (long time)
1 + O(41)
sage: EllipticCurve('817a1').sha().an_padic(43) #rank 2    (long time)
1 + O(43)
```

Supersingular cases:

```
sage: EllipticCurve('34a1').sha().an_padic(5) # rank 0     (long time)
1 + O(5^3)
sage: EllipticCurve('43a1').sha().an_padic(7) # rank 1    (very long time -- nearly a minute
1 + O(7)
sage: EllipticCurve('1483a1').sha().an_padic(5) # rank 2   (long time)
1 + O(5)
```

Cases that use a twist to a lower conductor

```
sage: EllipticCurve('99a1').sha().an_padic(5)
1 + O(5)
sage: EllipticCurve('240d3').sha().an_padic(5)  # sha has 4 elements here
4 + O(5)
sage: EllipticCurve('448c5').sha().an_padic(7,prec=4)  # long time
2 + 7 + O(7^3)
```

bound()

Compute a provably correct bound on the order of the Shafarevich-Tate group of this curve. The bound is a either False (no bound) or a list B of primes such that any divisor of Sha is in this list.

EXAMPLES:

```
sage: EllipticCurve('37a').sha().bound()
([2], 1)
```

bound_kato()

Returns a list p of primes such that the theorems of Kato's [Ka] and others (e.g., as explained in a paper/thesis of Grigor Grigorov [Gri]) imply that if p divides the order of Sha(E) then p is in the list.

If $L(E,1) = 0$, then this function gives no information, so it returns False.

THEOREM (Kato): Suppose $p \geq 5$ is a prime so the p-adic representation $\rho_{E,p}$ is surjective and $L(E,1) \neq 0$. Then $ord_p(\#Sha(E))$ divides $ord_p(L(E,1)/\Omega_E)$.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])    # 11A  = X_0(11)
sage: E.sha().bound_kato()
[2, 3, 5]
sage: E = EllipticCurve([0, -1, 1, 0, 0])        # X_1(11)
sage: E.sha().bound_kato()
[2, 3, 5]
sage: E = EllipticCurve([1,1,1,-352,-2689])      # 66B3
sage: E.sha().bound_kato()
[2, 3]
```

For the following curve one really has that 25 divides the order of Sha (by Grigorov-Stein paper [GS]):

```
sage: E = EllipticCurve([1, -1, 0, -332311, -73733731])    # 1058D1
sage: E.sha().bound_kato()                  # long time (about 1 second)
[2, 3, 5]
sage: E.non_surjective()                # long time (about 1 second)
[]
```

For this one, Sha is divisible by 7:

```
sage: E = EllipticCurve([0, 0, 0, -4062871, -3152083138])    # 3364C1
sage: E.sha().bound_kato()                  # long time (< 10 seconds)
[2, 3, 7]
```

No information about curves of rank > 0:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])        # 37A  (rank 1)
sage: E.sha().bound_kato()
False
```

REFERENCES:

- [Ka] Kayuza Kato, p-adic Hodge theory and values of zeta functions of modular forms, Cohomologies p-adiques et applications arithmetiques III, Asterisque vol 295, SMF, Paris, 2004.
- [Gri]
- [GS]

bound_kolyvagin(*D=0, regulator=None, ignore_nonsurj_hypothesis=False*)

Given a fundamental discriminant $D \neq -3, -4$ that satisfies the Heegner hypothesis for E, return a list of primes so that Kolyvagin's theorem (as in Gross's paper) implies that any prime divisor of Sha is in this list.

INPUT:

- `D` - (optional) a fundamental discriminant < -4 that satisfies the Heegner hypothesis for E; if not given, use the first such D
- `regulator` – (optional) regulator of $E(K)$; if not given, will be computed (which could take a long time)
- `ignore_nonsurj_hypothesis` (optional: default False) – If True, then gives the bound coming from Heegner point index, but without any hypothesis on surjectivity of the mod-p representation.

OUTPUT:

- list - a list of primes such that if p divides Sha(E/K), then p is in this list, unless E/K has complex multiplication or analytic rank greater than 2 (in which case we return 0).
- index - the odd part of the index of the Heegner point in the full group of K-rational points on E. (If E has CM, returns 0.)

REMARKS:

1. We do not have to assume that the Manin constant is 1 (or a power of 2). If the Manin constant were divisible by a prime, that prime would get included in the list of bad primes.
2. We assume the Gross-Zagier theorem is True under the hypothesis that $gcd(N, D) = 1$, instead of the stronger hypothesis $gcd(2 \cdot N, D) = 1$ that is in the original Gross-Zagier paper. That Gross-Zagier is true when $gcd(N, D) = 1$ is"well-known" to the experts, but doesn't seem to written up well in the literature.
3. Correctness of the computation is guaranteed using interval arithmetic, under the assumption that the regulator, square root, and period lattice are computed to precision at least 10^{-10}, i.e., they are correct up to addition or a real number with absolute value less than 10^{-10}.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.sha().bound_kolyvagin()
([2], 1)
sage: E = EllipticCurve('141a')
sage: E.sha().an()
1
sage: E.sha().bound_kolyvagin()
([2, 7], 49)
```

We get no information the curve has rank 2.:

```
sage: E = EllipticCurve('389a')
sage: E.sha().bound_kolyvagin()
(0, 0)
sage: E = EllipticCurve('681b')
sage: E.sha().an()
9
sage: E.sha().bound_kolyvagin()
([2, 3], 9)
```

p_primary_bound(*p*)

Returns a provable upper bound for the order of $Sha(E)(p)$. In particular, if this algorithm does not fail, then it proves that the p-primary part of Sha is finite.

INPUT: p – a prime > 3

OUTPUT: integer – power of p that bounds $Sha(E)(p)$ from above

The result is a proven upper bound on the order of $Sha(E)(p)$. So in particular it proves it finiteness even if the rank of the curve is larger than 1. Note also that this bound is sharp if one assumes the main conjecture of Iwasawa theory of elliptic curves (and this is known in certain cases).

EXAMPLES:

```
sage: e = EllipticCurve('11a3')
sage: e.sha().p_primary_bound(5)
0
sage: e.sha().p_primary_bound(7)
0
sage: e.sha().p_primary_bound(11)
0
sage: e.sha().p_primary_bound(13)
0

sage: e = EllipticCurve('389a1')
sage: e.sha().p_primary_bound(5)
0
sage: e.sha().p_primary_bound(7)
0
sage: e.sha().p_primary_bound(11)
0
sage: e.sha().p_primary_bound(13)
0

sage: e = EllipticCurve('858k2')
sage: e.sha().p_primary_bound(3)           # long time
0
sage: e.sha().p_primary_bound(7)           # long time
2
```

two_selmer_bound()

This returns a lower bound on the $\mathbb{F}_2$-dimension of the 2-torsion part of Sha, provided we can determine the rank of E. But it is not the best possible bound.

TO DO: This should be rewritten, to give the exact order of Sha[2], or if we can not find sufficently many points it should give a lower bound.

EXAMPLE:

```
sage: sh = EllipticCurve('571a1').sha()
sage: sh.two_selmer_bound()
2
sage: sh.an()
```

```
4

sage: sh = EllipticCurve('66a1').sha()
sage: sh.two_selmer_bound()
0
sage: sh.an()
1

sage: sh = EllipticCurve('960d1').sha()
sage: sh.two_selmer_bound()
0
sage: sh.an()
4
```

38.23 Miscellaneous p-adic functions

p-adic functions from ell_rational_field.py, moved here to reduce crowding in that file.

matrix_of_frobenius (*self, p, prec=20, check=False, check_hypotheses=True, algorithm='auto'*)
See the parameters and documentation for padic_E2.

padic_E2 (*self, p, prec=20, check=False, check_hypotheses=True, algorithm='auto'*)
Returns the value of the p-adic modular form $E2$ for (E, ω) where ω is the usual invariant differential $dx/(2y + a_1x + a_3)$.

INPUT:

- `p` - prime (= 5) for which E is good and ordinary
- `prec` - (relative) p-adic precision (= 1) for result
- `check` - boolean, whether to perform a consistency check. This will slow down the computation by a constant factor 2. (The consistency check is to compute the whole matrix of frobenius on Monsky-Washnitzer cohomology, and verify that its trace is correct to the specified precision. Otherwise, the trace is used to compute one column from the other one (possibly after a change of basis).)
- `check_hypotheses` - boolean, whether to check that this is a curve for which the p-adic sigma function makes sense
- `algorithm` - one of "standard", "sqrtp", or "auto". This selects which version of Kedlaya's algorithm is used. The "standard" one is the one described in Kedlaya's paper. The "sqrtp" one has better performance for large p, but only works when $p > 6N$ (N = prec). The "auto" option selects "sqrtp" whenever possible.

  Note that if the "sqrtp" algorithm is used, a consistency check will automatically be applied, regardless of the setting of the "check" flag.

OUTPUT: p-adic number to precision prec

Note: If the discriminant of the curve has nonzero valuation at p, then the result will not be returned mod p^{prec}, but it still *will* have prec *digits* of precision.

TODO: - Once we have a better implementation of the "standard" algorithm, the algorithm selection strategy for "auto" needs to be revisited.

AUTHORS:

- David Harvey (2006-09-01): partly based on code written by Robert Bradshaw at the MSRI 2006 modular forms workshop

ACKNOWLEDGMENT: - discussion with Eyal Goren that led to the trace trick.

EXAMPLES: Here is the example discussed in the paper "Computation of p-adic Heights and Log Convergence" (Mazur, Stein, Tate):

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^12 + 2*5^14 + 3*5^15
```

Let's try to higher precision (this is the same answer the MAGMA implementation gives):

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 100)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^12 + 2*5^14 + 3*5^15
```

Check it works at low precision too:

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 1)
2 + O(5)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 2)
2 + 4*5 + O(5^2)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 3)
2 + 4*5 + O(5^3)
```

TODO: With the old(-er), i.e., = sage-2.4 p-adics we got $5 + O(5^2)$ as output, i.e., relative precision 1, but with the newer p-adics we get relative precision 0 and absolute precision 1.

```
sage: EllipticCurve([1, 1, 1, 1, 1]).padic_E2(5, 1)
O(5)
```

Check it works for different models of the same curve (37a), even when the discriminant changes by a power of p (note that E2 depends on the differential too, which is why it gets scaled in some of the examples below):

```
sage: X1 = EllipticCurve([-1, 1/4])
sage: X1.j_invariant(), X1.discriminant()
 (110592/37, 37)
sage: X1.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)


sage: X2 = EllipticCurve([0, 0, 1, -1, 0])
sage: X2.j_invariant(), X2.discriminant()
 (110592/37, 37)
sage: X2.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)


sage: X3 = EllipticCurve([-1*(2**4), 1/4*(2**6)])
sage: X3.j_invariant(), X3.discriminant() / 2**12
 (110592/37, 37)
sage: 2**(-2) * X3.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)


sage: X4 = EllipticCurve([-1*(7**4), 1/4*(7**6)])
sage: X4.j_invariant(), X4.discriminant() / 7**12
 (110592/37, 37)
sage: 7**(-2) * X4.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)
```

```
sage: X5 = EllipticCurve([-1*(5**4), 1/4*(5**6)])
sage: X5.j_invariant(), X5.discriminant() / 5**12
 (110592/37, 37)
sage: 5**(-2) * X5.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)


sage: X6 = EllipticCurve([-1/(5**4), 1/4/(5**6)])
sage: X6.j_invariant(), X6.discriminant() * 5**12
 (110592/37, 37)
sage: 5**2 * X6.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)
```

Test check=True vs check=False:

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 1, check=False)
2 + O(5)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 1, check=True)
2 + O(5)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 30, check=False)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^12 + 2*5^14 + 3*5^15
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 30, check=True)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^12 + 2*5^14 + 3*5^15
```

Here's one using the $p^{1/2}$ algorithm:

```
sage: EllipticCurve([-1, 1/4]).padic_E2(3001, 3, algorithm="sqrtp")
1907 + 2819*3001 + 1124*3001^2 + O(3001^3)
```

padic_height (*self, p, prec=20, sigma=None, check_hypotheses=True*)

Computes the cyclotomic p-adic height.

The equation of the curve must be minimal at p.

INPUT:

- `p` - prime = 5 for which the curve has semi-stable reduction
- `prec` - integer = 1, desired precision of result
- `sigma` - precomputed value of sigma. If not supplied, this function will call padic_sigma to compute it.
- `check_hypotheses` - boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: A function that accepts two parameters:

- a Q-rational point on the curve whose height should be computed
- optional boolean flag 'check': if False, it skips some input checking, and returns the p-adic height of that point to the desired precision.
- The normalization (sign and a factor 1/2 with respect to some other normalizations that appear in the literature) is chosen in such a way as to make the p-adic Birch Swinnerton-Dyer conjecture hold as stated in [Mazur-Tate-Teitelbaum].

AUTHORS:

- Jennifer Balakrishnan: original code developed at the 2006 MSRI graduate workshop on modular forms
- David Harvey (2006-09-13): integrated into Sage, optimised to speed up repeated evaluations of the returned height function, addressed some thorny precision questions
- David Harvey (2006-09-30): rewrote to use division polynomials for computing denominator of nP.

•David Harvey (2007-02): cleaned up according to algorithms in "Efficient Computation of p-adic Heights"

•Chris Wuthrich (2007-05): added supersingular and multiplicative heights

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: P = E.gens()[0]
sage: h = E.padic_height(5, 10)
sage: h(P)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

An anomalous case:

```
sage: h = E.padic_height(53, 10)
sage: h(P)
26*53^-1 + 30 + 20*53 + 47*53^2 + 10*53^3 + 32*53^4 + 9*53^5 + 22*53^6 + 35*53^7 + 30*53^8 + 17*
```

Boundary case:

```
sage: E.padic_height(5, 3)(P)
5 + 5^2 + O(5^3)
```

A case that works the division polynomial code a little harder:

```
sage: E.padic_height(5, 10)(5*P)
5^3 + 5^4 + 5^5 + 3*5^8 + 4*5^9 + O(5^10)
```

Check that answers agree over a range of precisions:

```
sage: max_prec = 30    # make sure we get past p^2    # long time
sage: full = E.padic_height(5, max_prec)(P)           # long time
sage: for prec in range(1, max_prec):                 # long time
...       assert E.padic_height(5, prec)(P) == full   # long time
```

A supersingular prime for a curve:

```
sage: E = EllipticCurve('37a')
sage: E.is_supersingular(3)
True
sage: h = E.padic_height(3, 5)
sage: h(E.gens()[0])
(3 + 3^3 + O(3^6), 2*3^2 + 3^3 + 3^4 + 3^5 + 2*3^6 + O(3^7))
sage: E.padic_regulator(5)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + 5^10 + 3*5^11 + 3*5^12 + 5^13 + 4*5^14 + 5^15 + 2*5^16 + 5
sage: E.padic_regulator(3, 5)
(3 + 2*3^2 + 3^3 + O(3^4), 3^2 + 2*3^3 + 3^4 + O(3^5))
```

A torsion point in both the good and supersingular cases:

```
sage: E = EllipticCurve('11a')
sage: P = E.torsion_subgroup().gens()[0]; P
(5 : 5 : 1)
sage: h = E.padic_height(19, 5)
sage: h(P)
0
sage: h = E.padic_height(5, 5)
sage: h(P)
0
```

The result is not dependend on the model for the curve: sage: E = EllipticCurve([0,0,0,0,2^12*17]) sage: Em = E.minimal_model() sage: P = E.gens()[0] sage: Pm = Em.gens()[0] sage: h = E.padic_height(7) sage: hm = Em.padic_height(7) sage: h(P) == hm(Pm) True

padic_height_pairing_matrix(*self, p, prec=20, height=None, check_hypotheses=True*)

Computes the cyclotomic p-adic height pairing matrix of this curve with respect to the basis self.gens() for the Mordell-Weil group for a given odd prime p of good ordinary reduction.

INPUT:

- `p` - prime = 5
- `prec` - answer will be returned modulo p^{prec}
- `height` - precomputed height function. If not supplied, this function will call padic_height to compute it.
- `check_hypotheses` - boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: The p-adic cyclotomic height pairing matrix of this curve to the given precision.

TODO: - remove restriction that curve must be in minimal weierstrass form. This is currently required for E.gens().

AUTHORS:

- David Harvey, Liang Xiao, Robert Bradshaw, Jennifer Balakrishnan: original implementation at the 2006 MSRI graduate workshop on modular forms
- David Harvey (2006-09-13): cleaned up and integrated into Sage, removed some redundant height computations

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.padic_height_pairing_matrix(5, 10)
[5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)]
```

A rank two example:

```
sage: e =EllipticCurve('389a')
sage: e._set_gens([e(-1, 1), e(1,0)])  # avoid platform dependent gens
sage: e.padic_height_pairing_matrix(5,10)
[                      3*5 + 2*5^2 + 5^4 + 5^5 + 5^7 + 4*5^9 + O(5^10) 5 + 4*5^2 + 5^3 + 2*5^4 +
[5 + 4*5^2 + 5^3 + 2*5^4 + 3*5^5 + 4*5^6 + 5^7 + 5^8 + 2*5^9 + O(5^10)                           4
```

An anomalous rank 3 example:

```
sage: e = EllipticCurve("5077a")
sage: e._set_gens([e(-1,3), e(2,0), e(4,6)])
sage: e.padic_height_pairing_matrix(5,4)
[4 + 3*5 + 4*5^2 + 4*5^3 + O(5^4)       4 + 4*5^2 + 2*5^3 + O(5^4)       3*5 + 4*5^2 + 5^3 + O(5
[      4 + 4*5^2 + 2*5^3 + O(5^4)   3 + 4*5 + 3*5^2 + 5^3 + O(5^4)               2 + 4*5 + O(5
[      3*5 + 4*5^2 + 5^3 + O(5^4)                 2 + 4*5 + O(5^4)     1 + 3*5 + 5^2 + 5^3 + O(5
```

padic_height_via_multiply(*self, p, prec=20, E2=None, check_hypotheses=True*)

Computes the cyclotomic p-adic height.

The equation of the curve must be minimal at p.

INPUT:

- `p` - prime = 5 for which the curve has good ordinary reduction

- `prec` - integer = 2, desired precision of result
- `E2` - precomputed value of E2. If not supplied, this function will call padic_E2 to compute it. The value supplied must be correct mod $p^(prec - 2)$ (or slightly higher in the anomalous case; see the code for details).
- `check_hypotheses` - boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: A function that accepts two parameters:

- a Q-rational point on the curve whose height should be computed
- optional boolean flag 'check': if False, it skips some input checking, and returns the p-adic height of that point to the desired precision.
- The normalization (sign and a factor 1/2 with respect to some other normalizations that appear in the literature) is chosen in such a way as to make the p-adic Birch Swinnerton-Dyer conjecture hold as stated in [Mazur-Tate-Teitelbaum].

AUTHORS:

- David Harvey (2008-01): based on the padic_height() function, using the algorithm of"Computing p-adic heights via point multiplication"

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: P = E.gens()[0]
sage: h = E.padic_height_via_multiply(5, 10)
sage: h(P)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

An anomalous case:

```
sage: h = E.padic_height_via_multiply(53, 10)
sage: h(P)
26*53^-1 + 30 + 20*53 + 47*53^2 + 10*53^3 + 32*53^4 + 9*53^5 + 22*53^6 + 35*53^7 + 30*53^8 + 17*
```

Supply the value of E2 manually:

```
sage: E2 = E.padic_E2(5, 8)
sage: E2
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + O(5^8)
sage: h = E.padic_height_via_multiply(5, 10, E2=E2)
sage: h(P)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

Boundary case:

```
sage: E.padic_height_via_multiply(5, 3)(P)
5 + 5^2 + O(5^3)
```

Check that answers agree over a range of precisions:

```
sage: max_prec = 30    # make sure we get past p^2    # long time
sage: full = E.padic_height(5, max_prec)(P)           # long time
sage: for prec in range(2, max_prec):                 # long time
...       assert E.padic_height_via_multiply(5, prec)(P) == full   # long time
```

padic_lseries(*self, p, normalize='L_ratio', use_eclib=False*)

Return the p-adic L-series of self at p, which is an object whose approx method computes approximation to the true p-adic L-series to any desired precision.

INPUT:

- p - prime
- use_eclib - bool (default:False); whether or not to use John Cremona's eclib for the computation of modular symbols
- normalize - 'L_ratio' (default), 'period' or 'none'; this is describes the way the modular symbols are normalized. See modular_symbol for more details.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: L = E.padic_lseries(5); L
5-adic L-series of Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: type(L)
<class 'sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesOrdinary'>
```

We compute the 3-adic L-series of two curves of rank 0 and in each case verify the interpolation property for their leading coefficient (i.e., value at 0):

```
sage: e = EllipticCurve('11a')
sage: ms = e.modular_symbol()
sage: [ms(1/11), ms(1/3), ms(0), ms(oo)]
[0, -3/10, 1/5, 0]
sage: ms(0)
1/5
sage: L = e.padic_lseries(3)
sage: P = L.series(5)
sage: P(0)
2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + O(3^7)
sage: alpha = L.alpha(9); alpha
2 + 3^2 + 2*3^3 + 2*3^4 + 2*3^6 + 3^8 + O(3^9)
sage: R.<x> = QQ[]
sage: f = x^2 - e.ap(3)*x + 3
sage: f(alpha)
O(3^9)
sage: r = e.lseries().L_ratio(); r
1/5
sage: (1 - alpha^(-1))^2 * r
2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + 3^7 + O(3^9)
sage: P(0)
2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + O(3^7)
```

Next consider the curve 37b:

```
sage: e = EllipticCurve('37b')
sage: L = e.padic_lseries(3)
sage: P = L.series(5)
sage: alpha = L.alpha(9); alpha
1 + 2*3 + 3^2 + 2*3^5 + 2*3^7 + 3^8 + O(3^9)
sage: r = e.lseries().L_ratio(); r
1/3
sage: (1 - alpha^(-1))^2 * r
3 + 3^2 + 2*3^4 + 2*3^5 + 2*3^6 + 3^7 + O(3^9)
sage: P(0)
3 + 3^2 + 2*3^4 + 2*3^5 + O(3^6)
```

We can use eclib to compute the L-series:

```
sage: e = EllipticCurve('11a')
sage: L = e.padic_lseries(3,use_eclib=True)
sage: L.series(5,prec=10)
1 + 2*3^3 + 3^6 + O(3^7) + (2 + 2*3 + 3^2 + O(3^4))*T + (2 + 3 + 3^2 + 2*3^3 + O(3^4))*T^2 + (2*
```

padic_regulator(*self, p, prec=20, height=None, check_hypotheses=True*)

Computes the cyclotomic p-adic regulator of this curve.

INPUT:

- p - prime = 5
- prec - answer will be returned modulo p^{prec}
- height - precomputed height function. If not supplied, this function will call padic_height to compute it.
- check_hypotheses - boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: The p-adic cyclotomic regulator of this curve, to the requested precision.

If the rank is 0, we output 1.

TODO: - remove restriction that curve must be in minimal weierstrass form. This is currently required for E.gens().

AUTHORS:

- Liang Xiao: original implementation at the 2006 MSRI graduate workshop on modular forms
- David Harvey (2006-09-13): cleaned up and integrated into Sage, removed some redundant height computations
- Chris Wuthrich (2007-05-22): added multiplicative and supersingular cases
- David Harvey (2007-09-20): fixed some precision loss that was occurring

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.padic_regulator(5, 10)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

An anomalous case:

```
sage: E.padic_regulator(53, 10)
26*53^-1 + 30 + 20*53 + 47*53^2 + 10*53^3 + 32*53^4 + 9*53^5 + 22*53^6 + 35*53^7 + 30*53^8 + O(5
```

An anomalous case where the precision drops some:

```
sage: E = EllipticCurve("5077a")
sage: E.padic_regulator(5, 10)
5 + 5^2 + 4*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 4*5^7 + 2*5^8 + 5^9 + O(5^10)
```

Check that answers agree over a range of precisions:

```
sage: max_prec = 30    # make sure we get past p^2    # long time
sage: full = E.padic_regulator(5, max_prec)           # long time
sage: for prec in range(1, max_prec):                 # long time
...       assert E.padic_regulator(5, prec) == full   # long time
```

A case where the generator belongs to the formal group already (trac #3632):

```
sage: E = EllipticCurve([37,0])
sage: E.padic_regulator(5,10)
2*5^2 + 2*5^3 + 5^4 + 5^5 + 4*5^6 + 3*5^8 + 4*5^9 + O(5^10)
```

The result is not dependend on the model for the curve:

```
sage: E = EllipticCurve([0,0,0,0,2^12*17])
sage: Em = E.minimal_model()
sage: E.padic_regulator(7) == Em.padic_regulator(7)
True
```

padic_sigma(*self, p, N=20, E2=None, check=False, check_hypotheses=True*)

Computes the p-adic sigma function with respect to the standard invariant differential $dx/(2y + a_1x + a_3)$, as defined by Mazur and Tate, as a power series in the usual uniformiser t at the origin.

The equation of the curve must be minimal at p.

INPUT:

- `p` - prime = 5 for which the curve has good ordinary reduction
- `N` - integer = 1, indicates precision of result; see OUTPUT section for description
- `E2` - precomputed value of E2. If not supplied, this function will call padic_E2 to compute it. The value supplied must be correct mod p^{N-2}.
- `check` - boolean, whether to perform a consistency check (i.e. verify that the computed sigma satisfies the defining
- `differential equation` - note that this does NOT guarantee correctness of all the returned digits, but it comes pretty close :-))
- `check_hypotheses` - boolean, whether to check that this is a curve for which the p-adic sigma function makes sense

OUTPUT: A power series $t + \cdots$ with coefficients in $\mathbf{Z}_p$.

The output series will be truncated at $O(t^{N+1})$, and the coefficient of t^n for $n \geq 1$ will be correct to precision $O(p^{N-n+1})$.

In practice this means the following. If $t_0 = p^k u$, where u is a p-adic unit with at least N digits of precision, and $k \geq 1$, then the returned series may be used to compute $\sigma(t_0)$ correctly modulo p^{N+k} (i.e. with N correct p-adic digits).

ALGORITHM: Described in "Efficient Computation of p-adic Heights" (David Harvey), which is basically an optimised version of the algorithm from "p-adic Heights and Log Convergence" (Mazur, Stein, Tate).

Running time is soft-$O(N^2 \log p)$, plus whatever time is necessary to compute E_2.

AUTHORS:

- David Harvey (2006-09-12)
- David Harvey (2007-02): rewrote

EXAMPLES:

```
sage: EllipticCurve([-1, 1/4]).padic_sigma(5, 10)
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7 + O(5^8))*t^3 + O(5^
```

Run it with a consistency check:

```
sage: EllipticCurve("37a").padic_sigma(5, 10, check=True)
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7 + O(5^8))*t^3 + (3 +
```

Boundary cases:

```
sage: EllipticCurve([1, 1, 1, 1, 1]).padic_sigma(5, 1)
 (1 + O(5))*t + O(t^2)
sage: EllipticCurve([1, 1, 1, 1, 1]).padic_sigma(5, 2)
 (1 + O(5^2))*t + (3 + O(5))*t^2 + O(t^3)
```

Supply your very own value of E2:

```
sage: X = EllipticCurve("37a")
sage: my_E2 = X.padic_E2(5, 8)
sage: my_E2 = my_E2 + 5**5    # oops!!!
sage: X.padic_sigma(5, 10, E2=my_E2)
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 4*5^5 + 2*5^6 + 3*5^7 + O(5^8))*t^
```

Check that sigma is "weight 1".

```
sage: f = EllipticCurve([-1, 3]).padic_sigma(5, 10)
sage: g = EllipticCurve([-1*(2**4), 3*(2**6)]).padic_sigma(5, 10)
sage: t = f.parent().gen()
sage: f(2*t)/2
(1 + O(5^10))*t + (4 + 3*5 + 3*5^2 + 3*5^3 + 4*5^4 + 4*5^5 + 3*5^6 + 5^7 + O(5^8))*t^3 + (3 + 3*
sage: g
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (4 + 3*5 + 3*5^2 + 3*5^3 + 4*5^4 + 4*5^5 + 3*5^6 + 5^7
sage: f(2*t)/2 -g
O(t^11)
```

Test that it returns consistent results over a range of precision:

```
sage: max_N = 30   # get up to at least p^2          # long time
sage: E = EllipticCurve([1, 1, 1, 1, 1])             # long time
sage: p = 5                                          # long time
sage: E2 = E.padic_E2(5, max_N)                      # long time
sage: max_sigma = E.padic_sigma(p, max_N, E2=E2)     # long time
sage: for N in range(3, max_N):                      # long time
...       sigma = E.padic_sigma(p, N, E2=E2)         # long time
...       assert sigma == max_sigma
```

padic_sigma_truncated(*self, p, N=20, lamb=0, E2=None, check_hypotheses=True*)

Computes the p-adic sigma function with respect to the standard invariant differential $dx/(2y + a_1x + a_3)$, as defined by Mazur and Tate, as a power series in the usual uniformiser t at the origin.

The equation of the curve must be minimal at p.

This function differs from padic_sigma() in the precision profile of the returned power series; see OUTPUT below.

INPUT:

- `p` - prime = 5 for which the curve has good ordinary reduction
- `N` - integer = 2, indicates precision of result; see OUTPUT section for description
- `lamb` - integer = 0, see OUTPUT section for description
- `E2` - precomputed value of E2. If not supplied, this function will call padic_E2 to compute it. The value supplied must be correct mod p^{N-2}.
- `check_hypotheses` - boolean, whether to check that this is a curve for which the p-adic sigma function makes sense

OUTPUT: A power series $t + \cdots$ with coefficients in $\mathbf{Z}_p$.

The coefficient of t^j for $j \geq 1$ will be correct to precision $O(p^{N-2+(3-j)(lamb+1)})$.

ALGORITHM: Described in "Efficient Computation of p-adic Heights" (David Harvey, to appear in LMS JCM), which is basically an optimised version of the algorithm from "p-adic Heights and Log Convergence" (Mazur, Stein, Tate), and "Computing p-adic heights via point multiplication" (David Harvey, still draft form).

Running time is soft-$O(N^2\lambda^{-1}\log p)$, plus whatever time is necessary to compute E_2.

AUTHOR:

- David Harvey (2008-01): wrote based on previous padic_sigma function

EXAMPLES:

```
sage: E = EllipticCurve([-1, 1/4])
sage: E.padic_sigma_truncated(5, 10)
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7 + O(5^8))*t^3 + O(5^
```

Note the precision of the t^3 coefficient depends only on N, not on lamb:

```
sage: E.padic_sigma_truncated(5, 10, lamb=2)
O(5^17) + (1 + O(5^14))*t + O(5^11)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7 + O(5^8))*t^3 + O(5
```

Compare against plain padic_sigma() function over a dense range of N and lamb

```
sage: E = EllipticCurve([1, 2, 3, 4, 7])                              # long time
sage: E2 = E.padic_E2(5, 50)                                          # long time
sage: for N in range(2, 10):                                          # long time
...       for lamb in range(10):                                      # long time
...           correct = E.padic_sigma(5, N + 3*lamb, E2=E2)           # long time
...           compare = E.padic_sigma_truncated(5, N=N, lamb=lamb, E2=E2)    # long time
...           assert compare == correct                               # long time
```

CHAPTER

THIRTYNINE

HYPERELLIPTIC CURVES

39.1 Hyperelliptic curve constructor

HyperellipticCurve (*f, h=None, names=None, PP=None*)

Returns the hyperelliptic curve $y^2 + hy = f$, for univariate polynomials h and f. If h is not given, then it defaults to 0.

INPUT:

- `f` - univariate polynomial
- `h` - optional univariate polynomial

EXAMPLES: A curve with and without the h term:

```
sage: R.<x> = QQ[]
sage: HyperellipticCurve(x^5 + x + 1)
Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x + 1
sage: HyperellipticCurve(x^19 + x + 1, x-2)
Hyperelliptic Curve over Rational Field defined by y^2 + (x - 2)*y = x^19 + x + 1
```

A curve over a non-prime finite field:

```
sage: k.<a> = GF(9); R.<x> = k[]
sage: HyperellipticCurve(x^3 + x - 1, x+a)
Hyperelliptic Curve over Finite Field in a of size 3^2 defined by y^2 + (x + a)*y = x^3 + x + 2
```

Here's one where we change the names of the vars in the homogeneous polynomial:

```
sage: k.<a> = GF(9); R.<x> = k[]
sage: HyperellipticCurve(x^3 + x - 1, x+a, names=['X','Y'])
Hyperelliptic Curve over Finite Field in a of size 3^2 defined by Y^2 + (X + a)*Y = X^3 + X + 2
```

39.2 Hyperelliptic curves over a finite field

EXAMPLES:

```
sage: K.<a> = GF(9, 'a')
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^7 - x^5 - 2, x^2 + a)
sage: C._points_fast_sqrt()
[(0 : 1 : 0), (2*a : 2*a + 2 : 1), (2*a : 2*a : 1), (a + 1 : a : 1), (a + 1 : a + 1 : 1), (2 : a + 1
```

class HyperellipticCurve_finite_field(*PP, f, h=None, names=None, genus=None*)

frobenius_polynomial()

Charpoly of frobenius, as an element of ZZ[x].

TESTS:: sage: R.<t> = PolynomialRing(GF(37)) sage: H = HyperellipticCurve(t^5 + t + 2) sage: H.frobenius_polynomial() x^4 + x^3 - 52*x^2 + 37*x + 1369

A quadratic twist: sage: H = HyperellipticCurve(2*t^5 + 2*t + 4) sage: H.frobenius_polynomial() x^4 - x^3 - 52*x^2 - 37*x + 1369

points()

All the points on this hyperelliptic curve.

EXAMPLES:

```
sage: x = polygen(GF(7))
sage: C = HyperellipticCurve(x^7 - x^2 - 1)
sage: C.points()
[(0 : 1 : 0), (2 : 5 : 1), (2 : 2 : 1), (3 : 0 : 1), (4 : 6 : 1), (4 : 1 : 1), (5 : 0 : 1),

sage: x = polygen(GF(121, 'a'))
sage: C = HyperellipticCurve(x^5 + x - 1, x^2 + 2)
sage: len(C.points())
122
```

39.3 Hyperelliptic curves over a general ring

EXAMPLE:

```
sage: P.<x> = GF(5)[]
sage: f = x^5 - 3*x^4 - 2*x^3 + 6*x^2 + 3*x - 1
sage: C = HyperellipticCurve(f); C
Hyperelliptic Curve over Finite Field of size 5 defined by y^2 = x^5 + 2*x^4 + 3*x^3 + x^2 + 3*x + 4
```

EXAMPLE:

```
sage: P.<x> = QQ[]
sage: f = 4*x^5 - 30*x^3 + 45*x - 22
sage: C = HyperellipticCurve(f); C
Hyperelliptic Curve over Rational Field defined by y^2 = 4*x^5 - 30*x^3 + 45*x - 22
sage: C.genus()
2

sage: D = C.affine_patch(0)
sage: D.defining_polynomials()[0].parent()
Multivariate Polynomial Ring in x0, x1 over Rational Field
```

class HyperellipticCurve_generic(*PP, f, h=None, names=None, genus=None*)

change_ring(*R*)

genus()

has_odd_degree_model()

Return True if an odd degree model of self exists over the field of definition; False otherwise.

Use `odd_degree_model` to calculate an odd degree model.

EXAMPLES:: sage: x = QQ['x'].0 sage: HyperellipticCurve(x^5 + x).has_odd_degree_model() True sage: HyperellipticCurve(x^6 + x).has_odd_degree_model() True sage: HyperellipticCurve(x^6 + x + 1).has_odd_degree_model() False

hyperelliptic_polynomials(*K=None, var='x'*)

EXAMPLES:

```
sage: R.<x> = QQ[]; C = HyperellipticCurve(x^3 + x - 1, x^3/5); C
Hyperelliptic Curve over Rational Field defined by y^2 + 1/5*x^3*y = x^3 + x - 1
sage: C.hyperelliptic_polynomials()
(x^3 + x - 1, 1/5*x^3)
```

jacobian()

lift_x(*x, all=False*)

monsky_washnitzer_gens()

odd_degree_model()

Return an odd degree model of self, or raise ValueError if one does not exist over the field of definition.

EXAMPLES:

```
sage: x = QQ['x'].gen()
sage: H = HyperellipticCurve((x^2 + 2)*(x^2 + 3)*(x^2 + 5)); H
Hyperelliptic Curve over Rational Field defined by y^2 = x^6 + 10*x^4 + 31*x^2 + 30
sage: H.odd_degree_model()
...
ValueError: No odd degree model exists over field of definition

sage: K2 = QuadraticField(-2, 'a')
sage: Hp2 = H.change_ring(K2).odd_degree_model(); Hp2
Hyperelliptic Curve over Number Field in a with defining polynomial x^2 + 2 defined by y^2 =

sage: K3 = QuadraticField(-3, 'b')
sage: Hp3 = H.change_ring(QuadraticField(-3, 'b')).odd_degree_model(); Hp3
Hyperelliptic Curve over Number Field in b with defining polynomial x^2 + 3 defined by y^2 =

Of course, Hp2 and Hp3 are isomorphic over the composite
extension.  One consequence of this is that odd degree models
reduced over "different" fields should have the same number of
points on their reductions.  43 and 67 split completely in the
compositum, so when we reduce we find:

sage: P2 = K2.factor(43)[0][0]
sage: P3 = K3.factor(43)[0][0]
sage: Hp2.change_ring(K2.residue_field(P2)).frobenius_polynomial()
x^4 - 16*x^3 + 134*x^2 - 688*x + 1849
sage: Hp3.change_ring(K3.residue_field(P3)).frobenius_polynomial()
x^4 - 16*x^3 + 134*x^2 - 688*x + 1849
sage: H.change_ring(GF(43)).odd_degree_model().frobenius_polynomial()
x^4 - 16*x^3 + 134*x^2 - 688*x + 1849

sage: P2 = K2.factor(67)[0][0]
sage: P3 = K3.factor(67)[0][0]
sage: Hp2.change_ring(K2.residue_field(P2)).frobenius_polynomial()
x^4 - 8*x^3 + 150*x^2 - 536*x + 4489
sage: Hp3.change_ring(K3.residue_field(P3)).frobenius_polynomial()
x^4 - 8*x^3 + 150*x^2 - 536*x + 4489
sage: H.change_ring(GF(67)).odd_degree_model().frobenius_polynomial()
x^4 - 8*x^3 + 150*x^2 - 536*x + 4489
```

TESTS:: sage: HyperellipticCurve(x^5 + 1, 1).odd_degree_model() Traceback (most recent call last): ... NotImplementedError: odd_degree_model only implemented for curves in Weierstrass form sage: HyperellipticCurve(x^5 + 1, names="U, V").odd_degree_model() Hyperelliptic Curve over Rational Field defined by V^2 = U^5 + 1

is_HyperellipticCurve(*C*)
EXAMPLES:

```
sage: R.<x> = QQ[]; C = HyperellipticCurve(x^3 + x - 1); C
Hyperelliptic Curve over Rational Field defined by y^2 = x^3 + x - 1
sage: sage.schemes.hyperelliptic_curves.hyperelliptic_generic.is_HyperellipticCurve(C)
True
```

39.4 Constructor for Jacobian of a hyperelliptic curve

Jacobian(*C*)

39.5 Jacobian of a Hyperelliptic curve of Genus 2.

class **HyperellipticJacobian_g2**(*C*)

kummer_surface()

39.6 Jacobian of a General Hyperelliptic Curve

class **HyperellipticJacobian_generic**(*C*)
EXAMPLES:

```
sage: FF = FiniteField(2003)
sage: R.<x> = PolynomialRing(FF)
sage: f = x**5 + 1184*x**3 + 1846*x**2 + 956*x + 560
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: a = x**2 + 376*x + 245; b = 1015*x + 1368
sage: X = J(FF)
sage: D = X([a,b])
sage: D
(x^2 + 376*x + 245, y + 988*x + 635)
sage: J(0)
(1)
sage: D == J([a,b])
True
sage: D == D + J(0)
True
```

An more extended example, demonstrating arithmetic in J(QQ) and J(K) for a number field K/QQ.

```
sage: P.<x> = PolynomialRing(QQ)
sage: f = x^5 - x + 1; h = x
sage: C = HyperellipticCurve(f,h,'u,v')
sage: C
```

```
Hyperelliptic Curve over Rational Field defined by v^2 + u*v = u^5 - u + 1
sage: PP = C.ambient_space()
sage: PP
Projective Space of dimension 2 over Rational Field
sage: C.defining_polynomial()
-x0^5 + x0*x1*x2^3 + x1^2*x2^3 + x0*x2^4 - x2^5
sage: C(QQ)
Set of Rational Points of Hyperelliptic Curve over Rational Field defined by v^2 + u*v = u^5 - u
sage: K.<t> = NumberField(x^2-2)
sage: C(K)
       Set of Rational Points of Closed subscheme of Projective Space of dimension 2 over Number
sage: P = C(QQ)(0,1,1); P
(0 : 1 : 1)
sage: P == C(0,1,1)
True
sage: C(0,1,1).parent()
Set of Rational Points of Hyperelliptic Curve over Rational Field defined by v^2 + u*v = u^5 - u
sage: P1 = C(K)(P)
sage: P2 = C(K)([2,4*t-1,1])
sage: P3 = C(K)([-1/2,1/8*(7*t+2),1])
sage: P1, P2, P3
((0 : 1 : 1), (2 : 4*t - 1 : 1), (-1/2 : 7/8*t + 1/4 : 1))
sage: J = C.jacobian()
sage: J
Jacobian of Hyperelliptic Curve over Rational Field defined by v^2 + u*v = u^5 - u + 1
sage: Q = J(QQ)(P); Q
(u, v - 1)
sage: for i in range(6): Q*i
(1)
(u, v - 1)
(u^2, v + u - 1)
(u^2, v + 1)
(u, v + 1)
(1)
sage: Q1 = J(K)(P1); print "%s -> %s"%( P1, Q1 )
(0 : 1 : 1) -> (u, v - 1)
sage: Q2 = J(K)(P2); print "%s -> %s"%( P2, Q2 )
(2 : 4*t - 1 : 1) -> (u - 2, v - 4*t + 1)
sage: Q3 = J(K)(P3); print "%s -> %s"%( P3, Q3 )
(-1/2 : 7/8*t + 1/4 : 1) -> (u + 1/2, v - 7/8*t - 1/4)
sage: R.<x> = PolynomialRing(K)
sage: Q4 = J(K)([x^2-t,R(1)])
sage: for i in range(4): Q4*i
(1)
(u^2 - t, v - 1)
(u^2 + (-3/4*t - 9/16)*u + 1/2*t + 1/4, v + (-1/32*t - 57/64)*u + 1/2*t + 9/16)
(u^2 + (1352416/247009*t - 1636930/247009)*u - 1156544/247009*t + 1900544/247009, v + (-23263454
sage: R2 = Q2*5; R2
(u^2 - 3789465233/116983808*u - 267915823/58491904, v + (-233827256513849/1789384327168*t + 1/2)
sage: R3 = Q3*5; R3
(u^2 + 566330080839913890623/144264547989509096459 52*u - 265318141763956762312 73/28852909597901
sage: R4 = Q4*5; R4
(u^2 - 3789465233/116983808*u - 267915823/58491904, v + (233827256513849/1789384327168*t + 1/2)*
sage: # Thus we find the following identity:
sage: 5*Q2 + 5*Q4
(1)
sage: # Moreover the following relation holds in the 5-torsion subgroup:
sage: Q2 + Q4 == 2*Q1
```

```
True
```

dimension()

Return the dimension of this Jacobian.

OUTPUT: Integer

EXAMPLES:

```
sage: k.<a> = GF(9); R.<x> = k[]
sage: HyperellipticCurve(x^3 + x - 1, x+a).jacobian().dimension()
1
sage: g = HyperellipticCurve(x^6 + x - 1, x+a).jacobian().dimension(); g
2
sage: type(g)
<type 'sage.rings.integer.Integer'>
```

point(*mumford, check=True*)

39.7 Rational point sets on a Jacobian

EXAMPLES:

```
sage: x = QQ['x'].0
sage: f = x^5 + x + 1
sage: C = HyperellipticCurve(f); C
Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x + 1
sage: C(QQ)
Set of Rational Points of Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x + 1
sage: P = C([0,1,1])
sage: J = C.jacobian(); J
Jacobian of Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x + 1
sage: Q = J(QQ)(P); Q
(x, y - 1)
sage: Q + Q
(x^2, y - 1/2*x - 1)
sage: Q*3
(x^2 - 1/64*x + 1/8, y + 255/512*x + 65/64)
```

class JacobianHomset_divisor_classes(*X, S*)

base_extend(*R*)

curve()

value_ring()

Returns S for a homset X(T) where T = Spec(S).

39.8 Jacobian 'morphism' as a class in the Picard group.

This module implements the group operation in the Picard group of a hyperelliptic curve, represented as divisors in Mumford representation, using Cantor's algorithm.

A divisor on the hyperelliptic curve $y^2+yh(x) = f(x)$ is stored in Mumford representation, that is, as two polynomials $u(x)$ and $v(x)$ such that:

- $u(x)$ is monic,
- $u(x)$ divides $f(x) - h(x)v(x) - v(x)^2$,
- $deg(v(x)) < deg(u(x)) \leq g$.

REFERENCES:

A readable introduction to divisors, the Picard group, Mumford representation, and Cantor's algorithm:

- J. Scholten, F. Vercauteren. An Introduction to Elliptic and Hyperelliptic Curve Cryptography and the NTRU Cryptosystem. To appear in B. Preneel (Ed.) State of the Art in Applied Cryptography - COSIC '03, Lecture Notes in Computer Science, Springer 2004.

A standard reference in the field of cryptography:

- R. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren, Handbook of Elliptic and Hyperelliptic Curve Cryptography. CRC Press, 2005.

EXAMPLES: The following curve is the reduction of a curve whose Jacobian has complex multiplication.

```
sage: x = GF(37)['x'].gen()
sage: H = HyperellipticCurve(x^5 + 12*x^4 + 13*x^3 + 15*x^2 + 33*x); H
Hyperelliptic Curve over Finite Field of size 37 defined by y^2 = x^5 + 12*x^4 + 13*x^3 + 15*x^2 + 33
```

At this time, Jacobians of hyperelliptic curves are handled differently than elliptic curves:

```
sage: J = H.jacobian(); J
Jacobian of Hyperelliptic Curve over Finite Field of size 37 defined by y^2 = x^5 + 12*x^4 + 13*x^3 +
sage: J = J(J.base_ring()); J
Set of points of Jacobian of Hyperelliptic Curve over Finite Field of size 37 defined by y^2 = x^5 +
```

Points on the Jacobian are represented by Mumford's polynomials. First we find a couple of points on the curve:

```
sage: P1 = H.lift_x(2); P1
(2 : 11 : 1)
sage: Q1 = H.lift_x(10); Q1
(10 : 18 : 1)
```

Observe that 2 and 10 are the roots of the polynomials in x, respectively:

```
sage: P = J(P1); P
(x + 35, y + 26)
sage: Q = J(Q1); Q
(x + 27, y + 19)


sage: P + Q
(x^2 + 25*x + 20, y + 13*x)
sage: (x^2 + 25*x + 20).roots(multiplicities=False)
[10, 2]
```

Frobenius satisfies

$$x^4 + 12 * x^3 + 78 * x^2 + 444 * x + 1369$$

on the Jacobian of this reduction and the order of the Jacobian is $N = 1904$.

```
sage: 1904*P
(1)
sage: 34*P == 0
True
sage: 35*P == P
True
sage: 33*P == -P
True


sage: Q*1904
(1)
sage: Q*238 == 0
True
sage: Q*239 == Q
True
sage: Q*237 == -Q
True
```

class JacobianMorphism_divisor_class_field(*parent, polys, check=True*)

An element of a Jacobian defined over a field, i.e. in $J(K) = \mathrm{Pic}^0_K(C)$.

scheme()

Return the scheme this morphism maps to; or, where this divisor lives.

> **Warning:** Although a pointset is defined over a specific field, the scheme returned may be over a different (usually smaller) field. The example below demonstrates this: the pointset is determined over a number field of absolute degree 2 but the scheme returned is defined over the rationals.

EXAMPLES:

```
sage: x = QQ['x'].gen()
sage: f = x^5 + x
sage: H = HyperellipticCurve(f)
sage: F.<a> = NumberField(x^2 - 2, 'a')
sage: J = H.jacobian()(F); J
Set of points of Jacobian of Hyperelliptic Curve over Rational Field defined by y^2 = x^5 +

sage: P = J(H.lift_x(F(1)))
sage: P.scheme()
Jacobian of Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x
```

cantor_composition(*D1, D2, f, h, genus*)

EXAMPLES:

```
sage: F.<a> = GF(7^2, 'a')
sage: x = F['x'].gen()
sage: f = x^7 + x^2 + a
sage: H = HyperellipticCurve(f, 2*x); H
Hyperelliptic Curve over Finite Field in a of size 7^2 defined by y^2 + 2*x*y = x^7 + x^2 + a
sage: J = H.jacobian()(F); J
Set of points of Jacobian of Hyperelliptic Curve over Finite Field in a of size 7^2 defined by y

sage: Q = J(H.lift_x(F(1))); Q
(x + 6, y + 2*a + 2)
sage: 10*Q # indirect doctest
(x^3 + (3*a + 1)*x^2 + (2*a + 5)*x + a + 5, y + (4*a + 5)*x^2 + (a + 1)*x + 6*a + 3)
```

```
sage: 7*8297*Q
(1)

sage: Q = J(H.lift_x(F(a+1))); Q
(x + 6*a + 6, y + 2)
sage: 7*8297*Q # indirect doctest
(1)
```

cantor_composition_simple (*D1, D2, f, genus*)

Given D_1 and D_2 two reduced Mumford divisors on the Jacobian of the curve $y^2 = f(x)$, computes a representative $D_1 + D_2$.

Warning: The representative computed is NOT reduced! Use `cantor_reduction_simple()` to reduce it.

EXAMPLES:

```
sage: x = QQ['x'].gen()
sage: f = x^5 + x
sage: H = HyperellipticCurve(f); H
Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x

sage: F.<a> = NumberField(x^2 - 2, 'a')
sage: J = H.jacobian()(F); J
Set of points of Jacobian of Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x de

sage: P = J(H.lift_x(F(1))); P
(x - 1, y - a)
sage: Q = J(H.lift_x(F(0))); Q
(x, y)
sage: 2*P + 2*Q # indirect doctest
(x^2 - 2*x + 1, y - 3/2*a*x + 1/2*a)
sage: 2*(P + Q) # indirect doctest
(x^2 - 2*x + 1, y - 3/2*a*x + 1/2*a)
sage: 3*P # indirect doctest
(x^2 - 25/32*x + 49/32, y - 45/256*a*x - 315/256*a)
```

cantor_reduction (*a, b, f, h, genus*)

Return the unique reduced divisor linearly equivalent to (a, b) on the curve $y^2 + yh(x) = f(x)$.

See the docstring of `sage.schemes.hyperelliptic_curves.jacobian_morphism` for information about divisors, linear equivalence, and reduction.

EXAMPLES:

```
sage: x = QQ['x'].gen()
sage: f = x^5 - x
sage: H = HyperellipticCurve(f, x); H
Hyperelliptic Curve over Rational Field defined by y^2 + x*y = x^5 - x
sage: J = H.jacobian()(QQ); J
Set of points of Jacobian of Hyperelliptic Curve over Rational Field defined by y^2 + x*y = x^5
```

The following point is 2-torsion:

```
sage: Q = J(H.lift_x(0)); Q
(x, y)
sage: 2*Q # indirect doctest
(1)
```

The next point is not 2-torsion:

```
sage: P = J(H.lift_x(-1)); P
(x + 1, y - 1)
sage: 2 * J(H.lift_x(-1)) # indirect doctest
(x^2 + 2*x + 1, y - 3*x - 4)
sage: 3 * J(H.lift_x(-1)) # indirect doctest
(x^2 - 487*x - 324, y - 10754*x - 7146)
```

cantor_reduction_simple(*a, b, f, genus*)

Return the unique reduced divisor linearly equivalent to (a, b) on the curve $y^2 = f(x)$.

See the docstring of sage.schemes.hyperelliptic_curves.jacobian_morphism for information about divisors, linear equivalence, and reduction.

EXAMPLES:

```
sage: x = QQ['x'].gen()
sage: f = x^5 - x
sage: H = HyperellipticCurve(f); H
Hyperelliptic Curve over Rational Field defined by y^2 = x^5 - x
sage: J = H.jacobian()(QQ); J
Set of points of Jacobian of Hyperelliptic Curve over Rational Field defined by y^2 = x^5 - x de
```

The following point is 2-torsion:

```
sage: P = J(H.lift_x(-1)); P
(x + 1, y)
sage: 2 * P # indirect doctest
(1)
```

39.9 Conductor and Reduction Types for Genus 2 Curves

AUTHORS:

- Qing Liu and Henri Cohen (1994-1998): wrote genus2reduction C program
- William Stein (2006-03-05): wrote Sage interface to genus2reduction

ACKNOWLEDGMENT: (From Liu's website:) Many thanks to Henri Cohen who started writing this program. After this program is available, many people pointed out to me (mathematical as well as programming) bugs : B. Poonen, E. Schaefer, C. Stahlke, M. Stoll, F. Villegas. So thanks to all of them. Thanks also go to Ph. Depouilly who help me to compile the program.

Also Liu has given me explicit permission to include genus2reduction with Sage and for people to modify the C source code however they want.

class Genus2reduction()

Conductor and Reduction Types for Genus 2 Curves.

Use `R = genus2reduction(Q, P)` to obtain reduction information about the Jacobian of the projective smooth curve defined by $y^2 + Q(x)y = P(x)$. Type `R?` for further documentation and a description of how to interpret the local reduction data.

EXAMPLES:

```
sage: x = QQ['x'].0
sage: R = genus2reduction(x^3 - 2*x^2 - 2*x + 1, -5*x^5)
sage: R.conductor
1416875
sage: factor(R.conductor)
5^4 * 2267
```

This means that only the odd part of the conductor is known.

```
sage: R.prime_to_2_conductor_only
True
```

The discriminant is always minimal away from 2, but possibly not at 2.

```
sage: factor(R.minimal_disc)
2^3 * 5^5 * 2267
```

Printing R summarizes all the information computed about the curve

```
sage: R
Reduction data about this proper smooth genus 2 curve:
    y^2 + (x^3 - 2*x^2 - 2*x + 1)*y = -5*x^5
A Minimal Equation (away from 2):
    y^2 = x^6 - 240*x^4 - 2550*x^3 - 11400*x^2 - 24100*x - 19855
Minimal Discriminant (away from 2):  56675000
Conductor (away from 2): 1416875
Local Data:
    p=2
    (potential) stable reduction:  (II), j=1
    p=5
    (potential) stable reduction:  (I)
    reduction at p: [V] page 156, (3), f=4
    p=2267
    (potential) stable reduction:  (II), j=432
    reduction at p: [I{1-0-0}] page 170, (1), f=1
```

Here are some examples of curves with modular Jacobians:

```
sage: R = genus2reduction(x^3 + x + 1, -2*x^5 - 3*x^2 + 2*x - 2)
sage: factor(R.conductor)
23^2
sage: factor(genus2reduction(x^3 + 1, -x^5 - 3*x^4 + 2*x^2 + 2*x - 2).conductor)
29^2
sage: factor(genus2reduction(x^3 + x + 1, x^5 + 2*x^4 + 2*x^3 + x^2 - x - 1).conductor)
5^6
```

EXAMPLE:

```
sage: genus2reduction(0, x^6 + 3*x^3 + 63)
Reduction data about this proper smooth genus 2 curve:
        y^2 = x^6 + 3*x^3 + 63
A Minimal Equation (away from 2):
        y^2 = x^6 + 3*x^3 + 63
Minimal Discriminant (away from 2):  10628388316852992
```

```
Conductor (away from 2): 2893401
Local Data:
        p=2
        (potential) stable reduction:  (V), j1+j2=0, j1*j2=0
        p=3
        (potential) stable reduction:  (I)
        reduction at p: [III{9}] page 184, (3)^2, f=10
        p=7
        (potential) stable reduction:  (V), j1+j2=0, j1*j2=0
        reduction at p: [I{0}-II-0] page 159, (1), f=2
```

In the above example, Liu remarks that in fact at $p = 2$, the reduction is [II-II-0] page 163, (1), $f = 8$. So the conductor of J(C) is actually $2 \cdot 2893401 = 5786802$.

A MODULAR CURVE:

Consider the modular curve $X_1(13)$ defined by an equation

$$y^2 + (x^3 - x^2 - 1)y = x^2 - x.$$

We have:

```
sage: genus2reduction(x^3-x^2-1, x^2 - x)
Reduction data about this proper smooth genus 2 curve:
        y^2 + (x^3 - x^2 - 1)*y = x^2 - x
A Minimal Equation (away from 2):
        y^2 = x^6 + 58*x^5 + 1401*x^4 + 18038*x^3 + 130546*x^2 + 503516*x + 808561
Minimal Discriminant (away from 2):  169
Conductor: 169
Local Data:
        p=13
        (potential) stable reduction:  (V), j1+j2=0, j1*j2=0
        reduction at p: [I{0}-II-0] page 159, (1), f=2
```

So the curve has good reduction at 2. At $p = 13$, the stable reduction is union of two elliptic curves, and both of them have 0 as modular invariant. The reduction at 13 is of type [I_0-II-0] (see Namikawa-Ueno, page 159). It is an elliptic curve with a cusp. The group of connected components of the Neron model of $J(C)$ is trivial, and the exponent of the conductor of $J(C)$ at 13 is $f = 2$. The conductor of $J(C)$ is 13^2. (Note: It is a theorem of Conrad-Edixhoven-Stein that the component group of $J(X_1(p))$ is trivial for all primes p.)

console()

raw(*Q*, *P*)

Return the raw output of running the `genus2reduction` program on the hyperelliptic curve $y^2 + Q(x)y = P(x)$ as a string.

INPUT:

- `Q` - something coercible to a univariate polynomial over Q.
- `P` - something coercible to a univariate polynomial over Q.

OUTPUT:

- `string` - raw output
- `Q` - what Q was actually input to auxiliary genus2reduction program
- `P` - what P was actually input to auxiliary genus2reduction program

EXAMPLES:

```
sage: x = QQ['x'].0
sage: print genus2reduction.raw(x^3 - 2*x^2 - 2*x + 1, -5*x^5)[0]
a minimal equation over Z[1/2] is :
```

```
y^2 = x^6-240*x^4-2550*x^3-11400*x^2-24100*x-19855
<BLANKLINE>
factorization of the minimal (away from 2) discriminant :
[2,3;5,5;2267,1]
<BLANKLINE>
p=2
(potential) stable reduction :  (II), j=1
p=5
(potential) stable reduction :  (I)
reduction at p : [V] page 156, (3), f=4
p=2267
(potential) stable reduction :  (II), j=432
reduction at p : [I{1-0-0}] page 170, (1), f=1
<BLANKLINE>
the prime to 2 part of the conductor is 1416875
in factorized form : [2,0;5,4;2267,1]
```

Verify that we fix trac 5573:

```
sage: genus2reduction(x^3 + x^2 + x,-2*x^5 + 3*x^4 - x^3 - x^2 - 6*x - 2)
Reduction data about this proper smooth genus 2 curve:
y^2 + (x^3 + x^2 + x)*y = -2*x^5 + 3*x^4 - x^3 - x^2 - 6*x - 2
...
```

class Genus2reduction_expect (*server=None, server_tmpdir=None, logfile=None*)

class ReductionData (*raw, P, Q, minimal_equation, minimal_disc, local_data, conductor, prime_to_2_conductor_only*)

Reduction data for a genus 2 curve.

How to read `local_data` attribute, i.e., if this class is R, then the following is the meaning of `R.local_data[p]`.

For each prime number p dividing the discriminant of $y^2 + Q(x)y = P(x)$, there are two lines.

The first line contains information about the stable reduction after field extension. Here are the meanings of the symbols of stable reduction :

(I) The stable reduction is smooth (i.e. the curve has potentially good reduction).

(II) The stable reduction is an elliptic curve E with an ordinary double point. j mod p is the modular invariant of E.

(III) The stable reduction is a projective line with two ordinary double points.

(IV) The stable reduction is two projective lines crossing transversally at three points.

(V) The stable reduction is the union of two elliptic curves E_1 and E_2 intersecting transversally at one point. Let j_1, j_2 be their modular invariants, then $j_1 + j_2$ and $j_1 j_2$ are computed (they are numbers mod p).

(VI) The stable reduction is the union of an elliptic curve E and a projective line which has an ordinary double point. These two components intersect transversally at one point. j mod p is the modular invariant of E.

(VII) The stable reduction is as above, but the two components are both singular.

In the cases (I) and (V), the Jacobian $J(C)$ has potentially good reduction. In the cases (III), (IV) and (VII), $J(C)$ has potentially multiplicative reduction. In the two remaining cases, the (potential) semi-abelian reduction of $J(C)$ is extension of an elliptic curve (with modular invariant j mod p) by a torus.

The second line contains three data concerning the reduction at p without any field extension.

1. The first symbol describes the REDUCTION AT p of C. We use the symbols of Namikawa-Ueno for the type of the reduction (Namikawa, Ueno:"The complete classification of fibers in pencils of curves of genus two", Manuscripta Math., vol. 9, (1973), pages 143-186.) The reduction symbol is followed by the corresponding page number (or just an indiction) in the above article. The lower index is printed by ,

for instance, [I2-II-5] means [I_2-II-5]. Note that if K and K' are Kodaira symbols for singular fibers of elliptic curves, [K-K'-m] and [K'-K-m] are the same type. Finally, [K-K'-1] (not the same as [K-K'-1]) is [K'-K-alpha] in the notation of Namikawa-Ueno. The figure [2I_0-m] in Namikawa-Ueno, page 159 must be denoted by [2I_0-(m+1)].

2. The second datum is the GROUP OF CONNECTED COMPONENTS (over an ALGEBRAIC CLOSURE (!) of $\mathbf{F}_p$) of the Neron model of J(C). The symbol (n) means the cyclic group with n elements. When n=0, (0) is the trivial group (1). `Hn` is isomorphic to (2)x(2) if n is even and to (4) otherwise.

   Note - The set of rational points of Φ can be computed using Theorem 1.17 in S. Bosch and Q. Liu "Rational points of the group of components of a Neron model", Manuscripta Math. 98 (1999), 275-293.

3. Finally, f is the exponent of the conductor of $J(C)$ at p.

Warning: Be careful regarding the formula:

$$\text{valuation of the naive minimal discriminant} = f + n - 1 + 11c(X).$$

(Q. Liu : "Conducteur et discriminant minimal de courbes de genre 2", Compositio Math. 94 (1994) 51-79, Theoreme 2) is valid only if the residual field is algebraically closed as stated in the paper. So this equality does not hold in general over $\mathbf{Q}_p$. The fact is that the minimal discriminant may change after unramified extension. One can show however that, at worst, the change will stabilize after a quadratic unramified extension (Q. Liu : "Modeles entiers de courbes hyperelliptiques sur un corps de valuation discrete", Trans. AMS 348 (1996), 4577-4610, Section 7.2, Proposition 4).

genus2reduction_console()

CHAPTER

FORTY

CODING THEORY

40.1 Linear Codes

VERSION: 1.1

Let F be a finite field (we denote the finite field with q elements by $\mathbf{F}_q$). A subspace of F^n (with the standard basis) is called a linear code of length n. If its dimension is denoted k then we typically store a basis of C as a $k \times n$ matrix (the rows are the basis vectors) called the generator matrix of C. The rows of the parity check matrix of C are a basis for the code,

$$C^* = \{v \in GF(q)^n \mid v \cdot c = 0, \ for\ all\ c \in C\},$$

called the dual space of C.

If $F = \mathbf{F}_2$ then C is called a binary code. If $F = \mathbf{F}_q$ then C is called a q-ary code. The elements of a code C are called codewords.

The symmetric group S_n acts on F^n by permuting coordinates. If an element $p \in S_n$ sends a code C of length n to itself (in other words, every codeword of C is sent to some other codeword of C) then p is called a permutation automorphism of C. The (permutation) automorphism group is denoted $Aut(C)$.

This file contains

1. LinearCode class definition; LinearCodeFromVectorspace conversion function,
2. The spectrum (weight distribution), covering_radius, minimum distance programs (calling Steve Linton's or CJ Tjhal's C programs), characteristic_function, and several implementations of the Duursma zeta function (sd_zeta_polynomial, zeta_polynomial, zeta_function, chinen_polynomial, for example),
3. interface with best_known_linear_code_www (interface with codetables.de since A. Brouwer's online tables have been disabled), bounds_minimum_distance which call tables in GUAVA (updated May 2006) created by Cen Tjhai instead of the online internet tables,
4. gen_mat, list, check_mat, decode, dual_code, extended_code, shortened, punctured, genus, binomial_moment, and divisor methods for LinearCode,
5. Boolean-valued functions such as "==", is_self_dual, is_self_orthogonal, is_subcode, is_permutation_automorphism, is_permutation_equivalent (which interfaces with Robert Miller's partition refinement code),
6. permutation methods: automorphism_group_binary_code, is_permutation_automorphism, (permutation_automorphism_group is deprecated), permuted_code, standard_form, module_composition_factors,
7. design-theoretic methods: assmus_mattson_designs (implementing Assmus-Mattson Theorem),

8. code constructions, such as HammingCode and ToricCode, are in a separate `code_constructions.py` module; in the separate `guava.py` module, you will find constructions, such as RandomLinearCodeGuava and BinaryReedMullerCode, wrapped from the corresponding GUAVA codes.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G = MS([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0], [1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.basis()
[(1, 1, 1, 0, 0, 0, 0),
 (1, 0, 0, 1, 1, 0, 0),
 (0, 1, 0, 1, 0, 1, 0),
 (1, 1, 0, 1, 0, 0, 1)]
sage: c = C.basis()[1]
sage: c in C
True
sage: c.nonzero_positions()
[0, 3, 4]
sage: c.support()
[0, 3, 4]
sage: c.parent()
Vector space of dimension 7 over Finite Field of size 2
```

To be added:

1. More wrappers
2. GRS codes and special decoders.
3. P^1 Goppa codes and group actions on P^1 RR space codes.

REFERENCES:

- [HP] W. C. Huffman and V. Pless, Fundamentals of error-correcting codes, Cambridge Univ. Press, 2003.
- [Gu] GUAVA manual, http://www.gap-system.org/Packages/guava.html

AUTHORS:

- David Joyner (2005-11-22, 2006-12-03): initial version
- William Stein (2006-01-23): Inclusion in Sage
- David Joyner (2006-01-30, 2006-04): small fixes
- David Joyner (2006-07): added documentation, group-theoretical methods, ToricCode
- David Joyner (2006-08): hopeful latex fixes to documentation, added list and __iter__ methods to LinearCode and examples, added hamming_weight function, fixed random method to return a vector, TrivialCode, fixed subtle bug in dual_code, added galois_closure method, fixed mysterious bug in permutation_automorphism_group (GAP was over-using "G" somehow?)
- David Joyner (2006-08): hopeful latex fixes to documentation, added CyclicCode, best_known_linear_code, bounds_minimum_distance, assmus_mattson_designs (implementing Assmus-Mattson Theorem).
- David Joyner (2006-09): modified decode syntax, fixed bug in is_galois_closed, added LinearCode_from_vectorspace, extended_code, zeta_function

- Nick Alexander (2006-12-10): factor GUAVA code to guava.py
- David Joyner (2007-05): added methods punctured, shortened, divisor, characteristic_polynomial, binomial_moment, support for LinearCode. Completely rewritten zeta_function (old version is now zeta_function2) and a new function, LinearCodeFromVectorSpace.
- David Joyner (2007-11): added zeta_polynomial, weight_enumerator, chinen_polynomial; improved best_known_code; made some pythonic revisions; added is_equivalent (for binary codes)
- David Joyner (2008-01): fixed bug in decode reported by Harald Schilly, (with Mike Hansen) added some doctests.
- David Joyner (2008-02): translated standard_form, dual_code to Python.
- David Joyner (2008-03): translated punctured, shortened, extended_code, random (and renamed random to random_element), deleted zeta_function2, zeta_function3, added wrapper automorphism_group_binary_code to Robert Miller's code), added direct_sum_code, is_subcode, is_self_dual, is_self_orthogonal, redundancy_matrix, did some alphabetical reorganizing to make the file more readable. Fixed a bug in permutation_automorphism_group which caused it to crash.
- David Joyner (2008-03): fixed bugs in spectrum and zeta_polynomial (which misbehaved over non-prime base rings.
- David Joyner (2008-10): use CJ Tjhal's MinimumWeight if char = 2 or 3 for min_dist; add is_permutation_equivalent and improve permutation_automorphism_group using an interface with Robert Miller's code; added interface with Leon's code for the spectrum method.
- David Joyner (2009-02): added native decoding methods (see module_decoder.py)

TESTS:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G  = MS([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0], [1,1,0,1,0,0,1]])
sage: C  = LinearCode(G)
sage: C == loads(dumps(C))
True
```

class **LinearCode**(*gen_mat*)

A class for linear codes over a finite field or finite ring. Each instance is a linear code determined by a generator matrix G (i.e., a k x n matrix of (full) rank k, $k \leq n$ over a finite field F.

INPUT:

- G - a generator matrix over F. (G can be defined over a finite ring but the matrices over that ring must have certain attributes, such as"rank".)

OUTPUT: The linear code of length n over F having G as a generator matrix.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G  = MS([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0], [1,1,0,1,0,0,1]])
sage: C  = LinearCode(G)
sage: C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.base_ring()
Finite Field of size 2
sage: C.dimension()
4
sage: C.length()
```

```
7
sage: C.minimum_distance()
3
sage: C.spectrum()
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C.weight_distribution()
[1, 0, 0, 7, 7, 0, 0, 1]
sage: MS = MatrixSpace(GF(5),4,7)
sage: G  = MS([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0], [1,1,0,1,0,0,1]])
sage: C  = LinearCode(G)
sage: C
Linear code of length 7, dimension 4 over Finite Field of size 5
```

AUTHORS:

- David Joyner (11-2005)

ambient_space()

assmus_mattson_designs(*t, mode=None*)

Assmus and Mattson Theorem (section 8.4, page 303 of [HP]): Let $A_0, A_1, ..., A_n$ be the weights of the codewords in a binary linear $[n, k, d]$ code C, and let $A_0^*, A_1^*, ..., A_n^*$ be the weights of the codewords in its dual $[n, n-k, d^*]$ code C^*. Fix a t, $0 < t < d$, and let

$$s = |\{i \mid A_i^* \neq 0, 0 < i \leq n - t\}|.$$

Assume $s \leq d - t$.

1. If $A_i \neq 0$ and $d \leq i \leq n$ then $C_i = \{c \in C \mid wt(c) = i\}$ holds a simple t-design.
2. If $A_i^* \neq 0$ and $d* \leq i \leq n - t$ then $C_i^* = \{c \in C^* \mid wt(c) = i\}$ holds a simple t-design.

A block design is a pair (X, B), where X is a non-empty finite set of $v > 0$ elements called points, and B is a non-empty finite multiset of size b whose elements are called blocks, such that each block is a non-empty finite multiset of k points. A design without repeated blocks is called a simple block design. If every subset of points of size t is contained in exactly λ blocks the block design is called a $t - (v, k, \lambda)$ design (or simply a t-design when the parameters are not specified). When $\lambda = 1$ then the block design is called a $S(t, k, v)$ Steiner system.

In the Assmus and Mattson Theorem (1), X is the set $\{1, 2, ..., n\}$ of coordinate locations and $B = \{supp(c) \mid c \in C_i\}$ is the set of supports of the codewords of C of weight i. Therefore, the parameters of the t-design for C_i are

```
t =       given
v =       n
k =       i   (k not to be confused with dim(C))
b =       Ai
lambda = b*binomial(k,t)/binomial(v,t)  (by Theorem 8.1.6,
                                          p 294, in [HP])
```

Setting the mode="verbose" option prints out the values of the parameters.

The first example below means that the binary [24,12,8]-code C has the property that the (support of the) codewords of weight 8 (resp, 12, 16) form a 5-design. Similarly for its dual code C^* (of course $C = C^*$ in this case, so this info is extraneous). The test fails to produce 6-designs (ie, the hypotheses of the theorem fail to hold, not that the 6-designs definitely don't exist). The command assmus_mattson_designs(C,5,mode="verbose") returns the same value but prints out more detailed information.

The second example below illustrates the blocks of the 5-(24, 8, 1) design (ie, the S(5,8,24) Steiner system).

EXAMPLES:

```
sage: C = ExtendedBinaryGolayCode()              #  example 1
sage: C.assmus_mattson_designs(5)
['weights from C: ',
[8, 12, 16, 24],
'designs from C: ',
[[5, (24, 8, 1)], [5, (24, 12, 48)], [5, (24, 16, 78)], [5, (24, 24, 1)]],
'weights from C*: ',
[8, 12, 16],
'designs from C*: ',
[[5, (24, 8, 1)], [5, (24, 12, 48)], [5, (24, 16, 78)]]]
sage: C.assmus_mattson_designs(6)
0
sage: X = range(24)                              #  example 2
sage: blocks = [c.support() for c in C if hamming_weight(c)==8]; len(blocks)  # long time co
759
```

REFERENCE:

•[HP] W. C. Huffman and V. Pless, Fundamentals of ECC, Cambridge Univ. Press, 2003.

automorphism_group_binary_code()

This only applies to linear binary codes and returns its (permutation) automorphism group. In other words, if the code C has length n then it returns the subgroup of the symmetric group S_n:

$$\{g \in S_n \mid g(c) \in C, \forall c \in C\},$$

where S_n acts on $GF(2)^n$ by permuting coordinates.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: G = C.automorphism_group_binary_code(); G
Permutation Group with generators [(3,4)(5,6), (3,5)(4,6), (2,3)(5,7), (1,2)(5,6)]
sage: G.order()
168
```

basis()

binomial_moment(i)

Returns the i-th binomial moment of the $[n, k, d]_q$-code C:

$$B_i(C) = \sum_{S,|S|=i} \frac{q^{k_S} - 1}{q - 1}$$

where k_S is the dimension of the shortened code C_{J-S}, $J = [1, 2, ..., n]$. (The normalized binomial moment is $b_i(C) = \binom{(}{n}, d + i)^{-1} B_{d+i}(C)$.) In other words, C_{J-S} is the isomorphic to the subcode of C of codewords supported on S.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: C.binomial_moment(2)
0
sage: C.binomial_moment(3)     # long time
0
sage: C.binomial_moment(4)     # long time
35
```

,. warning:

```
This is slow.
```

REFERENCE:

•I. Duursma, "Combinatorics of the two-variable zeta function", Finite fields and applications, 109-136, Lecture Notes in Comput. Sci., 2948, Springer, Berlin, 2004.

characteristic()

characteristic_polynomial()

Returns the characteristic polynomial of a linear code, as defined in van Lint's text [vL].

EXAMPLES:

```
sage: C = ExtendedBinaryGolayCode()
sage: C.characteristic_polynomial()
-4/3*x^3 + 64*x^2 - 2816/3*x + 4096
```

REFERENCES:

•van Lint, Introduction to coding theory, 3rd ed., Springer-Verlag GTM, 86, 1999.

check_mat()

Returns the check matrix of self.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: Cperp = C.dual_code()
sage: C; Cperp
Linear code of length 7, dimension 4 over Finite Field of size 2
Linear code of length 7, dimension 3 over Finite Field of size 2
sage: C.gen_mat()
[1 0 0 1 0 1 0]
[0 1 0 1 0 1 1]
[0 0 1 1 0 0 1]
[0 0 0 0 1 1 1]
sage: C.check_mat()
[1 0 0 1 1 0 1]
[0 1 0 1 0 1 1]
[0 0 1 1 1 1 0]
sage: Cperp.check_mat()
[1 0 0 1 0 1 0]
[0 1 0 1 0 1 1]
[0 0 1 1 0 0 1]
[0 0 0 0 1 1 1]
sage: Cperp.gen_mat()
[1 0 0 1 1 0 1]
[0 1 0 1 0 1 1]
[0 0 1 1 1 1 0]
```

chinen_polynomial()

Returns the Chinen zeta polynomial of the code.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: C.chinen_polynomial()       # long time
1/5*(2*sqrt(2)*t^3 + 2*sqrt(2)*t^2 + 2*t^2 + sqrt(2)*t + 2*t + 1)/(sqrt(2) + 1)
sage: C = TernaryGolayCode()
sage: C.chinen_polynomial()       # long time
1/7*(3*sqrt(3)*t^3 + 3*sqrt(3)*t^2 + 3*t^2 + sqrt(3)*t + 3*t + 1)/(sqrt(3) + 1)
```

This last output agrees with the corresponding example given in Chinen's paper below.

REFERENCES:

•Chinen, K. "An abundance of invariant polynomials satisfying the Riemann hypothesis", April 2007 preprint.

covering_radius()

Wraps Guava's CoveringRadius command.

The covering radius of a linear code C is the smallest number r with the property that each element v of the ambient vector space of C has at most a distance r to the code C. So for each vector v there must be an element c of C with $d(v, c) \leq r$. A binary linear code with reasonable small covering radius is often referred to as a covering code.

For example, if C is a perfect code, the covering radius is equal to t, the number of errors the code can correct, where d = 2t+1, with d the minimum distance of C

EXAMPLES:

```
sage: C = HammingCode(5,GF(2))
sage: C.covering_radius()
1
```

decode(*right, method='syndrome'*)

Decodes a received vector v (=right) to an element c in the code C (=self). Optional methods are "guava", "nearest neighbor" or "syndrome". The method="guava" wraps GUAVA's Decodeword (Hamming codes have a special decoding algorithm; otherwise, syndrome decoding is used). The default is "syndrome".

INPUT:

- v - must be a vector of length = length(C)

OUTPUT: The codeword c in C closest to v.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: MS = MatrixSpace(GF(2),1,7)
sage: F = GF(2); a = F.gen()
sage: v1 = [a,a,F(0),a,a,F(0),a]
sage: C.decode(v1)
(1, 0, 0, 1, 1, 0, 1)
sage: C.decode(v1,method="nearest neighbor")
(1, 0, 0, 1, 1, 0, 1)
sage: C.decode(v1,method="guava")
(1, 0, 0, 1, 1, 0, 1)
sage: v2 = matrix([[a,a,F(0),a,a,F(0),a]])
sage: C.decode(v2)
(1, 0, 0, 1, 1, 0, 1)
sage: v3 = vector([a,a,F(0),a,a,F(0),a])
sage: c = C.decode(v3); c
(1, 0, 0, 1, 1, 0, 1)
sage: c in C
True
sage: C = HammingCode(2,GF(5))
sage: v = vector(GF(5),[1,0,0,2,1,0])
sage: C.decode(v)
(2, 0, 0, 2, 1, 0)
sage: F = GF(4,"a")
sage: C = HammingCode(2,F)
sage: v = vector(F, [1,0,0,a,1])
sage: C.decode(v)
(1, 0, 0, 1, 1)
sage: C.decode(v, method="nearest neighbor")
(1, 0, 0, 1, 1)
sage: C.decode(v, method="guava")
(1, 0, 0, 1, 1)
```

Does not work for very long codes since the syndrome table grows too large.

dimension()

direct_sum(*other*)

C1, C2 must be linear codes defined over the same base ring. Returns the (usual vector space) direct sum of the codes.

EXAMPLES:

```
sage: C1 = HammingCode(3,GF(2))
sage: C2 = C1.direct_sum(C1); C2
Linear code of length 14, dimension 8 over Finite Field of size 2
sage: C3 = C1.direct_sum(C2); C3
Linear code of length 21, dimension 12 over Finite Field of size 2
```

divisor()

Returns the divisor of a code (the divisor is the smallest integer $d_0 > 0$ such that each $A_i > 0$ iff i is divisible by d_0).

EXAMPLES:

```
sage: C = ExtendedBinaryGolayCode()
sage: C.divisor()   # Type II self-dual
4
sage: C = QuadraticResidueCodeEvenPair(17,GF(2))[0]
sage: C.divisor()
2
```

dual_code()

This computes the dual code Cd of the code C,

$$Cd = \{v \in V \mid v \cdot c = 0, \ \forall c \in C\}.$$

Does not call GAP.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: C.dual_code()
Linear code of length 7, dimension 3 over Finite Field of size 2
sage: C = HammingCode(3,GF(4,'a'))
sage: C.dual_code()
Linear code of length 21, dimension 3 over Finite Field in a of size 2^2
```

extended_code()

If self is a linear code of length n defined over F then this returns the code of length n+1 where the last digit c_n satisfies the check condition $c_0 + ... + c_n = 0$. If self is an $[n, k, d]$ binary code then the extended code $C^\vee$ is an $[n + 1, k, d^\vee]$ code, where $d^=d$ (if d is even) and $d^\vee = d + 1$ (if d is odd).

EXAMPLES:

```
sage: C = HammingCode(3,GF(4,'a'))
sage: C
Linear code of length 21, dimension 18 over Finite Field in a of size 2^2
sage: Cx = C.extended_code()
sage: Cx
Linear code of length 22, dimension 18 over Finite Field in a of size 2^2
```

galois_closure(*F0*)

If self is a linear code defined over F and F_0 is a subfield with Galois group $G = Gal(F/F_0)$ then this returns the G-module C^- containing C.

EXAMPLES:

```
sage: C = HammingCode(3,GF(4,'a'))
sage: Cc = C.galois_closure(GF(2))
sage: C; Cc
```

```
Linear code of length 21, dimension 18 over Finite Field in a of size 2^2
Linear code of length 21, dimension 20 over Finite Field in a of size 2^2
sage: c = C.basis()[1]
sage: V = VectorSpace(GF(4,'a'),21)
sage: c2 = V([x^2 for x in c.list()])
sage: c2 in C
False
sage: c2 in Cc
True
```

gen_mat()

Returns the generator matrix of the code.

EXAMPLES:

```
sage: C1 = HammingCode(3,GF(2))
sage: C1.gen_mat()
[1 0 0 1 0 1 0]
[0 1 0 1 0 1 1]
[0 0 1 1 0 0 1]
[0 0 0 0 1 1 1]
sage: C2 = HammingCode(2,GF(4,"a"))
sage: C2.gen_mat()
[    1     0     0     1     1]
[    0     1     0     1 a + 1]
[    0     0     1     1     a]
```

gens()

genus()

Returns the "Duursma genus" of the code, $\gamma_C = n + 1 - k - d$.

EXAMPLES:

```
sage: C1 = HammingCode(3,GF(2)); C1
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C1.genus()
1
sage: C2 = HammingCode(2,GF(4,"a")); C2
Linear code of length 5, dimension 3 over Finite Field in a of size 2^2
sage: C2.genus()
0
```

Since all Hamming codes have minimum distance 3, these computations agree with the definition, n+1-k-d.

is_galois_closed()

Checks if C is equal to its Galois closure.

is_permutation_automorphism(g)

Returns 1 if g is an element of S_n (n = length of self) and if g is an automorphism of self.

EXAMPLES:

```
sage: C = HammingCode(3,GF(3))
sage: g = SymmetricGroup(13).random_element()
sage: C.is_permutation_automorphism(g)
0
sage: MS = MatrixSpace(GF(2),4,8)
sage: G  = MS([[1,0,0,0,1,1,1,0],[0,1,1,1,0,0,0,0],[0,0,0,0,0,0,0,1],[0,0,0,0,0,1,0,0]])
sage: C  = LinearCode(G)
sage: S8 = SymmetricGroup(8)
sage: g = S8("(2,3)")
sage: C.is_permutation_automorphism(g)
```

```
1
sage: g = S8("(1,2,3,4)")
sage: C.is_permutation_automorphism(g)
0
```

is_permutation_equivalent(*other, method=None*)

Returns true if self and other are permutation equivalent codes and false otherwise. The method="verbose" option also returns a permutation (if true) sending self to other. Uses Robert Miller's double coset partition refinement work.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(2),"x")
sage: g = x^3+x+1
sage: C1 = CyclicCodeFromGeneratingPolynomial(7,g); C1
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C2 = HammingCode(3,GF(2)); C2
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C1.is_permutation_equivalent(C2)
True
sage: C1.is_permutation_equivalent(C2,method="verbose")
(True, (4,6,5,7))
sage: C1 = RandomLinearCode(10,5,GF(2))
sage: C2 = RandomLinearCode(10,5,GF(3))
sage: C1.is_permutation_equivalent(C2)
False
```

is_self_dual()

A code C is self-dual if C == C.dual_code() is True.

Returns True if the code is self-dual (in the usual Hamming inner product) and False otherwise.

EXAMPLES:

```
sage: C = ExtendedBinaryGolayCode()
sage: C.is_self_dual()
True
sage: C = HammingCode(3,GF(2))
sage: C.is_self_dual()
False
```

is_self_orthogonal(*C*)

A code C is self-orthogonal if C is a subcode of C.dual_code().

Returns True if the code is self-dual (in the usual Hamming inner product) and False otherwise.

EXAMPLES:

```
sage: C = ExtendedBinaryGolayCode()
sage: C.is_self_orthogonal()
True
sage: C = HammingCode(3,GF(2))
sage: C.is_self_orthogonal()
False
sage: C = QuasiQuadraticResidueCode(11)
sage: C.is_self_orthogonal()
True
```

is_subcode(*other*)

Returns true if the first is a subcode of the second.

EXAMPLES:

```
sage: C1 = HammingCode(3,GF(2))
sage: G1 = C1.gen_mat()
sage: G2 = G1.matrix_from_rows([0,1,2])
sage: C2 = LinearCode(G2)
sage: C2.is_subcode(C1)
True
sage: C1.is_subcode(C2)
False
sage: C3 = C1.extended_code()
sage: C1.is_subcode(C3)
False
sage: C4 = C1.punctured([1])
sage: C4.is_subcode(C1)
False
sage: C5 = C1.shortened([1])
sage: C5.is_subcode(C1)
False
sage: C1 = HammingCode(3,GF(9,"z"))
sage: G1 = C1.gen_mat()
sage: G2 = G1.matrix_from_rows([0,1,2])
sage: C2 = LinearCode(G2)
sage: C2.is_subcode(C1)
True
```

length()

list()

Return list of all elements of this linear code.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: Clist = C.list()
sage: Clist[5]; Clist[5] in C
(1, 0, 1, 0, 0, 1, 1)
True
```

minimum_distance()

If q is not 2 or 3 then this uses a GAP kernel function (in C) written by Steve Linton. If q is 2 or 3 then this uses a very fast program written in C written by CJ Tjhal (this is much faster, except in some small examples).

EXAMPLES:

```
sage: MS = MatrixSpace(GF(3),4,7)
sage: G = MS([[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0], [1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.minimum_distance()
3
sage: C = HammingCode(2,GF(4,"a")); C
Linear code of length 5, dimension 3 over Finite Field in a of size 2^2
sage: C.minimum_distance()
3
```

module_composition_factors(*gp*)

Prints the GAP record of the Meataxe composition factors module in Meataxe notation.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,8)
sage: G  = MS([[1,0,0,0,1,1,1,0],[0,1,1,1,0,0,0,0],[0,0,0,0,0,0,0,1],[0,0,0,0,0,1,0,0]])
sage: C  = LinearCode(G)
```

```
sage: gp = C.automorphism_group_binary_code()
```

Now type "C.module_composition_factors(gp)" to get the record printed.

permutation_automorphism_group(*method='partition'*)

If C is an $[n, k, d]$ code over F, this function computes the subgroup $Aut(C) \subset S_n$ of all permutation automorphisms of C. The binary case always uses the (default) partition refinement method of Robert Miller.

Options: If method="gap" then GAP's MatrixAutomorphism function (written by Thomas Breuer) is used. The implementation combines an idea of mine with an improvement suggested by Cary Huffman. If method="gap+verbose" then code-theoretic data is printed out at several stages of the computation. If method="partition" then the (default) partition refinement method of Robert Miller is used.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,8)
sage: G  = MS([[1,0,0,0,1,1,1,0],[0,1,1,1,0,0,0,0],[0,0,0,0,0,0,0,1],[0,0,0,0,0,1,0,0]])
sage: C  = LinearCode(G)
sage: C
Linear code of length 8, dimension 4 over Finite Field of size 2
sage: G = C.permutation_automorphism_group()
sage: G.order()
144
```

A less easy example involves showing that the permutation automorphism group of the extended ternary Golay code is the Mathieu group M_{11}.

```
sage: C = ExtendedTernaryGolayCode()
sage: M11 = MathieuGroup(11)
sage: M11.order()
7920
sage: G = C.permutation_automorphism_group()  # this should take < 5 seconds
sage: G.is_isomorphic(M11)                    # this should take < 5 seconds
True
```

In the binary case, uses sage.coding.binary_code:

```
sage: C = ExtendedBinaryGolayCode()
sage: G = C.permutation_automorphism_group()
sage: G.order()
244823040
```

In the non-binary case:

```
sage: C = HammingCode(2,GF(3)); C
Linear code of length 4, dimension 2 over Finite Field of size 3
sage: C.permutation_automorphism_group(method="partition")
Permutation Group with generators [(1,2,3)]
sage: C = HammingCode(2,GF(4,"z")); C
Linear code of length 5, dimension 3 over Finite Field in z of size 2^2
sage: C.permutation_automorphism_group(method="partition")
Permutation Group with generators [(1,2)(3,4), (1,3)(2,4)]
sage: C.permutation_automorphism_group(method="gap")
Permutation Group with generators [(1,2)(3,4), (1,3)(2,4)]
sage: C = TernaryGolayCode()
sage: C.permutation_automorphism_group(method="gap")
Permutation Group with generators [(3,4)(5,7)(6,9)(8,11), (3,5,8)(4,11,7)(6,9,10), (2,3)(4,6
```

However, the option `method="gap+verbose"`, will print out

Minimum distance: 5 Weight distribution: [1, 0, 0, 0, 0, 132, 132, 0, 330, 110, 0, 24]

Using the 132 codewords of weight 5 Supergroup size: 39916800

in addition to the output of C.permutation_automorphism_group(method="gap").

permuted_code(p)

Returns the permuted code - the code C which is equivalent to self via the column permutation p.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: G = C.automorphism_group_binary_code(); G
Permutation Group with generators [(3,4)(5,6), (3,5)(4,6), (2,3)(5,7), (1,2)(5,6)]
sage: g = G("(2,3)(5,7)")
sage: Cg = C.permuted_code(g)
sage: Cg
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C == Cg
True
```

punctured(L)

Returns the code punctured at the positions L, $L \subset \{1, 2, ..., n\}$. If C is a code of length n in GF(q) then the code C^L obtained from C by puncturing at the positions in L is the code of length n-L consisting of codewords of C which have their $i - th$ coordinate deleted if $i \in L$ and left alone if $i \notin L$:

$$C^L = \{(c_{i_1}, ..., c_{i_N}) \mid (c_1, ..., c_n) \in C\},$$

where $\{1, 2, ..., n\} - T = \{i_1, ..., i_N\}$. In particular, if $L = \{j\}$ then C^L is simply the code obtained from C by deleting the $j - th$ coordinate of each codeword. The code C^L is called the punctured code at L. The dimension of C^L can decrease if $|L| > d - 1$.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: C.punctured([1,2])
Linear code of length 5, dimension 4 over Finite Field of size 2
```

random_element()

Returns a random codeword.

EXAMPLES:

```
sage: C = HammingCode(3,GF(4,'a'))
sage: Cc = C.galois_closure(GF(2))
sage: c = C.gen_mat()[1]
sage: V = VectorSpace(GF(4,'a'),21)
sage: c2 = V([x^2 for x in c.list()])
sage: c2 in C
False
sage: c2 in Cc
True
```

redundancy_matrix(C)

If C is a linear [n,k,d] code then this function returns a kx(n-k) matrix A such that G = (I,A) generates a code (in standard form) equiv to C. If C is already in standard form and G = (I,A) is its gen mat then this function simply returns that A.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: C.gen_mat()
[1 0 0 1 0 1 0]
[0 1 0 1 0 1 1]
[0 0 1 1 0 0 1]
[0 0 0 0 1 1 1]
sage: C.redundancy_matrix()
[1 1 0]
[1 1 1]
```

```
[1 0 1]
[0 1 1]
sage: C.standard_form()[0].gen_mat()
[1 0 0 0 1 1 0]
[0 1 0 0 1 1 1]
[0 0 1 0 1 0 1]
[0 0 0 1 0 1 1]
sage: C = HammingCode(2,GF(3))
sage: C.gen_mat()
[1 0 2 2]
[0 1 2 1]
sage: C.redundancy_matrix()
[2 2]
[2 1]
```

sd_duursma_data(*C, i*)

INPUT: The formally s.d. code C and the type number (1,2,3,4) (does not check if C is actually sd)

RETURN: The data v,m as in Duursama [D]

EXAMPLES:

REFERENCES:

- [D] I. Duursma, "Extremal weight enumerators and ultraspherical polynomials"

sd_duursma_q(*C, i, d0*)

INPUT:

- `C` - an sd code (does not check if C is actually an sd code),
- `i` - the type number, one of 1,2,3,4,
- `d0` - and the divisor d0 (the smallest integer d00 such that each A_i0 iff i is divisible by d0).

RETURN: The coefficients $q_0, q_1, ...$, of $q(T)$ as in Duursama [D].

EXAMPLES:

REFERENCES:

- [D] I. Duursma, "Extremal weight enumerators and ultraspherical polynomials"

sd_zeta_polynomial(*C, typ=1*)

Returns the Duursma zeta function of a self-dual code using the construction in [D].

INPUT:

- `typ` - type of the s.d. code; one of 1,2,3, or 4.

EXAMPLES:

```
sage: C1 = HammingCode(3,GF(2))
sage: C2 = C1.extended_code(); C2
Linear code of length 8, dimension 4 over Finite Field of size 2
sage: C2.is_self_dual()
True
sage: C2.sd_zeta_polynomial()
2/5*T^2 + 2/5*T + 1/5
sage: C2.zeta_polynomial()
2/5*T^2 + 2/5*T + 1/5
sage: P = C2.sd_zeta_polynomial(); P(1)
1
sage: F.<z> = GF(4,"z")
sage: MS = MatrixSpace(F, 3, 6)
sage: G = MS([[1,0,0,1,z,z],[0,1,0,z,1,z],[0,0,1,z,z,1]])
sage: C = LinearCode(G)  # the "hexacode"
sage: C.sd_zeta_polynomial(4)
1
```

It is a general fact about Duursma zeta polynomials that P(1) = 1.

REFERENCES:

•[D] I. Duursma, "Extremal weight enumerators and ultraspherical polynomials"

shortened(*L*)

Returns the code shortened at the positions L, $L \subset \{1, 2, ..., n\}$. Consider the subcode $C(L)$ consisting of all codewords $c \in C$ which satisfy $c_i = 0$ for all $i \in L$. The punctured code $C(L)^L$ is called the shortened code on L and is denoted C_L. The code constructed is actually only isomorphic to the shortened code defined in this way.

By Theorem 1.5.7 in [HP], C_L is $((C^\perp)^L)^\perp$. This is used in the construction below.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: C.shortened([1,2])
Linear code of length 5, dimension 2 over Finite Field of size 2
```

spectrum(*method=None*)

The default method (gap) uses a GAP kernel function (in C) written by Steve Linton.

METHOD:

•`None` - defaults to gap except in the binary case

•`gap` - uses the GAP function

•`leon` - uses Jeffrey Leon's software via Guava

•`binary` - uses Sage native Cython code

The optional method (leon) may create a stack smashing error and a traceback but should return the correct answer. It appears to run much faster than the"gap" method in some ("small") examples and much slower than the "gap" method in other ("larger") examples.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G = MS([[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.spectrum()
[1, 0, 0, 7, 7, 0, 0, 1]
sage: F.<z> = GF(2^2,"z")
sage: C = HammingCode(2, F); C
Linear code of length 5, dimension 3 over Finite Field in z of size 2^2
sage: C.spectrum()
[1, 0, 0, 30, 15, 18]
sage: C = HammingCode(3,GF(2)); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.spectrum(method="leon")
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C = HammingCode(3,GF(3)); C
Linear code of length 13, dimension 10 over Finite Field of size 3
sage: C.spectrum() == C.spectrum(method="leon")
True
sage: C = HammingCode(2,GF(5)); C
Linear code of length 6, dimension 4 over Finite Field of size 5
sage: C.spectrum() == C.spectrum(method="leon")
True
sage: C = HammingCode(2,GF(7)); C
Linear code of length 8, dimension 6 over Finite Field of size 7
sage: C.spectrum() == C.spectrum(method="leon")
True
```

standard_form()

An $[n, k]$ linear code with generator matrix G is in standard form is the row-reduced echelon form of G is

(I, A), where I denotes the $k \times k$ identity matrix and A is a $k \times (n-k)$ block. This method returns a pair (C, p) where C is a code permutation equivalent to self and p in S_n (n = length of C) is the permutation sending self to C. This does not call GAP.

Thanks to Frank Luebeck for (the GAP version of) this code.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: C.gen_mat()
[1 0 0 1 0 1 0]
[0 1 0 1 0 1 1]
[0 0 1 1 0 0 1]
[0 0 0 0 1 1 1]
sage: Cs,p = C.standard_form()
sage: p
(4,5)
sage: Cs
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: Cs.gen_mat()
[1 0 0 0 1 1 0]
[0 1 0 0 1 1 1]
[0 0 1 0 1 0 1]
[0 0 0 1 0 1 1]
sage: MS = MatrixSpace(GF(3),3,7)
sage: G = MS([[1,0,0,0,1,1,0],[0,1,0,1,0,1,0],[0,0,0,0,0,0,1]])
sage: C = LinearCode(G)
sage: G; C.standard_form()[0].gen_mat()
[1 0 0 0 1 1 0]
[0 1 0 1 0 1 0]
[0 0 0 0 0 0 1]
[1 0 0 0 1 1 0]
[0 1 0 1 0 1 0]
[0 0 1 0 0 0 0]
sage: C.standard_form()[1]
(3,7)
```

support()

Returns the set of indices j where A_j is nonzero, where spectrum(self) = $[A_0, A_1, ..., A_n]$.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: C.spectrum()
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C.support()
[0, 3, 4, 7]
```

weight_distribution(*method=None*)

The default method (gap) uses a GAP kernel function (in C) written by Steve Linton.

METHOD:

- `None` - defaults to gap except in the binary case
- `gap` - uses the GAP function
- `leon` - uses Jeffrey Leon's software via Guava
- `binary` - uses Sage native Cython code

The optional method (leon) may create a stack smashing error and a traceback but should return the correct answer. It appears to run much faster than the"gap" method in some ("small") examples and much slower than the "gap" method in other ("larger") examples.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),4,7)
sage: G = MS([[1,1,1,0,0,0,0],[1,0,0,1,1,0,0],[0,1,0,1,0,1,0],[1,1,0,1,0,0,1]])
sage: C = LinearCode(G)
sage: C.spectrum()
[1, 0, 0, 7, 7, 0, 0, 1]
sage: F.<z> = GF(2^2,"z")
sage: C = HammingCode(2, F); C
Linear code of length 5, dimension 3 over Finite Field in z of size 2^2
sage: C.spectrum()
[1, 0, 0, 30, 15, 18]
sage: C = HammingCode(3,GF(2)); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.spectrum(method="leon")
[1, 0, 0, 7, 7, 0, 0, 1]
sage: C = HammingCode(3,GF(3)); C
Linear code of length 13, dimension 10 over Finite Field of size 3
sage: C.spectrum() == C.spectrum(method="leon")
True
sage: C = HammingCode(2,GF(5)); C
Linear code of length 6, dimension 4 over Finite Field of size 5
sage: C.spectrum() == C.spectrum(method="leon")
True
sage: C = HammingCode(2,GF(7)); C
Linear code of length 8, dimension 6 over Finite Field of size 7
sage: C.spectrum() == C.spectrum(method="leon")
True
```

weight_enumerator(*names='xy'*)

Returns the weight enumerator of the code.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: C.weight_enumerator()
x^7 + 7*x^4*y^3 + 7*x^3*y^4 + y^7
sage: C.weight_enumerator(names="st")
s^7 + 7*s^4*t^3 + 7*s^3*t^4 + t^7
```

zeta_function(*name='T'*)

Returns the Duursma zeta function of the code.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: C.zeta_function()
(2/5*T^2 + 2/5*T + 1/5)/(2*T^2 - 3*T + 1)
```

zeta_polynomial(*name='T'*)

Returns the Duursma zeta polynomial of the code C.

Assumes `C.minimum_distance()` 1 and minimum_distance $(C^{\perp}) > 1$.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: C.zeta_polynomial()
2/5*T^2 + 2/5*T + 1/5
sage: C = best_known_linear_code(6,3,GF(2))
sage: C.minimum_distance()
3
sage: C.zeta_polynomial()
2/5*T^2 + 2/5*T + 1/5
```

```
sage: C = HammingCode(4,GF(2))
sage: C.zeta_polynomial()
16/429*T^6 + 16/143*T^5 + 80/429*T^4 + 32/143*T^3 + 30/143*T^2 + 2/13*T + 1/13
sage: F.<z> = GF(4,"z")
sage: MS = MatrixSpace(F, 3, 6)
sage: G = MS([[1,0,0,1,z,z],[0,1,0,z,1,z],[0,0,1,z,z,1]])
sage: C = LinearCode(G)  # the "hexacode"
sage: C.zeta_polynomial()
1
```

REFERENCES:

- I. Duursma, "From weight enumerators to zeta functions", in Discrete Applied Mathematics, vol. 111, no. 1-2, pp. 55-73, 2001.

LinearCodeFromVectorSpace(*self*)

Simply converts a vector subspace V of $GF(q)^n$ into a LinearCode.

best_known_linear_code(*n, k, F*)

best_known_linear_code returns the best known (as of 11 May 2006) linear code of length n, dimension k over field F. The function uses the tables described in bounds_minimum_distance to construct this code.

This does not require an internet connection.

EXAMPLES:

```
sage: best_known_linear_code(10,5,GF(2))     # long time
Linear code of length 10, dimension 5 over Finite Field of size 2
sage: gap.eval("C:=BestKnownLinearCode(10,5,GF(2))")      # long time
'a linear [10,5,4]2..4 shortened code'
```

This means that best possible binary linear code of length 10 and dimension 5 is a code with minimum distance 4 and covering radius somewhere between 2 and 4. Use `minimum_distance_why(10,5,GF(2))` or `print bounds_minimum_distance(10,5,GF(2))` for further details.

best_known_linear_code_www(*n, k, F, verbose=False*)

Explains the construction of the best known linear code over GF(q) with length n and dimension k, courtesy of the www page http://www.codetables.de/.

INPUT:

- `n` - integer, the length of the code
- `k` - integer, the dimension of the code
- `F` - finite field, whose field order must be in [2, 3, 4, 5, 7, 8, 9]
- `verbose` - bool (default=False), print verbose message

OUTPUT:

- `str` - text about why the bounds are as given

EXAMPLES:

```
sage: L = best_known_linear_code_www(72, 36, GF(2)) # requires internet, optional
sage: print L                                       # requires internet, optional
Construction of a linear code
[72,36,15] over GF(2):
[1]:  [73, 36, 16] Cyclic Linear Code over GF(2)
     CyclicCode of length 73 with generating polynomial x^37 + x^36 + x^34 +
x^33 + x^32 + x^27 + x^25 + x^24 + x^22 + x^21 + x^19 + x^18 + x^15 + x^11 +
x^10 + x^8 + x^7 + x^5 + x^3 + 1
[2]:  [72, 36, 15] Linear Code over GF(2)
```

```
     Puncturing of [1] at 1
last modified: 2002-03-20
```

This function raises an IOError if an error occurs downloading data or parsing it. It raises a ValueError if the q input is invalid.

AUTHORS:

- Steven Sivek (2005-11-14)
- David Joyner (2008-03)

bounds_minimum_distance(*n, k, F*)

The function bounds_minimum_distance calculates a lower and upper bound for the minimum distance of an optimal linear code with word length n, dimension k over field F. The function returns a record with the two bounds and an explanation for each bound. The function Display can be used to show the explanations.

The values for the lower and upper bound are obtained from a table constructed by Cen Tjhai for GUAVA, derived from the table of Brouwer. (See http://www.win.tue.nl/ aeb/voorlincod.html or use the Sage function minimum_distance_why for the most recent data.) These tables contain lower and upper bounds for q=2 (n <= 257), 3 (n <= 243), 4 (n <= 256). (Current as of 11 May 2006.) For codes over other fields and for larger word lengths, trivial bounds are used.

This does not require an internet connection. The format of the output is a little non-intuitive. Try print bounds_minimum_distance(10,5,GF(2)) for example.

code2leon(*C*)

Writes a file in Sage's temp directory representing the code C, returning the absolute path to the file. This is the Sage translation of the GuavaToLeon command in Guava's codefun.gi file.

INPUT:

- C - a linear code (over GF(p), p < 11)

OUTPUT:

- Absolute path to the file written.

EXAMPLES:

```
sage: C = HammingCode(3,GF(2)); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: file_loc = sage.coding.linear_code.code2leon(C)
sage: f = open(file_loc); print f.read()
LIBRARY code;
code=seq(2,4,7,seq(
1,0,0,1,0,1,0,
0,1,0,1,0,1,1,
0,0,1,1,0,0,1,
0,0,0,0,1,1,1
));
FINISH;
sage: f.close()
```

hamming_weight(*v*)

min_wt_vec_gap(*Gmat, F*)

Uses C programs written by Steve Linton in the kernel of GAP, so is fairly fast.

INPUT: Same as wtdist.

OUTPUT: Returns a minimum weight vector v of the code generated by Gmat ## , the "message" vector m such that m*G = v, and the (minimum) distance, as a triple.

EXAMPLES:

```
sage: Gstr = "Z(2)*[[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0], [1,1,0,1,0,0,1]]"
sage: sage.coding.linear_code.min_wt_vec_gap(Gstr,GF(2))
(0, 0, 1, 0, 1, 1, 0)
```

Here Gstr is a generator matrix of the Hamming [7,4,3] binary code.

AUTHORS:

- David Joyner (11-2005)

self_orthogonal_binary_codes(*n, k, b=2, parent=None, BC=None, equal=False, in_test=None*)

Returns a Python iterator which generates a complete set of representatives of all permutation equivalence classes of self-orthogonal binary linear codes of length in [1..n] and dimension in [1..k].

INPUT:

- `n` - maximal length
- `k` - maximal dimension
- `b` - require that the generators all have weight divisible by b (if b=2, all self-orthogonal codes are generated, and if b=4, all doubly even codes are generated). Must be an even positive integer.
- `parent` - default None, used in recursion
- `BC` - default None, used in recursion
- `equal` - default False, if True generates only [n, k] codes
- `in_test` - default None, used in recursion

EXAMPLES: Generate all self-orthogonal codes of length up to 7 and dimension up to 3:

```
sage: for B in self_orthogonal_binary_codes(7,3):
...     print B
...
Linear code of length 2, dimension 1 over Finite Field of size 2
Linear code of length 4, dimension 2 over Finite Field of size 2
Linear code of length 6, dimension 3 over Finite Field of size 2
Linear code of length 4, dimension 1 over Finite Field of size 2
Linear code of length 6, dimension 2 over Finite Field of size 2
Linear code of length 6, dimension 2 over Finite Field of size 2
Linear code of length 7, dimension 3 over Finite Field of size 2
Linear code of length 6, dimension 1 over Finite Field of size 2
```

Generate all doubly-even codes of length up to 7 and dimension up to 3:

```
sage: for B in self_orthogonal_binary_codes(7,3,4):
...     print B; print B.gen_mat()
...
Linear code of length 4, dimension 1 over Finite Field of size 2
[1 1 1 1]
Linear code of length 6, dimension 2 over Finite Field of size 2
[1 1 1 1 0 0]
[0 1 0 1 1 1]
Linear code of length 7, dimension 3 over Finite Field of size 2
[1 0 1 1 0 1 0]
[0 1 0 1 1 1 0]
[0 0 1 0 1 1 1]
```

Generate all doubly-even codes of length up to 7 and dimension up to 2:

```
sage: for B in self_orthogonal_binary_codes(7,2,4):
...    print B; print B.gen_mat()
Linear code of length 4, dimension 1 over Finite Field of size 2
[1 1 1 1]
Linear code of length 6, dimension 2 over Finite Field of size 2
[1 1 1 1 0 0]
[0 1 0 1 1 1]
```

Generate all self-orthogonal codes of length equal to 8 and dimension equal to 4:

```
sage: for B in self_orthogonal_binary_codes(8, 4, equal=True):
...     print B; print B.gen_mat()
Linear code of length 8, dimension 4 over Finite Field of size 2
[1 0 0 1 0 0 0 0]
[0 1 0 0 1 0 0 0]
[0 0 1 0 0 1 0 0]
[0 0 0 0 0 0 1 1]
Linear code of length 8, dimension 4 over Finite Field of size 2
[1 0 0 1 1 0 1 0]
[0 1 0 1 1 1 0 0]
[0 0 1 0 1 1 1 0]
[0 0 0 1 0 1 1 1]
```

Since all the codes will be self-orthogonal, b must be divisible by 2:

```
sage: list(self_orthogonal_binary_codes(8, 4, 1, equal=True))
...
ValueError: b (1) must be a positive even integer.
```

wtdist_gap(*Gmat, F*)

INPUT:

- `Gmat` - a string representing a GAP generator matrix G of a linear code.
- `F` - a (Sage) finite field - the base field of the code.

OUTPUT: Returns the spectrum of the associated code.

EXAMPLES:

```
sage: Gstr = 'Z(2)*[[1,1,1,0,0,0,0], [1,0,0,1,1,0,0], [0,1,0,1,0,1,0], [1,1,0,1,0,0,1]]'
sage: F = GF(2)
sage: sage.coding.linear_code.wtdist_gap(Gstr, F)
[1, 0, 0, 7, 7, 0, 0, 1]
```

Here Gstr is a generator matrix of the Hamming [7,4,3] binary code.

ALGORITHM: Uses C programs written by Steve Linton in the kernel of GAP, so is fairly fast.

AUTHORS:

- David Joyner (2005-11)

40.2 Linear code constructions.

AUTHOR:

- David Joyner (2007-05): initial version

- " (2008-02): added cyclic codes, Hamming codes
- " (2008-03): added BCH code, LinearCodeFromCheckmatrix, ReedSolomonCode, WalshCode, DuadicCodeEvenPair, DuadicCodeOddPair, QR codes (even and odd)
- " (2008-09) fix for bug in BCHCode reported by F. Voloch
- " (2008-10) small docstring changes to WalshCode and walsh_matrix

This file contains constructions of error-correcting codes which are pure Python/Sage and not obtained from wrapping GUAVA functions. The GUAVA wrappers are in guava.py.

Let F be a finite field with q elements. Here's a constructive definition of a cyclic code of length n.

1. Pick a monic polynomial $g(x) \in F[x]$ dividing $x^n - 1$. This is called the generating polynomial of the code.
2. For each polynomial $p(x) \in F[x]$, compute $p(x)g(x) \pmod{x^n - 1}$.'$Denotetheanswerby$'$c_0 + c_1x + ... + c_{n-1}x^{n-1}$.
3. $\mathbf{c} = (c_0, c_1, ..., c_{n-1})$ is a codeword in C. Every codeword in C arises in this way (from some $p(x)$).

The polynomial notation for the code is to call $c_0 + c_1x + ... + c_{n-1}x^{n-1}$ the codeword (instead of $(c_0, c_1, ..., c_{n-1})$). The polynomial $h(x) = (x^n - 1)/g(x)$ is called the check polynomial of C.

Let n be a positive integer relatively prime to q and let α be a primitive n-th root of unity. Each generator polynomial g of a cyclic code C of length n has a factorization of the form

$$g(x) = (x - \alpha^{k_1})...(x - \alpha^{k_r}),$$

where $\{k_1, ..., k_r\} \subset \{0, ..., n - 1\}$. The numbers α^{k_i}, $1 \leq i \leq r$, are called the zeros of the code C. Many families of cyclic codes (such as BCH codes and the quadratic residue codes) are defined using properties of the zeros of C.

- BCHCode - A 'Bose-Chaudhuri-Hockenghem code' (or BCH code for short) is the largest possible cyclic code of length n over field F=GF(q), whose generator polynomial has zeros (which contain the set) $Z = \{a^i \mid i \in C_b \cup ...C_{b+delta-2}\}$, where a is a primitive n^{th} root of unity in the splitting field $GF(q^m)$, b is an integer $0 \leq b \leq n - delta + 1$ and m is the multiplicative order of q modulo n. The default here is $b = 0$ (unlike Guava, which has default $b = 1$). Here C_k are the cyclotomic codes (see `cyclotomic_cosets`).
- BinaryGolayCode, ExtendedBinaryGolayCode, TernaryGolayCode, ExtendedTernaryGolayCode the well-known"extremal" Golay codes, http://en.wikipedia.org/wiki/Golay_code
- cyclic codes - CyclicCodeFromGeneratingPolynomial (= CyclicCode), CyclicCodeFromCheckPolynomial, http://en.wikipedia.org/wiki/Cyclic_code
- DuadicCodeEvenPair, DuadicCodeOddPair: Constructs the "even (resp. odd) pair" of duadic codes associated to the "splitting" S1, S2 of n. This is a special type of cyclic code whose generator is determined by S1, S2. See chapter 6 in [HP].
- HammingCode - the well-known Hamming code, http://en.wikipedia.org/wiki/Hamming_code
- LinearCodeFromCheckMatrix - for specifying the code using the check matrix instead of the generator matrix.
- QuadraticResidueCodeEvenPair, QuadraticResidueCodeOddPair: Quadratic residue codes of a given odd prime length and base ring either don't exist at all or occur as 4-tuples - a pair of "odd-like" codes and a pair of "even-like" codes. If n 2 is prime then (Theorem 6.6.2 in [HP]) a QR code exists over GF(q) iff q is a quadratic residue mod n. Here they are constructed as"even-like" duadic codes associated the splitting (Q,N) mod n, where Q is the set of non-zero quadratic residues and N is the non-residues. QuadraticResidueCode (a special case) and ExtendedQuadraticResidueCode are included as well.

- RandomLinearCode - Repeatedly applies Sage's random_element applied to the ambient MatrixSpace of the generator matrix until a full rank matrix is found.
- ReedSolomonCode - Given a finite field F of order q, let n and k be chosen such that $1 \leq k \leq n \leq q$. Pick n distinct elements of F, denoted $\{x_1, x_2, ..., x_n\}$. Then, the codewords are obtained by evaluating every polynomial in $F[x]$ of degree less than k at each x_i.
- ToricCode - Let P denote a list of lattice points in $\mathbf{Z}^d$ and let T denote a listing of all points in $(F^x)^d$. Put $n = |T|$ and let k denote the dimension of the vector space of functions $V = \mathrm{Span}\{x^e \mid e \in P\}$. The associated toric code C is the evaluation code which is the image of the evaluation map $eval_T : V \to F^n$, where x^e is the multi-index notation.
- WalshCode - a binary linear $[2^m, m, 2^{m-1}]$ code related to Hadamard matrices. http://en.wikipedia.org/wiki/Walsh_code

Please see the docstrings below for further details.

BCHCode (*n, delta, F, b=0*)

A 'Bose-Chaudhuri-Hockenghem code' (or BCH code for short) is the largest possible cyclic code of length n over field F=GF(q), whose generator polynomial has zeros (which contain the set) $Z = \{a^b, a^{b+1}, ..., a^{b+delta-2}\}$, where a is a primitive n^{th} root of unity in the splitting field $GF(q^m)$, b is an integer $0 \leq b \leq n - delta + 1$ and m is the multiplicative order of q modulo n. (The integers $b, ..., b + delta - 2$ typically lie in the range $1, ..., n - 1$.) The integer $delta \geq 1$ is called the "designed distance". The length n of the code and the size q of the base field must be relatively prime. The generator polynomial is equal to the least common multiple of the minimal polynomials of the elements of the set Z above.

Special cases are b=1 (resulting codes are called 'narrow-sense' BCH codes), and $n = q^m - 1$ (known as 'primitive' BCH codes).

It may happen that several values of delta give rise to the same BCH code. The largest one is called the Bose distance of the code. The true minimum distance, d, of the code is greater than or equal to the Bose distance, so $d \geq delta$.

EXAMPLES:

```
sage: FF.<a> = GF(3^2,"a")
sage: x = PolynomialRing(FF,"x").gen()
sage: L = [b.minpoly() for b in [a,a^2,a^3]]; g = LCM(L)
sage: f = x^(8)-1
sage: g.divides(f)
True
sage: C = CyclicCode(8,g); C
Linear code of length 8, dimension 4 over Finite Field of size 3
sage: C.minimum_distance()
4
sage: C = BCHCode(8,3,GF(3),1); C
Linear code of length 8, dimension 4 over Finite Field of size 3
sage: C.minimum_distance()
4
sage: C = BCHCode(8,3,GF(3)); C
Linear code of length 8, dimension 5 over Finite Field of size 3
sage: C.minimum_distance()
3
sage: C = BCHCode(26, 5, GF(5), b=1); C
Linear code of length 26, dimension 10 over Finite Field of size 5
```

REFERENCES:

•[HP] W. C. Huffman, V. Pless, Fundamentals of Error-Correcting Codes, Cambridge Univ. Press, 2003.

BinaryGolayCode()

BinaryGolayCode() returns a binary Golay code. This is a perfect [23,12,7] code. It is also (equivalent to) a cyclic code, with generator polynomial $g(x) = 1 + x^2 + x^4 + x^5 + x^6 + x^{10} + x^{11}$. Extending it yields the extended Golay code (see ExtendedBinaryGolayCode).

EXAMPLE:

```
sage: C = BinaryGolayCode()
sage: C
Linear code of length 23, dimension 12 over Finite Field of size 2
sage: C.minimum_distance()                # long time
7
```

AUTHORS:

- David Joyner (2007-05)

CyclicCode(*n, g, ignore=True*)

If g is a polynomial over GF(q) which divides $x^n - 1$ then this constructs the code "generated by g" (ie, the code associated with the principle ideal gR in the ring $R = GF(q)[x]/(x^n - 1)$ in the usual way).

The option "ignore" says to ignore the condition that (a) the characteristic of the base field does not divide the length (the usual assumption in the theory of cyclic codes), and (b) g must divide $x^n - 1$. If ignore=True, instead of returning an error, a code generated by $gcd(x^n - 1, g)$ is created.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x-1
sage: C = CyclicCodeFromGeneratingPolynomial(4,g); C
Linear code of length 4, dimension 3 over Finite Field of size 3
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^3+1
sage: C = CyclicCodeFromGeneratingPolynomial(9,g); C
Linear code of length 9, dimension 6 over Finite Field in a of size 2^2
sage: P.<x> = PolynomialRing(GF(2),"x")
sage: g = x^3+x+1
sage: C = CyclicCodeFromGeneratingPolynomial(7,g); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.gen_mat()
[1 1 0 1 0 0 0]
[0 1 1 0 1 0 0]
[0 0 1 1 0 1 0]
[0 0 0 1 1 0 1]
sage: g = x+1
sage: C = CyclicCodeFromGeneratingPolynomial(4,g); C
Linear code of length 4, dimension 3 over Finite Field of size 2
sage: C.gen_mat()
[1 1 0 0]
[0 1 1 0]
[0 0 1 1]
```

On the other hand, CyclicCodeFromPolynomial(4,x) will produce a ValueError including a traceback error message: "x must divide $x^4 - 1$". You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^2+1
```

followed by CyclicCodeFromGeneratingPolynomial(6,g). You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x^2-1
sage: C = CyclicCodeFromGeneratingPolynomial(5,g); C
Linear code of length 5, dimension 4 over Finite Field of size 3
```

followed by C = CyclicCodeFromGeneratingPolynomial(5,g,False), with a traceback message including "x^2+2 must divide x^5-1".

CyclicCodeFromCheckPolynomial (*n, h, ignore=True*)

If h is a polynomial over GF(q) which divides x^n-1 then this constructs the code "generated by $g=(x^n-1)/h$" (ie, the code associated with the principle ideal gR in the ring $R=GF(q)[x]/(x^n-1)$ in the usual way). The option "ignore" says to ignore the condition that the characteristic of the base field does not divide the length (the usual assumption in the theory of cyclic codes).

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: C = CyclicCodeFromCheckPolynomial(4,x + 1); C
Linear code of length 4, dimension 1 over Finite Field of size 3
sage: C = CyclicCodeFromCheckPolynomial(4,x^3 + x^2 + x + 1); C
Linear code of length 4, dimension 3 over Finite Field of size 3
sage: C.gen_mat()
[2 1 0 0]
[0 2 1 0]
[0 0 2 1]
```

CyclicCodeFromGeneratingPolynomial (*n, g, ignore=True*)

If g is a polynomial over GF(q) which divides x^n-1 then this constructs the code "generated by g" (ie, the code associated with the principle ideal gR in the ring $R=GF(q)[x]/(x^n-1)$ in the usual way).

The option "ignore" says to ignore the condition that (a) the characteristic of the base field does not divide the length (the usual assumption in the theory of cyclic codes), and (b) g must divide x^n-1. If ignore=True, instead of returning an error, a code generated by $gcd(x^n-1,g)$ is created.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x-1
sage: C = CyclicCodeFromGeneratingPolynomial(4,g); C
Linear code of length 4, dimension 3 over Finite Field of size 3
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^3+1
sage: C = CyclicCodeFromGeneratingPolynomial(9,g); C
Linear code of length 9, dimension 6 over Finite Field in a of size 2^2
sage: P.<x> = PolynomialRing(GF(2),"x")
sage: g = x^3+x+1
sage: C = CyclicCodeFromGeneratingPolynomial(7,g); C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C.gen_mat()
[1 1 0 1 0 0 0]
[0 1 1 0 1 0 0]
[0 0 1 1 0 1 0]
[0 0 0 1 1 0 1]
sage: g = x+1
sage: C = CyclicCodeFromGeneratingPolynomial(4,g); C
Linear code of length 4, dimension 3 over Finite Field of size 2
sage: C.gen_mat()
[1 1 0 0]
```

```
[0 1 1 0]
[0 0 1 1]
```

On the other hand, CyclicCodeFromPolynomial(4,x) will produce a ValueError including a traceback error message: "x must divide $x^4 - 1$". You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(4,"a"),"x")
sage: g = x^2+1
```

followed by CyclicCodeFromGeneratingPolynomial(6,g). You will also get a ValueError if you type

```
sage: P.<x> = PolynomialRing(GF(3),"x")
sage: g = x^2-1
sage: C = CyclicCodeFromGeneratingPolynomial(5,g); C
Linear code of length 5, dimension 4 over Finite Field of size 3
```

followed by C = CyclicCodeFromGeneratingPolynomial(5,g,False), with a traceback message including "x^2+2 must divide $x^5 - 1$".

DuadicCodeEvenPair(*F, S1, S2*)

Constructs the "even pair" of duadic codes associated to the "splitting" (see the docstring for `is_a_splitting` for the definition) S1, S2 of n.

> **Warning:** Maybe the splitting should be associated to a sum of q-cyclotomic cosets mod n, where q is a *prime*.

EXAMPLES:

```
sage: from sage.coding.code_constructions import is_a_splitting
sage: n = 11; q = 3
sage: C = cyclotomic_cosets(q,n); C
[[0], [1, 3, 4, 5, 9], [2, 6, 7, 8, 10]]
sage: S1 = C[1]
sage: S2 = C[2]
sage: is_a_splitting(S1,S2,11)
(True, 2)
sage: DuadicCodeEvenPair(GF(q),S1,S2)
(Linear code of length 11, dimension 5 over Finite Field of size 3,
 Linear code of length 11, dimension 5 over Finite Field of size 3)
```

DuadicCodeOddPair(*F, S1, S2*)

Constructs the "odd pair" of duadic codes associated to the "splitting" S1, S2 of n.

> **Warning:** Maybe the splitting should be associated to a sum of q-cyclotomic cosets mod n, where q is a *prime*.

EXAMPLES:

```
sage: from sage.coding.code_constructions import is_a_splitting
sage: n = 11; q = 3
sage: C = cyclotomic_cosets(q,n); C
[[0], [1, 3, 4, 5, 9], [2, 6, 7, 8, 10]]
sage: S1 = C[1]
sage: S2 = C[2]
sage: is_a_splitting(S1,S2,11)
(True, 2)
sage: DuadicCodeOddPair(GF(q),S1,S2)
```

```
(Linear code of length 11, dimension 6 over Finite Field of size 3,
 Linear code of length 11, dimension 6 over Finite Field of size 3)
```

This is consistent with Theorem 6.1.3 in [HP].

ExtendedBinaryGolayCode()

ExtendedBinaryGolayCode() returns the extended binary Golay code. This is a perfect [24,12,8] code. This code is self-dual.

EXAMPLES:

```
sage: C = ExtendedBinaryGolayCode()
sage: C
Linear code of length 24, dimension 12 over Finite Field of size 2
sage: C.minimum_distance()
8
```

AUTHORS:

- David Joyner (2007-05)

ExtendedQuadraticResidueCode(*n, F*)

The extended quadratic residue code (or XQR code) is obtained from a QR code by adding a check bit to the last coordinate. (These codes have very remarkable properties such as large automorphism groups and duality properties - see [HP], Section 6.6.3-6.6.4.)

INPUT:

- n - an odd prime
- F - a finite prime field F whose order must be a quadratic residue modulo n.

OUTPUT: Returns an extended quadratic residue code.

EXAMPLES:

```
sage: C1 = QuadraticResidueCode(7,GF(2))
sage: C2 = C1.extended_code()
sage: C3 = ExtendedQuadraticResidueCode(7,GF(2)); C3
Linear code of length 8, dimension 4 over Finite Field of size 2
sage: C2 == C3
True
sage: C = ExtendedQuadraticResidueCode(17,GF(2))
sage: C
Linear code of length 18, dimension 9 over Finite Field of size 2
sage: C3 = QuadraticResidueCodeOddPair(7,GF(2))[0]
sage: C3x = C3.extended_code()
sage: C4 = ExtendedQuadraticResidueCode(7,GF(2))
sage: C3x == C4
True
```

AUTHORS:

- David Joyner (07-2006)

ExtendedTernaryGolayCode()

ExtendedTernaryGolayCode returns a ternary Golay code. This is a self-dual perfect [12,6,6] code.

EXAMPLES:

```
sage: C = ExtendedTernaryGolayCode()
sage: C
Linear code of length 12, dimension 6 over Finite Field of size 3
sage: C.minimum_distance()
6
```

AUTHORS:

- David Joyner (11-2005)

HammingCode(*r*, *F*)

Implements the Hamming codes.

The r^{th} Hamming code over $F = GF(q)$ is an $[n, k, d]$ code with length $n = (q^r - 1)/(q - 1)$, dimension $k = (q^r - 1)/(q - 1) - r$ and minimum distance $d = 3$. The parity check matrix of a Hamming code has rows consisting of all nonzero vectors of length r in its columns, modulo a scalar factor so no parallel columns arise. A Hamming code is a single error-correcting code.

INPUT:

- `r` - an integer 2
- `F` - a finite field.

OUTPUT: Returns the r-th q-ary Hamming code.

EXAMPLES:

```
sage: HammingCode(3,GF(2))
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C = HammingCode(3,GF(3)); C
Linear code of length 13, dimension 10 over Finite Field of size 3
sage: C.minimum_distance()
3
sage: C = HammingCode(3,GF(4,'a')); C
Linear code of length 21, dimension 18 over Finite Field in a of size 2^2
```

LinearCodeFromCheckMatrix(*H*)

A linear [n,k]-code C is uniquely determined by its generator matrix G and check matrix H. We have the following short exact sequence

$$0 \to \mathbf{F}^k \xrightarrow{G} \mathbf{F}^n \xrightarrow{H} \mathbf{F}^{n-k} \to 0.$$

("Short exact" means (a) the arrow G is injective, i.e., G is a full-rank $k \times n$ matrix, (b) the arrow H is surjective, and (c) $\text{image}(G) = \text{kernel}(H)$.)

EXAMPLES:

```
sage: C = HammingCode(3,GF(2))
sage: H = C.check_mat(); H
[1 0 0 1 1 0 1]
[0 1 0 1 0 1 1]
[0 0 1 1 1 1 0]
sage: LinearCodeFromCheckMatrix(H) == C
True
sage: C = HammingCode(2,GF(3))
sage: H = C.check_mat(); H
[1 0 2 2]
[0 1 2 1]
sage: LinearCodeFromCheckMatrix(H) == C
True
```

```
sage: C = RandomLinearCode(10,5,GF(4,"a"))
sage: H = C.check_mat()
sage: LinearCodeFromCheckMatrix(H) == C
True
```

QuadraticResidueCode(*n, F*)

A quadratic residue code (or QR code) is a cyclic code whose generator polynomial is the product of the polynomials $x - \alpha^i$ (α is a primitive n^{th} root of unity; i ranges over the set of quadratic residues modulo n).

See QuadraticResidueCodeEvenPair and QuadraticResidueCodeOddPair for a more general construction.

INPUT:

- n - an odd prime
- F - a finite prime field F whose order must be a quadratic residue modulo n.

OUTPUT: Returns a quadratic residue code.

EXAMPLES:

```
sage: C = QuadraticResidueCode(7,GF(2))
sage: C
Linear code of length 7, dimension 4 over Finite Field of size 2
sage: C = QuadraticResidueCode(17,GF(2))
sage: C
Linear code of length 17, dimension 9 over Finite Field of size 2
sage: C1 = QuadraticResidueCodeOddPair(7,GF(2))[0]
sage: C2 = QuadraticResidueCode(7,GF(2))
sage: C1 == C2
True
sage: C1 = QuadraticResidueCodeOddPair(17,GF(2))[0]
sage: C2 = QuadraticResidueCode(17,GF(2))
sage: C1 == C2
True
```

AUTHORS:

- David Joyner (11-2005)

QuadraticResidueCodeEvenPair(*n, F*)

Quadratic residue codes of a given odd prime length and base ring either don't exist at all or occur as 4-tuples - a pair of "odd-like" codes and a pair of "even-like" codes. If n 2 is prime then (Theorem 6.6.2 in [HP]) a QR code exists over GF(q) iff q is a quadratic residue mod n.

They are constructed as "even-like" duadic codes associated the splitting (Q,N) mod n, where Q is the set of non-zero quadratic residues and N is the non-residues.

EXAMPLES:

```
sage: QuadraticResidueCodeEvenPair(17,GF(13))
(Linear code of length 17, dimension 8 over Finite Field of size 13,
 Linear code of length 17, dimension 8 over Finite Field of size 13)
sage: QuadraticResidueCodeEvenPair(17,GF(2))
(Linear code of length 17, dimension 8 over Finite Field of size 2,
 Linear code of length 17, dimension 8 over Finite Field of size 2)
sage: QuadraticResidueCodeEvenPair(13,GF(9,"z"))
(Linear code of length 13, dimension 6 over Finite Field in z of size 3^2,
 Linear code of length 13, dimension 6 over Finite Field in z of size 3^2)
sage: C1 = QuadraticResidueCodeEvenPair(7,GF(2))[0]
sage: C1.is_self_orthogonal()
True
```

```
sage: C2 = QuadraticResidueCodeEvenPair(7,GF(2))[1]
sage: C2.is_self_orthogonal()
True
sage: C3 = QuadraticResidueCodeOddPair(17,GF(2))[0]
sage: C4 = QuadraticResidueCodeEvenPair(17,GF(2))[1]
sage: C3 == C4.dual_code()
True
```

This is consistent with Theorem 6.6.9 and Exercise 365 in [HP].

QuadraticResidueCodeOddPair(*n, F*)

Quadratic residue codes of a given odd prime length and base ring either don't exist at all or occur as 4-tuples - a pair of "odd-like" codes and a pair of "even-like" codes. If n 2 is prime then (Theorem 6.6.2 in [HP]) a QR code exists over GF(q) iff q is a quadratic residue mod n.

They are constructed as "odd-like" duadic codes associated the splitting (Q,N) mod n, where Q is the set of non-zero quadratic residues and N is the non-residues.

EXAMPLES:

```
sage: QuadraticResidueCodeOddPair(17,GF(13))
(Linear code of length 17, dimension 9 over Finite Field of size 13,
 Linear code of length 17, dimension 9 over Finite Field of size 13)
sage: QuadraticResidueCodeOddPair(17,GF(2))
(Linear code of length 17, dimension 9 over Finite Field of size 2,
 Linear code of length 17, dimension 9 over Finite Field of size 2)
sage: QuadraticResidueCodeOddPair(13,GF(9,"z"))
(Linear code of length 13, dimension 7 over Finite Field in z of size 3^2,
 Linear code of length 13, dimension 7 over Finite Field in z of size 3^2)
sage: C1 = QuadraticResidueCodeOddPair(17,GF(2))[1]
sage: C1x = C1.extended_code()
sage: C2 = QuadraticResidueCodeOddPair(17,GF(2))[0]
sage: C2x = C2.extended_code()
sage: C2x.spectrum(); C1x.spectrum()
[1, 0, 0, 0, 0, 0, 102, 0, 153, 0, 153, 0, 102, 0, 0, 0, 0, 0, 1]
[1, 0, 0, 0, 0, 0, 102, 0, 153, 0, 153, 0, 102, 0, 0, 0, 0, 0, 1]
sage: C2x == C1x.dual_code()
True
sage: C3 = QuadraticResidueCodeOddPair(7,GF(2))[0]
sage: C3x = C3.extended_code()
sage: C3x.spectrum()
[1, 0, 0, 0, 14, 0, 0, 0, 1]
sage: C3x.is_self_dual()
True
```

This is consistent with Theorem 6.6.14 in [HP].

RandomLinearCode(*n, k, F*)

The method used is to first construct a $k \times n$ matrix using Sage's random_element method for the MatrixSpace class. The construction is probabilistic but should only fail extremely rarely.

INPUT: Integers n,k, with nk1, and a finite field F

OUTPUT: Returns a "random" linear code with length n, dimension k over field F.

EXAMPLES:

```
sage: C = RandomLinearCode(30,15,GF(2))
sage: C
Linear code of length 30, dimension 15 over Finite Field of size 2
sage: C = RandomLinearCode(10,5,GF(4,'a'))
```

```
sage: C
Linear code of length 10, dimension 5 over Finite Field in a of size 2^2
```

AUTHORS:

- David Joyner (2007-05)

ReedSolomonCode(*n, k, F, pts=None*)

Given a finite field F of order q, let n and k be chosen such that $1 \leq k \leq n \leq q$. Pick n distinct elements of F, denoted $\{x_1, x_2, ..., x_n\}$. Then, the codewords are obtained by evaluating every polynomial in $F[x]$ of degree less than k at each x_i:

$$C = \{(f(x_1), f(x_2), ..., f(x_n)), f \in F[x], \deg(f) < k\}.$$

C is a $[n, k, n-k+1]$ code. (In particular, C is MDS.)

INPUT: n : the length k : the dimension F : the base ring pts : (optional) list of n points in F (if None then Sage picks n of them in the order given to the elements of F)

EXAMPLES:

```
sage: C = ReedSolomonCode(6,4,GF(7)); C
Linear code of length 6, dimension 4 over Finite Field of size 7
sage: C.minimum_distance()
3
sage: C = ReedSolomonCode(6,4,GF(8,"a")); C
Linear code of length 6, dimension 4 over Finite Field in a of size 2^3
sage: C.minimum_distance()
3
sage: F.<a> = GF(3^2,"a")
sage: pts = [0,1,a,a^2,2*a,2*a+1]
sage: len(Set(pts)) == 6 # to make sure there are no duplicates
True
sage: C = ReedSolomonCode(6,4,F,pts); C
Linear code of length 6, dimension 4 over Finite Field in a of size 3^2
sage: C.minimum_distance()
3
```

REFERENCES:

- [HP] W. C. Huffman, V. Pless, Fundamentals of Error-Correcting Codes, Cambridge Univ. Press, 2003.
- [W] http://en.wikipedia.org/wiki/Reed-Solomon

TernaryGolayCode()

TernaryGolayCode returns a ternary Golay code. This is a perfect [11,6,5] code. It is also equivalent to a cyclic code, with generator polynomial $g(x) = 2 + x^2 + 2x^3 + x^4 + x^5$.

EXAMPLES:

```
sage: C = TernaryGolayCode()
sage: C
Linear code of length 11, dimension 6 over Finite Field of size 3
sage: C.minimum_distance()
5
```

AUTHORS:

- David Joyner (2007-5)

ToricCode(*P, F*)

Let P denote a list of lattice points in $\mathbf{Z}^d$ and let T denote the set of all points in $(F^x)^d$ (ordered in some fixed way). Put $n = |T|$ and let k denote the dimension of the vector space of functions $V = \mathrm{Span}\{x^e \mid e \in P\}$. The associated toric code C is the evaluation code which is the image of the evaluation map

$$\mathrm{eval}_\mathrm{T} : V \to F^n,$$

where x^e is the multi-index notation ($x = (x_1, ..., x_d)$, $e = (e_1, ..., e_d)$, and $x^e = x_1^{e_1}...x_d^{e_d}$), where $eval_T(f(x)) = (f(t_1), ..., f(t_n))$, and where $T = \{t_1, ..., t_n\}$. This function returns the toric codes discussed in [J].

INPUT:

- `P` - all the integer lattice points in a polytope defining the toric variety.
- `F` - a finite field.

OUTPUT: Returns toric code with length n = , dimension k over field F.

EXAMPLES:

```
sage: C = ToricCode([[0,0],[1,0],[2,0],[0,1],[1,1]],GF(7))
sage: C
Linear code of length 36, dimension 5 over Finite Field of size 7
sage: C.minimum_distance()
24
sage: C = ToricCode([[-2,-2],[-1,-2],[-1,-1],[-1,0],[0,-1],[0,0],[0,1],[1,-1],[1,0]],GF(5))
sage: C
Linear code of length 16, dimension 9 over Finite Field of size 5
sage: C.minimum_distance()
6
sage: C = ToricCode([ [0,0],[1,1],[1,2],[1,3],[1,4],[2,1],[2,2],[2,3],[3,1],[3,2],[4,1]],GF(8,"a
sage: C
Linear code of length 49, dimension 11 over Finite Field in a of size 2^3
```

This is in fact a [49,11,28] code over GF(8). If you type next `C.minimum_distance()` and wait overnight (!), you should get 28.

AUTHOR:

- David Joyner (07-2006)

REFERENCES:

- [J] D. Joyner, Toric codes over finite fields, Applicable Algebra in Engineering, Communication and Computing, 15, (2004), p. 63-79

TrivialCode(*F, n*)

WalshCode(*m*)

Returns the binary Walsh code of length 2^m. The matrix of codewords correspond to a Hadamard matrix. This is a (constant rate) binary linear $[2^m, m, 2^{m-1}]$ code.

EXAMPLES:

```
sage: C = WalshCode(4); C
Linear code of length 16, dimension 4 over Finite Field of size 2
sage: C = WalshCode(3); C
Linear code of length 8, dimension 3 over Finite Field of size 2
sage: C.spectrum()
[1, 0, 0, 0, 7, 0, 0, 0]
```

REFERENCES:

- http://en.wikipedia.org/wiki/Hadamard_matrix
- http://en.wikipedia.org/wiki/Walsh_code

cyclotomic_cosets(*q, n, t=None*)

INPUT: q,n,t positive integers (or t=None) Some type-checking of inputs is performed.

OUTPUT: q-cyclotomic cosets mod n (or, if t is not None, the q-cyclotomic coset mod n containing t)

Let q, n be relatively print positive integers and let $A = q^{ZZ}$. The group A acts on ZZ/nZZ by multiplication. The orbits of this action are "cyclotomic cosets", or more precisely "q-cyclotomic cosets mod n". Sometimes the smallest element of the coset is called the "coset leader". The algorithm will always return the cosets as sorted lists of lists, so the coset leader will always be the first element in the list.

These cosets arise in the theory of duadic codes and minimal polynomials of finite fields. Fix a primitive element z of $GF(q^k)$. The minimal polynomial of z^s over $GF(q)$ is given by

$$M_s(x) = \prod_{i \in C_s} (x - z^i),$$

where C_s is the q-cyclotomic coset mod n containing s, $n = q^k - 1$.

EXAMPLES:

```
sage: cyclotomic_cosets(2,11)
[[0], [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]]
sage: cyclotomic_cosets(2,15)
[[0], [1, 2, 4, 8], [3, 6, 9, 12], [5, 10], [7, 11, 13, 14]]
sage: cyclotomic_cosets(2,15,5)
[5, 10]
sage: cyclotomic_cosets(3,16)
[[0], [1, 3, 9, 11], [2, 6], [4, 12], [5, 7, 13, 15], [8], [10, 14]]
sage: F.<z> = GF(2^4, "z")
sage: P.<x> = PolynomialRing(F,"x")
sage: a = z^5
sage: a.minimal_polynomial()
x^2 + x + 1
sage: prod([x-z^i for i in [5, 10]])
x^2 + x + 1
sage: cyclotomic_cosets(3,2,0)
[0]
sage: cyclotomic_cosets(3,2,1)
[1]
sage: cyclotomic_cosets(3,2,2)
[0]
```

This last output looks strange but is correct, since the elements of the cosets are in ZZ/nZZ and 2 = 0 in ZZ/2ZZ.

is_a_splitting(*S1, S2, n*)

INPUT: S1, S2 are disjoint sublists partitioning [1, 2, ..., n-1] n1 is an integer

OUTPUT: a, b where a is True or False, depending on whether S1, S2 form a "splitting" of n (ie, if there is a b1 such that b*S1=S2 (point-wise multiplication mod n), and b is a splitting (if a = True) or 0 (if a = False)

Splittings are useful for computing idempotents in the quotient ring $Q = GF(q)[x]/(x^n - 1)$. For

EXAMPLES:

```
sage: from sage.coding.code_constructions import is_a_splitting
sage: n = 11; q = 3
sage: C = cyclotomic_cosets(q,n); C
[[0], [1, 3, 4, 5, 9], [2, 6, 7, 8, 10]]
sage: S1 = C[1]
```

```
sage: S2 = C[2]
sage: is_a_splitting(S1,S2,11)
(True, 2)
sage: F = GF(q)
sage: P.<x> = PolynomialRing(F,"x")
sage: I = Ideal(P,[x^n-1])
sage: Q.<x> = QuotientRing(P,I)
sage: i1 = -sum([x^i for i in S1]); i1
2*x^9 + 2*x^5 + 2*x^4 + 2*x^3 + 2*x
sage: i2 = -sum([x^i for i in S2]); i2
2*x^10 + 2*x^8 + 2*x^7 + 2*x^6 + 2*x^2
sage: i1^2 == i1
True
sage: i2^2 == i2
True
sage: (1-i1)^2 == 1-i1
True
sage: (1-i2)^2 == 1-i2
True
```

We return to dealing with polynomials (rather than elements of quotient rings), so we can construct cyclic codes:

```
sage: P.<x> = PolynomialRing(F,"x")
sage: i1 = -sum([x^i for i in S1])
sage: i2 = -sum([x^i for i in S2])
sage: i1_sqrd = (i1^2).quo_rem(x^n-1)[1]
sage: i1_sqrd  == i1
True
sage: i2_sqrd = (i2^2).quo_rem(x^n-1)[1]
sage: i2_sqrd  == i2
True
sage: C1 = CyclicCodeFromGeneratingPolynomial(n,i1)
sage: C2 = CyclicCodeFromGeneratingPolynomial(n,1-i2)
sage: C1.dual_code() == C2
True
```

This is a special case of Theorem 6.4.3 in [HP].

lift2smallest_field(*a*)

INPUT: a is an element of a finite field GF(q)

OUTPUT: the element b of the smallest subfield F of GF(q) for which F(b)=a.

EXAMPLES:

```
sage: from sage.coding.code_constructions import lift2smallest_field
sage: FF.<z> = GF(3^4,"z")
sage: a = z^10
sage: lift2smallest_field(a)
(2*z + 1, Finite Field in z of size 3^2)
sage: a = z^40
sage: lift2smallest_field(a)
(2, Finite Field of size 3)
```

AUTHORS:

- John Cremona

lift2smallest_field2(*a*)

INPUT: a is an element of a finite field GF(q) OUTPUT: the element b of the smallest subfield F of GF(q) for which F(b)=a.

EXAMPLES:

```
sage: from sage.coding.code_constructions import lift2smallest_field2
sage: FF.<z> = GF(3^4,"z")
sage: a = z^40
sage: lift2smallest_field2(a)
(2, Finite Field of size 3)
sage: FF.<z> = GF(2^4,"z")
sage: a = z^15
sage: lift2smallest_field2(a)
(1, Finite Field of size 2)
```

Warning: Since coercion (the FF(b) step) has a bug in it, this *only works* in the case when you *know* F is a prime field.

AUTHORS:

- David Joyner

permutation_action(*g*, *v*)

Returns permutation of rows g*v. Works on lists, matrices, sequences and vectors (by permuting coordinates). The code requires switching from i to i+1 (and back again) since the SymmetricGroup is, by convention, the symmetric group on the "letters" 1, 2, ..., n (not 0, 1, ..., n-1).

EXAMPLES:

```
sage: V = VectorSpace(GF(3),5)
sage: v = V([0,1,2,0,1])
sage: G = SymmetricGroup(5)
sage: g = G([(1,2,3)])
sage: permutation_action(g,v)
(1, 2, 0, 0, 1)
sage: g = G([()])
sage: permutation_action(g,v)
(0, 1, 2, 0, 1)
sage: g = G([(1,2,3,4,5)])
sage: permutation_action(g,v)
(1, 2, 0, 1, 0)
sage: L = Sequence([1,2,3,4,5])
sage: permutation_action(g,L)
[2, 3, 4, 5, 1]
sage: MS = MatrixSpace(GF(3),3,7)
sage: A = MS([[1,0,0,0,1,1,0],[0,1,0,1,0,1,0],[0,0,0,0,0,0,1]])
sage: S5 = SymmetricGroup(5)
sage: g = S5([(1,2,3)])
sage: A
[1 0 0 0 1 1 0]
[0 1 0 1 0 1 0]
[0 0 0 0 0 0 1]
sage: permutation_action(g,A)
[0 1 0 1 0 1 0]
[0 0 0 0 0 0 1]
[1 0 0 0 1 1 0]
```

It also works on lists and is a "left action":

```
sage: v = [0,1,2,0,1]
sage: G = SymmetricGroup(5)
sage: g = G([(1,2,3)])
sage: gv = permutation_action(g,v); gv
[1, 2, 0, 0, 1]
sage: permutation_action(g,v) == g(v)
True
sage: h = G([(3,4)])
sage: gv = permutation_action(g,v)
sage: hgv = permutation_action(h,gv)
sage: hgv == permutation_action(h*g,v)
True
```

AUTHORS:

- David Joyner, licensed under the GPL v2 or greater.

walsh_matrix(*m0*)

This is the generator matrix of a Walsh code. The matrix of codewords correspond to a Hadamard matrix.

EXAMPLES:

```
sage: walsh_matrix(2)
[0 0 1 1]
[0 1 0 1]
sage: walsh_matrix(3)
[0 0 0 0 1 1 1 1]
[0 0 1 1 0 0 1 1]
[0 1 0 1 0 1 0 1]
sage: C = LinearCode(walsh_matrix(4)); C
Linear code of length 16, dimension 4 over Finite Field of size 2
sage: C.spectrum()
[1, 0, 0, 0, 0, 0, 0, 0, 15, 0, 0, 0, 0, 0, 0, 0, 0]
```

This last code has minimum distance 8.

REFERENCES:

- http://en.wikipedia.org/wiki/Hadamard_matrix

40.3 Binary self-dual codes

This module implements functions useful for studying binary self-dual codes. The main function is `self_dual_codes_binary`, which is a case-by-case list of entries, each represented by a Python dictionary.

Format of each entry: a Python dictionary with keys "order autgp", "spectrum", "code", "Comment", "Type", where

- "code" - a sd code C of length n, dim n/2, over GF(2)
- "order autgp" - order of the permutation automorphism group of C
- "Type" - the type of C (which can be "I" or "II", in the binary case)
- "spectrum" - the spectrum [A0,A1,...,An]
- "Comment" - possibly an empty string.

Python dictionaries were used since they seemed to be both human-readable and allow others to update the database easiest.

- The following double for loop can be time-consuming but should be run once in awhile for testing purposes. It should only print True and have no trace-back errors:

```
for n in [4,6,8,10,12,14,16,18,20,22]:
     C = self_dual_codes_binary(n); m = len(C.keys())
     for i in range(m):
         C0 = C["%s"%n]["%s"%i]["code"]
         print n, '  ',i, '    ',C["%s"%n]["%s"%i]["spectrum"] == C0.spectrum()
         print C0 == C0.dual_code()
         G = C0.automorphism_group_binary_code()
         print C["%s"%n]["%s"%i]["order autgp"] == G.order()
```

- To check if the "Riemann hypothesis" holds, run the following code:

```
R = PolynomialRing(CC,"T")
T = R.gen()
for n in [4,6,8,10,12,14,16,18,20,22]:
     C = self_dual_codes_binary(n); m = len(C["%s"%n].keys())
     for i in range(m):
         C0 = C["%s"%n]["%s"%i]["code"]
         if C0.minimum_distance()>2:
             f = R(C0.sd_zeta_polynomial())
             print n,i,[z[0].abs() for z in f.roots()]
```

You should get lists of numbers equal to 0.707106781186548.

Here's a rather naive construction of self-dual codes in the binary case:

For even m, let A_m denote the mxm matrix over GF(2) given by adding the all 1's matrix to the identity matrix (in `MatrixSpace(GF(2),m,m)` of course). If M_1, ..., M_r are square matrices, let $diag(M_1, M_2, ..., M_r)$ denote the"block diagonal" matrix with the M_i 's on the diagonal and 0's elsewhere. Let $C(m_1, ..., m_r, s)$ denote the linear code with generator matrix having block form $G = (I, A)$, where $A = diag(A_{m_1}, A_{m_2}, ..., A_{m_r}, I_s)$, for some (even) m_i 's and s, where $m_1 + m_2 + ... + m_r + s = n/2$. Note: Such codes $C(m_1, ..., m_r, s)$ are SD.

SD codes not of this form will be called (for the purpose of documenting the code below) "exceptional". Except when n is "small", most sd codes are exceptional (based on a counting argument and table 9.1 in the Huffman+Pless [HP], page 347).

AUTHORS:

- David Joyner (2007-08-11)

REFERENCES:

- [HP] W. C. Huffman, V. Pless, Fundamentals of Error-Correcting Codes, Cambridge Univ. Press, 2003.
- [P] V. Pless, "A classification of self-orthogonal codes over GF(2)", Discrete Math 3 (1972) 209-246.

I2(n)

MS(n)

MS2(n)

matA(n)

matId(n)

self_dual_codes_binary(*n*)

Returns the dictionary of inequivalent sd codes of length n.

For n=4 even, returns the sd codes of a given length, up to (perm) equivalence, the (perm) aut gp, and the type.

The number of inequiv "diagonal" sd binary codes in the database of length n is ("diagonal" is defined by the conjecture above) is the same as the restricted partition number of n, where only integers from the set 1,4,6,8,... are allowed. This is the coeff of x^n in the series expansion $(1-x)^{-1}\prod_{2^\infty(1-x^{2j})^{-1}}$. Typing the command f = (1-x)(-1)*prod([(1-x(2*j))(-1) for j in range(2,18)]) into Sage, we obtain for the coeffs of x^4, x^6, ... [1, 1, 2, 2, 3, 3, 5, 5, 7, 7, 11, 11, 15, 15, 22, 22, 30, 30, 42, 42, 56, 56, 77, 77, 101, 101, 135, 135, 176, 176, 231] These numbers grow too slowly to account for all the sd codes (see Huffman+Pless' Table 9.1, referenced above). In fact, in Table 9.10 of [HP], the number B_n of inequivalent sd binary codes of length n is given:

```
n   2 4 6 8 10 12 14 16 18 20 22 24  26  28  30
B_n 1 1 1 2  2  3  4  7  9 16 25 55 103 261 731
```

According to http://www.research.att.com/~njas/sequences/A003179, the next 2 entries are: 3295, 24147.

EXAMPLES:

```
sage: C = self_dual_codes_binary(10)
sage: C["10"]["0"]["code"] == C["10"]["0"]["code"].dual_code()
True
sage: C["10"]["1"]["code"] == C["10"]["1"]["code"].dual_code()
True
sage: len(C["10"].keys()) # number of inequiv sd codes of length 10
2
sage: C = self_dual_codes_binary(12)
sage: C["12"]["0"]["code"] == C["12"]["0"]["code"].dual_code()
True
sage: C["12"]["1"]["code"] == C["12"]["1"]["code"].dual_code()
True
sage: C["12"]["2"]["code"] == C["12"]["2"]["code"].dual_code()
True
```

40.4 Bounds for Parameters of Codes

AUTHORS:

- David Joyner (2006-07): initial implementation.
- William Stein (2006-07): minor editing of docs and code (fixed bug in elias_bound_asymp)
- David Joyner (2006-07): fixed dimension_upper_bound to return an integer, added example to elias_bound_asymp.

This module provided some upper and lower bounds for the parameters of codes.

Let F be a finite field (we denote the finite field with q elements by $\mathbf{F}_q$). A subset C of $V = F^n$ is called a code of length n. A subspace of V (with the standard basis) is called a linear code of length n. If its dimension is denoted k then we typically store a basis of C as a $k \times n$ matrix (the rows are the basis vectors). If $F = \mathbf{F}_2$ then C is called a binary code. If F has q elements then C is called a q-ary code. The elements of a code C are called codewords. The information rate of C is

$$R = \frac{\log_q |C|}{n},$$

where $|C|$ denotes the number of elements of C. If $\mathbf{v} = (v_1, v_2, ..., v_n)$, $\mathbf{w} = (w_1, w_2, ..., w_n)$ are vectors in $V = F^n$ then we define

$$d(\mathbf{v}, \mathbf{w}) = |\{i \mid 1 \leq i \leq n,\ v_i \neq w_i\}|$$

to be the Hamming distance between $\mathbf{v}$ and $\mathbf{w}$. The function $d : V \times V \to \mathbf{N}$ is called the Hamming metric. The weight of a vector (in the Hamming metric) is $d(\mathbf{v}, \mathbf{0})$. The minimum distance of a linear code is the smallest non-zero weight of a codeword in C. The relatively minimum distance is denoted

$$\delta = d/n.$$

A linear code with length n, dimension k, and minimum distance d is called an $[n, k, d]_q$-code and n, k, d are called its parameters. A (not necessarily linear) code C with length n, size $M = |C|$, and minimum distance d is called an $(n, M, d)_q$-code (using parentheses instead of square brackets). Of course, $k = \log_q(M)$ for linear codes.

What is the "best" code of a given length? Let F be a finite field with q elements. Let $A_q(n, d)$ denote the largest M such that there exists a (n, M, d) code in F^n. Let $B_q(n, d)$ (also denoted $A_q^{lin}(n, d)$) denote the largest k such that there exists a $[n, k, d]$ code in F^n. (Of course, $A_q(n, d) \geq B_q(n, d)$.) Determining $A_q(n, d)$ and $B_q(n, d)$ is one of the main problems in the theory of error-correcting codes.

These quantities related to solving a generalization of the childhood game of "20 questions".

GAME: Player 1 secretly chooses a number from 1 to M (M is large but fixed). Player 2 asks a series of "yes/no questions" in an attempt to determine that number. Player 1 may lie at most e times ($e \geq 0$ is fixed). What is the minimum number of "yes/no questions" Player 2 must ask to (always) be able to correctly determine the number Player 1 chose?

If feedback is not allowed (the only situation considered here), call this minimum number $g(M, e)$.

Lemma: For fixed e and M, $g(M, e)$ is the smallest n such that $A_2(n, 2e + 1) \geq M$.

Thus, solving the solving a generalization of the game of "20 questions" is equivalent to determining $A_2(n, d)$! Using Sage, you can determine the best known estimates for this number in 2 ways:

(1) Indirectly, using minimum_distance_lower_bound(n,k,F) and minimum_distance_upper_bound(n,k,F) (both of which which connect to the internet using Steven Sivek's linear_code_bound(q,n,k)) (2) codesize_upper_bound(n,d,q), dimension_upper_bound(n,d,q), which use GUAVA's UpperBound(n, d, q)

This module implements:

- codesize_upper_bound(n,d,q), for the best known (as of May, 2006) upper bound A(n,d) for the size of a code of length n, minimum distance d over a field of size q.
- dimension_upper_bound(n,d,q), an upper bound $B(n, d) = B_q(n, d)$ for the dimension of a linear code of length n, minimum distance d over a field of size q.
- gilbert_lower_bound(n,q,d), a lower bound for number of elements in the largest code of min distance d in $\mathbf{F}_q^n$.
- gv_info_rate(n,delta,q), $log_q(GLB)/n$, where GLB is the Gilbert lower bound and delta = d/n.
- gv_bound_asymp(delta,q), asymptotic analog of Gilbert lower bound.
- plotkin_upper_bound(n,q,d)
- plotkin_bound_asymp(delta,q), asymptotic analog of Plotkin bound.
- griesmer_upper_bound(n,q,d)
- elias_upper_bound(n,q,d)
- elias_bound_asymp(delta,q), asymptotic analog of Elias bound.
- hamming_upper_bound(n,q,d)

- hamming_bound_asymp(delta,q), asymptotic analog of Hamming bound.
- singleton_upper_bound(n,q,d)
- singleton_bound_asymp(delta,q), asymptotic analog of Singleton bound.
- mrrw1_bound_asymp(delta,q), "first" asymptotic McEliese-Rumsey-Rodemich-Welsh bound for the information rate.

PROBLEM: In this module we shall typically either (a) seek bounds on k, given n, d, q, (b) seek bounds on R, delta, q (assuming n is "infinity").

TODO:

- Johnson bounds for binary codes.
- mrrw2_bound_asymp(delta,q), "second" asymptotic McEliese-Rumsey-Rodemich-Welsh bound for the information rate.

REFERENCES:

- C. Huffman, V. Pless, Fundamentals of error-correcting codes, Cambridge Univ. Press, 2003.

codesize_upper_bound(*n, d, q*)

Returns the best known upper bound $A(n, d) = A_q(n, d)$ for the size of a code of length n, minimum distance d over a field of size q. The function first checks for trivial cases (like d=1 or n=d), and if the value is in the built-in table. Then it calculates the minimum value of the upper bound using the methods of Singleton, Hamming, Johnson, Plotkin and Elias. If the code is binary, $A(n, 2\ell - 1) = A(n + 1, 2\ell)$, so the function takes the minimum of the values obtained from all methods for the parameters $(n, 2\ell - 1)$ and $(n + 1, 2\ell)$.

Wraps GUAVA's UpperBound(n, d, q).

EXAMPLES:

```
sage: codesize_upper_bound(10,3,2)
85
```

dimension_upper_bound(*n, d, q*)

Returns an upper bound $B(n, d) = B_q(n, d)$ for the dimension of a linear code of length n, minimum distance d over a field of size q.

EXAMPLES:

```
sage: dimension_upper_bound(10,3,2)
6
```

elias_bound_asymp(*delta, q*)

Computes the asymptotic Elias bound for the information rate, provided $0 < \delta 1 - 1/q$.

EXAMPLES:

```
sage: elias_bound_asymp(1/4,2)
0.39912396330...
```

elias_upper_bound(*n, q, d*)

Returns the Elias upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundElias.

EXAMPLES:

```
sage: elias_upper_bound(10,2,3)
232
```

entropy(*x, q*)
Computes the entropy on the q-ary symmetric channel.

gilbert_lower_bound(*n, q, d*)
Returns lower bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$.

EXAMPLES:

```
sage: gilbert_lower_bound(10,2,3)
128/7
```

griesmer_upper_bound(*n, q, d*)
Returns the Griesmer upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundGriesmer.

EXAMPLES:

```
sage: griesmer_upper_bound(10,2,3)
128
```

gv_bound_asymp(*delta, q*)
Computes the asymptotic GV bound for the information rate, R.

EXAMPLES:

```
sage: f = lambda x: gv_bound_asymp(x,2)
sage: plot(f,0,1)
```

gv_info_rate(*n, delta, q*)
GV lower bound for information rate of a q-ary code of length n minimum distance delta*n

EXAMPLES:

```
sage: RDF(gv_info_rate(100,1/4,3))
0.367049926083
```

hamming_bound_asymp(*delta, q*)
Computes the asymptotic Hamming bound for the information rate.

EXAMPLES:

```
sage: f = lambda x: hamming_bound_asymp(x,2)
sage: plot(f,0,1)
```

hamming_upper_bound(*n, q, d*)
Returns the Hamming upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundHamming.

The Hamming bound (also known as the sphere packing bound) returns an upper bound on the size of a code of length n, minimum distance d, over a field of size q. The Hamming bound is obtained by dividing the contents of the entire space $\mathbf{F}_q^n$ by the contents of a ball with radius floor((d-1)/2). As all these balls are disjoint, they can never contain more than the whole vector space.

$$M \leq \frac{q^n}{V(n,e)},$$

where M is the maximum number of codewords and $V(n,e)$ is equal to the contents of a ball of radius e. This bound is useful for small values of d. Codes for which equality holds are called perfect.

EXAMPLES:

```
sage: hamming_upper_bound(10,2,3)
93
```

mrrw1_bound_asymp(*delta, q*)

Computes the first asymptotic McEliese-Rumsey-Rodemich-Welsh bound for the information rate, provided $0 < \delta < 1 - 1/q$.

EXAMPLES:

```
sage: mrrw1_bound_asymp(1/4,2)
0.354578902665
```

plotkin_bound_asymp(*delta, q*)

Computes the asymptotic Plotkin bound for the information rate, provided $0 < \delta < 1 - 1/q$.

EXAMPLES:

```
sage: plotkin_bound_asymp(1/4,2)
1/2
```

plotkin_upper_bound(*n, q, d*)

Returns Plotkin upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundPlotkin.

EXAMPLES:

```
sage: plotkin_upper_bound(10,2,3)
192
```

singleton_bound_asymp(*delta, q*)

Computes the asymptotic Singleton bound for the information rate.

EXAMPLES:

```
sage: f = lambda x: singleton_bound_asymp(x,2)
sage: plot(f,0,1)
```

singleton_upper_bound(*n, q, d*)

Returns the Singleton upper bound for number of elements in the largest code of minimum distance d in $\mathbf{F}_q^n$. Wraps GAP's UpperBoundSingleton.

This bound is based on the shortening of codes. By shortening an (n, M, d) code d-1 times, an $(n-d+1, M, 1)$ code results, with $M \leq q^n - d + 1$. Thus

$$M \leq q^{n-d+1}.$$

Codes that meet this bound are called maximum distance separable (MDS).

EXAMPLES:

```
sage: singleton_upper_bound(10,2,3)
256
```

volume_hamming(*n, q, r*)

Returns number of elements in a Hamming ball of radius r in $\mathbf{F}_q^n$.

EXAMPLES:

```
sage: volume_hamming(10,2,3)
176
```

CHAPTER

FORTYONE

ARITHMETIC SUBGROUPS OF $\mathrm{SL}_2(\mathbf{Z})$

This chapter describes the basic functionality for finite index subgroups of the modular group $\mathrm{SL}_2(\mathbf{Z})$.

41.1 Arithmetic subgroups (finite index subgroups of $\mathrm{SL}_2(\mathrm{Z})$)

class ArithmeticSubgroup()

Base class for arithmetic subgroups of $\mathrm{SL}_2(\mathbf{Z})$. Not intended to be used directly, but still includes quite a few general-purpose routines which compute data about an arithmetic subgroup assuming that it has a working element testing routine.

are_equivalent(*x*, *y*)

Determine whether x and y are equivalent by an element of self, i.e. whether or not there exists an element g of self such that $g \cdot x = y$.

NOTE: This function should be overridden by all subclasses.

EXAMPLES:

```
sage: sage.modular.arithgroup.congroup_generic.CongruenceSubgroup(5).are_equivalent(0, 0)
...
NotImplementedError
```

are_equivalent_cusps(*x*, *y*, *trans=False*)

Test whether or not cusps x and y are equivalent modulo self. If self has a reduce_cusp() method, use that; otherwise do a slow explicit test.

If trans = False, returns True or False. If trans = True, then return either False or an element of self mapping x onto y.

EXAMPLE:

```
sage: Gamma0(7).are_equivalent_cusps(Cusp(1/3), Cusp(0), trans=True)
[  3  -1]
[-14   5]
sage: Gamma0(7).are_equivalent_cusps(Cusp(1/3), Cusp(1/7))
False
```

as_permutation_group()

Return a representation of this arithmetic subgroup in terms of the permutation action of SL2Z on the cosets of G.

At present this is only implemented for even subgroups.

EXAMPLE:

```
sage: Gamma0(3).as_permutation_group()
Arithmetic subgroup corresponding to permutations L=(2,3,4), R=(1,3,4)
```

coset_reps(*G=None*)

Return coset representatives for self \ G, where G is another arithmetic subgroup that contains self. If G = None, default to G = SL2Z.

For generic arithmetic subgroups G this is carried out by Todd-Coxeter enumeration; here G is treated as a black box, implementing nothing but membership testing.

EXAMPLES:

```
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup().coset_reps()
...
NotImplementedError
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup.coset_reps(Gamma0(3))
[[1 0]
[0 1], [ 0 -1]
[ 1  0], [ 0 -1]
[ 1  1], [ 0 -1]
[ 1  2]]
```

cusp_data(*c*)

Return a triple (g, w, t) where g is an element of self generating the stabiliser of the given cusp, w is the width of the cusp, and t is 1 if the cusp is regular and -1 if not.

EXAMPLES:

```
sage: Gamma1(4).cusp_data(Cusps(1/2))
([ 1 -1] [ 4 -3], 1, -1)
```

cusp_width(*c*)

Return the width of the orbit of cusps represented by c.

EXAMPLES:

```
sage: Gamma0(11).cusp_width(Cusps(oo))
1
sage: Gamma0(11).cusp_width(0)
11
sage: [Gamma0(100).cusp_width(c) for c in Gamma0(100).cusps()]
[100, 1, 4, 1, 1, 1, 4, 25, 1, 1, 4, 1, 25, 4, 1, 4, 1, 1]
```

cusps(*algorithm='default'*)

Return a sorted list of inequivalent cusps for self, i.e. a set of representatives for the orbits of self on $\mathbb{P}^1(\mathbf{Q})$. These should be returned in a reduced form where this makes sense.

INPUTS: **algorithm – which algorithm to use to compute the cusps** of self. 'default' finds representatives for a known complete set of cusps. 'modsym' computes the boundary map on the space of weight two modular symbols associated to self, which finds the cusps for self in the process.

EXAMPLES:

```
sage: Gamma0(36).cusps()
[0, 1/18, 1/12, 1/9, 1/6, 1/4, 1/3, 5/12, 1/2, 2/3, 5/6, Infinity]
sage: Gamma0(36).cusps(algorithm='modsym') == Gamma0(36).cusps()
True
sage: GammaH(36, [19,29]).cusps() == Gamma0(36).cusps()
True
sage: Gamma0(1).cusps()
[Infinity]
```

dimension_cusp_forms(*k=2*)

Return the dimension of the space of weight k cusp forms for this group. This is given by a standard formula in terms of k and various invariants of the group; see Diamond + Shurman, "A First Course in Modular Forms", section 3.5 and 3.6. If k is not given, default to k = 2.

For dimensions of spaces of cusp forms with character for Gamma1, use the standalone function dimension_cusp_forms().

For weight 1 cusp forms this function only works in cases where one can prove solely in terms of Riemann-Roch theory that there aren't any cusp forms (i.e. when the number of regular cusps is strictly greater than the degree of the canonical divisor). Otherwise a NotImplementedError is raised.

EXAMPLE:

```
sage: Gamma1(31).dimension_cusp_forms(2)
26
sage: Gamma1(3).dimension_cusp_forms(1)
0
sage: Gamma1(4).dimension_cusp_forms(1) # irregular cusp
0
sage: Gamma1(31).dimension_cusp_forms(1)
...
NotImplementedError: Computation of dimensions of weight 1 cusp forms spaces not implemented
```

dimension_eis(*k=2*)

Return the dimension of the space of weight k Eisenstein series for this group, which is a subspace of the space of modular forms complementary to the space of cusp forms.

INPUT:

- k - an integer (default 2).

EXAMPLES:

```
sage: GammaH(33,[2]).dimension_eis()
7
sage: GammaH(33,[2]).dimension_eis(3)
0
sage: GammaH(33, [2,5]).dimension_eis(2)
3
sage: GammaH(33, [4]).dimension_eis(1)
4
```

dimension_modular_forms(*k=2*)

Return the dimension of the space of weight k modular forms for this group. This is given by a standard formula in terms of k and various invariants of the group; see Diamond + Shurman, "A First Course in Modular Forms", section 3.5 and 3.6. If k is not given, defaults to k = 2.

For dimensions of spaces of modular forms with character for Gamma1, use the standalone function dimension_modular_forms().

For weight 1 modular forms this function only works in cases where one can prove solely in terms of Riemann-Roch theory that there aren't any cusp forms (i.e. when the number of regular cusps is strictly greater than the degree of the canonical divisor). Otherwise a NotImplementedError is raised.

EXAMPLE:

```
sage: Gamma1(31).dimension_modular_forms(2)
55
sage: Gamma1(3).dimension_modular_forms(1)
1
sage: Gamma1(4).dimension_modular_forms(1) # irregular cusp
1
sage: Gamma1(31).dimension_modular_forms(1)
...
NotImplementedError: Computation of dimensions of weight 1 cusp forms spaces not implemented
```

gen(*i*)

Return the i-th generator of self, i.e. the i-th element of the tuple self.gens().

EXAMPLES:

```
sage: SL2Z.gen(1)
[1 1]
[0 1]
```

generalised_level()

Return the generalised level of self, i.e. the least common multiple of the widths of all cusps. Wohlfart's theorem tells us that this is equal to the (conventional) level of self when self is a congruence subgroup.

EXAMPLE:

```
sage: Gamma0(18).generalised_level()
18
sage: sage.modular.arithgroup.congroup_generic.CongruenceSubgroup(5).index()
...
NotImplementedError
```

generators()

Return generators for this congruence subgroup.

This is carried out using an "inverse Todd-Coxeter" algorithm. A Cython version of this for the special case of Γ_0 and Γ_1 is implemented in the function `congroup_pyx.generators_helper`. See the documentation there for further details. Here we are assuming far less about the group, so the computation is exceptionally slow!

EXAMPLE:

```
sage: Gamma(2).generators()
[[1 2]
[0 1],
[-1  0]
[ 0 -1],
[-1  0]
[ 0 -1],
[ 1  0]
[-2  1],
[-1  2]
[-2  3],
[-1  0]
[ 2 -1],
[1 0]
[2 1]]
```

gens()

Return a tuple of generators for this congruence subgroup.

The generators need not be minimal.

EXAMPLES:

```
sage: SL2Z.gens()
([ 0 -1]
[ 1  0], [1 1]
[0 1])
```

genus()

Return the genus of the modular curve of self.

EXAMPLES:

```
sage: Gamma1(5).genus()
0
sage: Gamma1(31).genus()
26
sage: Gamma1(157).genus() == dimension_cusp_forms(Gamma1(157), 2)
```

```
True
sage: GammaH(7, [2]).genus()
0
sage: [Gamma0(n).genus() for n in [1..23]]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 1, 0, 1, 1, 1, 2, 2]
sage: [n for n in [1..200] if Gamma0(n).genus() == 1]
[11, 14, 15, 17, 19, 20, 21, 24, 27, 32, 36, 49]
```

index()

Return the index of self in the full modular group.

EXAMPLES:

```
sage: Gamma0(17).index()
18
sage: sage.modular.arithgroup.congroup_generic.CongruenceSubgroup(5).index()
...
NotImplementedError
```

is_abelian()

Return True if this arithmetic subgroup is abelian.

Since arithmetic subgroups are always nonabelian, this always returns False.

EXAMPLES:

```
sage: SL2Z.is_abelian()
False
sage: Gamma0(3).is_abelian()
False
sage: Gamma1(12).is_abelian()
False
sage: GammaH(4, [2]).is_abelian()
False
```

is_congruence()

Return True if self is a congruence subgroup.

EXAMPLE:

```
sage: Gamma0(5).is_congruence()
True
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup().is_congruence()
...
NotImplementedError
```

is_even()

Return True precisely if this subgroup contains the matrix -1.

EXAMPLES:

```
sage: SL2Z.is_even()
True
sage: Gamma0(20).is_even()
True
sage: Gamma1(5).is_even()
False
sage: GammaH(11, [3]).is_even()
False
```

is_finite()

Return True if this arithmetic subgroup is finite.

Since arithmetic subgroups are always infinite, this always returns False.

EXAMPLES:

```
sage: SL2Z.is_finite()
False
sage: Gamma0(3).is_finite()
False
sage: Gamma1(12).is_finite()
False
sage: GammaH(4, [2]).is_finite()
False
```

is_normal()

Return True precisely if this subgroup is a normal subgroup of SL2Z.

EXAMPLES:

```
sage: Gamma(3).is_normal()
True
sage: Gamma1(3).is_normal()
False
```

is_odd()

Return True precisely if this subgroup does not contain the matrix -1.

EXAMPLES:

```
sage: SL2Z.is_odd()
False
sage: Gamma0(20).is_odd()
False
sage: Gamma1(5).is_odd()
True
sage: GammaH(11, [3]).is_odd()
True
```

is_regular_cusp(*c*)

Return True if the orbit of the given cusp is a regular cusp for self, otherwise False. This is automatically true if -1 is in self.

EXAMPLES:

```
sage: Gamma1(4).is_regular_cusp(Cusps(1/2))
False
sage: Gamma1(4).is_regular_cusp(Cusps(oo))
True
```

is_subgroup(*right*)

Return True if self is a subgroup of right, and False otherwise. For generic arithmetic subgroups this is done by the absurdly slow algorithm of checking all of the generators of self to see if they are in right.

EXAMPLES:

```
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup().is_subgroup(SL2Z)
...
NotImplementedError
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup.is_subgroup(Gamma1(18),
True
```

ncusps()

Return the number of cusps of this arithmetic subgroup. This is provided as a separate function since for dimension formulae in even weight all we need to know is the number of cusps, and this can be calculated very quickly, while enumerating all cusps is much slower.

EXAMPLES:

```
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup.ncusps(Gamma0(7))
2
```

ngens()

Return the number of generators for this arithmetic subgroup.

This need not be the minimal number of generators of self.

EXAMPLES:

```
sage: Gamma0(22).ngens()
42
sage: Gamma1(14).ngens()
219
sage: GammaH(11, [3]).ngens()
31
sage: SL2Z.ngens()
2
```

nirregcusps()

Return the number of cusps of self that are "irregular", i.e. their stabiliser can only be generated by elements with both eigenvalues -1 rather than +1. If the group contains -1, every cusp is clearly regular.

EXAMPLES:

```
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup.nirregcusps(Gamma1(4))
1
```

nregcusps()

Return the number of cusps of self that are "regular", i.e. their stabiliser has a generator with both eigenvalues +1 rather than -1. If the group contains -1, every cusp is clearly regular.

EXAMPLES:

```
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup.nregcusps(Gamma1(4))
2
```

nu2()

Return the number of orbits of elliptic points of order 2 for this arithmetic subgroup.

EXAMPLES:

```
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup().nu2()
...
NotImplementedError
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup.nu2(Gamma0(1105)) == 8
True
```

nu3()

Return the number of orbits of elliptic points of order 3 for this arithmetic subgroup.

EXAMPLES:

```
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup().nu3()
...
NotImplementedError
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup.nu3(Gamma0(1729)) == 8
True
```

We test that a bug in handling of subgroups not containing -1 is fixed:

```
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup.nu3(GammaH(7, [2]))
2
```

order()

Return the number of elements in this arithmetic subgroup.

Since arithmetic subgroups are always infinite, this always returns infinity.

EXAMPLES:

```
sage: SL2Z.order()
+Infinity
sage: Gamma0(5).order()
+Infinity
sage: Gamma1(2).order()
+Infinity
sage: GammaH(12, [5]).order()
+Infinity
```

projective_index()

Return the index of the image of self in $\mathrm{PSL}_2(\mathbf{Z})$. This is equal to the index of self if self contains -1, and half of this otherwise.

This is equal to the degree of the natural map from the modular curve of self to the j-line.

EXAMPLE:

```
sage: Gamma0(5).projective_index()
6
sage: Gamma1(5).projective_index()
12
```

reduce_cusp(c)

Given a cusp $c \in \mathbb{P}^1(\mathbf{Q})$, return the unique reduced cusp equivalent to c under the action of self, where a reduced cusp is an element $\frac{r}{s}$ with r,s coprime integers, s as small as possible, and r as small as possible for that s.

NOTE: This function should be overridden by all subclasses.

EXAMPLES:

```
sage: sage.modular.arithgroup.arithgroup_generic.ArithmeticSubgroup().reduce_cusp(1/4)
...
NotImplementedError
```

todd_coxeter()

Compute coset representatives for self \ G via Todd-Coxeter enumeration. If G = None, default to G = SL2Z. Also computes generators for G at the same time. Return value is a tuple (list of coset reps, list of generators).

This is *extremely* slow in general.

EXAMPLE:

sage: Gamma0(3).todd_coxeter() ([[1 0] [0 1], [0 -1] [1 0], [0 -1] [1 1], [0 -1] [1 2]], [[1 1] [0 1], [-1 0] [0 -1], [1 0] [-3 1], [-2 -1] [3 1], [-1 -1] [3 2]])

is_ArithmeticSubgroup(x)

Return True if x is of type ArithmeticSubgroup.

EXAMPLE:

```
sage: from sage.modular.arithgroup.all import is_ArithmeticSubgroup
sage: is_ArithmeticSubgroup(GL(2, GF(7)))
False
sage: is_ArithmeticSubgroup(Gamma0(4))
True
```

41.2 Arithmetic subgroups defined by permutations

A theorem of Millington states that an arithmetic subgroup of index N is uniquely determined by two elements generating a transitive subgroup of the symmetric group S_N and satisfying a certain algebraic relation.

These functions are based on Chris Kurth's *KFarey* package.

AUTHORS:

- Chris Kurth (2008): created KFarey package
- David Loeffler (2009): adapted functions from KFarey for inclusion into Sage

class ArithmeticSubgroup_Permutation(*L*, *R*)

An arithmetic subgroup Γ defined by two permutations, giving the action of the parabolic generators

$$L = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad R = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$$

by right multiplication on the coset representatives $\Gamma\backslash \mathrm{SL}_2(\mathbf{Z})$.

EXAMPLES:

We construct a noncongruence subgroup of index 7 (the smallest possible):

```
sage: a2 = SymmetricGroup(7)([(1,2),(3,4),(5,6)]); a3 = SymmetricGroup(7)([(1,3,5),(2,6,7)])
sage: G = ArithmeticSubgroup_Permutation(a2*a3, ~a2 * ~a3); G
Arithmetic subgroup corresponding to permutations L=(1,6)(2,3,4,5,7), R=(1,7,6,3,4)(2,5)
sage: G.index()
7
sage: G.dimension_cusp_forms(4)
1
sage: G.is_congruence()
False
```

We convert some standard congruence subgroups into permutation form:

```
sage: Gamma0(12).as_permutation_group()
Arithmetic subgroup corresponding to permutations L=(2,3,4,5,6,7,8,9,10,11,12,13)(14,15,16)(17,1
```

The following is the unique index 2 even subgroup of $\mathrm{SL}_2(\mathbf{Z})$:

```
sage: w = SymmetricGroup(2)([2,1])
sage: G = ArithmeticSubgroup_Permutation(w, w)
sage: G.dimension_cusp_forms(6)
1
sage: G.genus()
0
```

We test unpickling:

```
sage: G == loads(dumps(G))
True
sage: G is loads(dumps(G))
False
```

index()

Return the index of self in the full modular group.

EXAMPLE:

```
sage: import sage.modular.arithgroup.arithgroup_perm as ap
sage: ap.HsuExample18().index()
18
```

is_congruence()
Return True if this is a congruence subgroup. Uses Hsu's algorithm, as implemented by Chris Kurth in KFarey.

EXAMPLES:

This example is congruence – it's Gamma0(3) in disguise:

```
sage: import sage.modular.arithgroup.arithgroup_perm as ap
sage: G=ap.ArithmeticSubgroup_Permutation(SymmetricGroup(4)((2,3,4)), SymmetricGroup(4)((1,3
Arithmetic subgroup corresponding to permutations L=(2,3,4), R=(1,3,4)
sage: G.is_congruence()
True
```

This one is noncongruence:

```
sage: import sage.modular.arithgroup.arithgroup_perm as ap
sage: ap.HsuExample10().is_congruence()
False
```

perm_group()
Return the underlying permutation group.

EXAMPLE:

```
sage: import sage.modular.arithgroup.arithgroup_perm as ap
sage: ap.HsuExample10().perm_group()
Permutation Group with generators [(1,4)(2,5,9,10,8)(3,7,6), (1,7,9,10,6)(2,3)(4,5,8)]
```

permutation_action(*x*)
Given an element x of $\mathrm{SL}_2(\mathbf{Z})$, compute the permutation of the cosets of self given by right multiplication by x.

EXAMPLE:

```
sage: import sage.modular.arithgroup.arithgroup_perm as ap
sage: ap.HsuExample10().permutation_action(SL2Z([32, -21, -67, 44]))
(1,10,5,6,3,8,9,2,7,4)
```

HsuExample10()
An example of an index 10 arithmetic subgroup studied by Tim Hsu.

EXAMPLE:

```
sage: import sage.modular.arithgroup.arithgroup_perm as ap
sage: ap.HsuExample10()
Arithmetic subgroup corresponding to permutations L=(1,4)(2,5,9,10,8)(3,7,6), R=(1,7,9,10,6)(2,3
```

HsuExample18()
An example of an index 18 arithmetic subgroup studied by Tim Hsu.

EXAMPLE:

```
sage: import sage.modular.arithgroup.arithgroup_perm as ap
sage: ap.HsuExample18()
Arithmetic subgroup corresponding to permutations L=(1,2)(3,4)(5,6,7)(8,9,10)(11,12,13,14,15,16,
```

LREvalPerm(*w, L, R*)
Given a word w as output by sl2z_word_problem, evaluate the word with the given permutations for L and R. Because we are dealing with a right rather than a left action, arguments are evaluated back to front.

EXAMPLE:

```
sage: import  sage.modular.arithgroup.arithgroup_perm as ap
sage: L = SymmetricGroup(4)('(1,2)(3,4)'); R = SymmetricGroup(4)('(1,2,3,4)')
sage: ap.LREvalPerm([(1,1),(0,1)], L, R) ==  L * R
True
```

convert_to_permgroup(*G*)

Given an arbitrary arithmetic subgroup, convert it to permutation form.

Note that the permutation representation is not always unique, so if G is already of permutation type, then the return value won't necessarily be identical to G, but it will represent the same subgroup.

EXAMPLES:

```
sage: import sage.modular.arithgroup.arithgroup_perm as ap
sage: ap.convert_to_permgroup(Gamma0(5))
Arithmetic subgroup corresponding to permutations L=(2,3,4,5,6), R=(1,3,5,4,6)
sage: ap.convert_to_permgroup(ap.HsuExample10())
Arithmetic subgroup corresponding to permutations L=(1,2)(3,5,6,7,8)(4,9,10), R=(1,9,6,7,10)(2,5
```

eval_word(*B*)

Given a word in the format output by sl2z_word_problem, convert it back into an element of SL2(Z).

EXAMPLES:

```
sage: import sage.modular.arithgroup.arithgroup_perm as ap
sage: ap.eval_word([(0, 1), (1, -1), (0, 0), (1, 3), (0, 2), (1, 9), (0, -1)])
[ 66 -59]
[ 47 -42]
```

sl2z_word_problem(*A, max_iterations=20*)

Given an element of SL2Z, express it as a word in the generators L = [1,1,0,1] and R = [1,0,1,1].

The return format is a list of pairs (a,b), where a = 0 or 1 denoting L or R respectively, and b is an integer exponent.

The parameter iterations (default 20) controls the maximum number of iterations to allow in the program's main loop; an error is raised if the algorithm has not terminated after this many iterations.

EXAMPLE:

```
sage: import sage.modular.arithgroup.arithgroup_perm as ap
sage: L = SL2Z([1,1,0,1]); R = SL2Z([1,0,1,1])
sage: ap.sl2z_word_problem(L)
[(0, 1)]
sage: ap.sl2z_word_problem(R**(-1))
[(1, -1)]
sage: ap.sl2z_word_problem(L*R)
[(0, 1), (1, -1), (0, 0), (1, 2)]
```

41.3 Elements of Arithmetic Subgroups

class ArithmeticSubgroupElement(*parent, x, check=True*)

An element of an arithmetic subgroup of $\mathrm{SL}_2(\mathbf{Z})$.

a()

Return the upper left entry of self.

EXAMPLES:

```
sage: Gamma0(13)([7,1,13,2]).a()
7
```

acton(*z*)

Return the result of the action of self on z as a fractional linear transformation.

EXAMPLES:

```
sage: G = Gamma0(15)
sage: g = G([1, 2, 15, 31])
```

An example of g acting on a symbolic variable:

```
sage: z = var('z')
sage: g.acton(z)
(z + 2)/(15*z + 31)
```

An example involving the Gaussian numbers:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: g.acton(i)
1/1186*i + 77/1186
```

An example with complex numbers:

```
sage: C.<i> = ComplexField()
sage: g.acton(i)
0.0649241146711636 + 0.000843170320404721*I
```

b()

Return the upper right entry of self.

EXAMPLES:

```
sage: Gamma0(13)([7,1,13,2]).b()
1
```

c()

Return the lower left entry of self.

EXAMPLES:

```
sage: Gamma0(13)([7,1,13,2]).c()
13
```

d()

Return the lower right entry of self.

EXAMPLES:

```
sage: Gamma0(13)([7,1,13,2]).d()
2
```

det()

Return the determinant of self, which is always 1.

EXAMPLES:

```
sage: Gamma1(11)([12,11,-11,-10]).det()
1
```

determinant()

Return the determinant of self, which is always 1.

EXAMPLES:

```
sage: Gamma0(691)([1,0,691,1]).determinant()
1
```

matrix()

Return the matrix corresponding to self.

EXAMPLES:

```
sage: x = Gamma1(3)([4,5,3,4]) ; x
[4 5]
[3 4]
sage: x.matrix()
[4 5]
[3 4]
sage: type(x.matrix())
<type 'sage.matrix.matrix_integer_2x2.Matrix_integer_2x2'>
```

41.4 Congruence arithmetic subgroups of $\mathrm{SL}_2(\mathbf{Z})$

Sage can compute extensively with the standard congruence subgroups $\Gamma_0(N)$, $\Gamma_1(N)$, and $\Gamma_H(N)$.

AUTHOR: – William Stein

class CongruenceSubgroup (*level*)

is_congruence()

Return True, since this is a congruence subgroup.

EXAMPLE:

```
sage: Gamma0(7).is_congruence()
True
```

level()

Return the level of this congruence subgroup.

EXAMPLES:

```
sage: SL2Z.level()
1
sage: Gamma0(20).level()
20
sage: Gamma1(11).level()
11
sage: GammaH(14, [2]).level()
14
```

modular_abelian_variety()

Return the modular abelian variety corresponding to the congruence subgroup self.

EXAMPLES:

```
sage: Gamma0(11).modular_abelian_variety()
Abelian variety J0(11) of dimension 1
sage: Gamma1(11).modular_abelian_variety()
Abelian variety J1(11) of dimension 1
sage: GammaH(11,[3]).modular_abelian_variety()
Abelian variety JH(11,[3]) of dimension 1
```

modular_symbols(*sign=0, weight=2, base_ring=Rational Field*)

Return the space of modular symbols of the specified weight and sign on the congruence subgroup self.

EXAMPLES:

```
sage: G = Gamma0(23)
sage: G.modular_symbols()
Modular Symbols space of dimension 5 for Gamma_0(23) of weight 2 with sign 0 over Rational F
sage: G.modular_symbols(weight=4)
Modular Symbols space of dimension 12 for Gamma_0(23) of weight 4 with sign 0 over Rational
sage: G.modular_symbols(base_ring=GF(7))
Modular Symbols space of dimension 5 for Gamma_0(23) of weight 2 with sign 0 over Finite Fie
sage: G.modular_symbols(sign=1)
Modular Symbols space of dimension 3 for Gamma_0(23) of weight 2 with sign 1 over Rational F
```

sturm_bound(*weight=2*)

Returns the Sturm bound for modular forms of the given weight and level this congruence subgroup.

INPUT:

- `weight` - an integer ≥ 2 (default: 2)

EXAMPLES:: sage: Gamma0(11).sturm_bound(2) 2 sage: Gamma0(389).sturm_bound(2) 65 sage: Gamma0(1).sturm_bound(12) 1 sage: Gamma0(100).sturm_bound(2) 30 sage: Gamma0(1).sturm_bound(36) 3 sage: Gamma0(11).sturm_bound() 2 sage: Gamma0(13).sturm_bound() 3 sage: Gamma0(16).sturm_bound() 4 sage: GammaH(16,[13]).sturm_bound() 8 sage: GammaH(16,[15]).sturm_bound() 16 sage: Gamma1(16).sturm_bound() 32 sage: Gamma1(13).sturm_bound() 28 sage: Gamma1(13).sturm_bound(5) 70

FURTHER DETAILS: This function returns a positive integer n such that the Hecke operators $T_1, \ldots, T_n$ acting on *cusp forms* generate the Hecke algebra as a $\mathbf{Z}$-module when the character is trivial or quadratic. Otherwise, $T_1, \ldots, T_n$ generate the Hecke algebra at least as a $\mathbf{Z}[\varepsilon]$-module, where $\mathbf{Z}[\varepsilon]$ is the ring generated by the values of the Dirichlet character ε. Alternatively, this is a bound such that if two cusp forms associated to this space of modular symbols are congruent modulo (λ, q^n), then they are congruent modulo λ.

REFERENCES:

- See the Agashe-Stein appendix to Lario and Schoof, *Some computations with Hecke rings and deformation rings*, Experimental Math., 11 (2002), no. 2, 303-311.
- This result originated in the paper Sturm, *On the congruence of modular forms*, Springer LNM 1240, 275-280, 1987.

REMARK: Kevin Buzzard pointed out to me (William Stein) in Fall 2002 that the above bound is fine for $\Gamma_1(N)$ with character, as one sees by taking a power of f. More precisely, if $f \cong 0 \pmod{p}$ for first s coefficients, then $f^r \cong 0 \pmod{p}$ for first sr coefficents. Since the weight of f^r is $r \cdot k(f)$, it follows that if $s \geq b$, where b is the Sturm bound for $\Gamma_0(N)$ at weight $k(f)$, then f^r has valuation large enough to be forced to be 0 at $r * k(f)$ by Sturm bound (which is valid if we choose r correctly). Thus $f \cong 0 \pmod{p}$. Conclusion: For $\Gamma_1(N)$ with fixed character, the Sturm bound is *exactly* the same as for $\Gamma_0(N)$.

A key point is that we are finding $\mathbf{Z}[\varepsilon]$ generators for the Hecke algebra here, not $\mathbf{Z}$-generators. So if one wants generators for the Hecke algebra over $\mathbf{Z}$, this bound must be suitably modified (and I'm not sure what the modification is).

AUTHORS:

- William Stein

is_CongruenceSubgroup(*x*)

Return True if x is of type CongruenceSubgroup.

EXAMPLES:

```
sage: from sage.modular.arithgroup.congroup_generic import is_CongruenceSubgroup
sage: is_CongruenceSubgroup(SL2Z)
True
sage: is_CongruenceSubgroup(Gamma0(13))
True
sage: is_CongruenceSubgroup(Gamma1(6))
True
sage: is_CongruenceSubgroup(GammaH(11, [3]))
True
sage: is_CongruenceSubgroup(SymmetricGroup(3))
False
```

41.5 Congruence Subgroup $\Gamma_H(N)$

AUTHORS:

- Jordi Quer
- David Loeffler

class GammaH_class (*level, H*)

The congruence subgroup $\Gamma_H(N)$ for some subgroup $H \trianglelefteq (\mathbf{Z}/N\mathbf{Z})^\times$, which is the subgroup of $\mathrm{SL}_2(\mathbf{Z})$ consisting of matrices of the form $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $N \mid c$ and $a, b \in H$.

TESTS:

We test calculation of various invariants of the group:

```
sage: GammaH(33,[2]).projective_index()
96
sage: GammaH(33,[2]).genus()
5
sage: GammaH(7,[2]).genus()
0
sage: GammaH(23, [1..22]).genus()
2
sage: Gamma0(23).genus()
2
sage: GammaH(23, [1]).genus()
12
sage: Gamma1(23).genus()
12
```

We calculate the dimensions of some modular forms spaces:

```
sage: GammaH(33,[2]).dimension_cusp_forms(2)
5
sage: GammaH(33,[2]).dimension_cusp_forms(3)
0
sage: GammaH(33,[2,5]).dimension_cusp_forms(2)
3
sage: GammaH(32079, [21676]).dimension_cusp_forms(20)
180266112
```

We can sometimes show that there are no weight 1 cusp forms:

```
sage: GammaH(20, [9]).dimension_cusp_forms(1)
0
```

coset_reps()

Return a set of coset representatives for self \ SL2Z.

EXAMPLES:

```
sage: list(Gamma1(3).coset_reps())
[[1 0]
[0 1], [-1 -2]
[ 3  5], [ 0 -1]
[ 1  0], [-2  1]
[ 5 -3], [1 0]
[1 1], [-3 -2]
[ 8  5], [ 0 -1]
[ 1  2], [-2 -3]
[ 5  7]]
sage: len(list(Gamma1(31).coset_reps())) == 31**2 - 1
True
```

dimension_new_cusp_forms(*k=2, p=0*)

Return the dimension of the space of new (or p-new) weight k cusp forms for this congruence subgroup.

INPUT:

- k - an integer (default: 2), the weight. Not fully implemented for k = 1.
- p - integer (default: 0); if nonzero, compute the p-new subspace.

OUTPUT: Integer

EXAMPLES:

```
sage: GammaH(33,[2]).dimension_new_cusp_forms()
3
sage: Gamma1(4*25).dimension_new_cusp_forms(2, p=5)
225
sage: Gamma1(33).dimension_new_cusp_forms(2)
19
sage: Gamma1(33).dimension_new_cusp_forms(2,p=11)
21
```

divisor_subgroups()

Given this congruence subgroup $\Gamma_H(N)$, return all subgroups $\Gamma_G(M)$ for M a divisor of N and such that G is equal to the image of H modulo M.

EXAMPLES:

```
sage: G = GammaH(33,[2]); G
Congruence Subgroup Gamma_H(33) with H generated by [2]
sage: G._list_of_elements_in_H()
[1, 2, 4, 8, 16, 17, 25, 29, 31, 32]
sage: G.divisor_subgroups()
[Modular Group SL(2,Z),
 Congruence Subgroup Gamma_H(3) with H generated by [2],
 Congruence Subgroup Gamma_H(11) with H generated by [2],
 Congruence Subgroup Gamma_H(33) with H generated by [2]]
```

gamma0_coset_reps()

Return a set of coset representatives for self \ Gamma0(N), where N is the level of self.

EXAMPLE:

```
sage: GammaH(108, [1,-1]).gamma0_coset_reps()
[[1 0] [0 1], [-43 -45] [108 113], [ 31  33] [108 115], [-49 -54]
[108 119], [ 25  28] [108 121], [-19 -22] [108 125], [-17 -20] [108
127], [ 47  57] [108 131], [ 13  16] [108 133], [ 41  52] [108
137], [  7   9] [108 139], [-37 -49] [108 143], [-35 -47] [108
145], [ 29  40] [108 149], [ -5  -7] [108 151], [ 23  33] [108
155], [-11 -16] [108 157], [ 53  79] [108 161]]
```

generators()

Return generators for this congruence subgroup.

The result is cached.

EXAMPLE:

```
sage: for g in GammaH(3, [2]).generators():
...     print g
...     print '---'
[1 1]
[0 1]
 ---
[-1  0]
[ 0 -1]
---
[ 1 -1]
[ 0  1]
---
[1 0]
[3 1]
---
[1 1]
[0 1]
---
[-1  0]
[ 3 -1]
---
[ 1  0]
[-3  1]
---
```

index()

Return the index of self in SL2Z.

EXAMPLE:

```
sage: [G.index() for G in Gamma0(40).gamma_h_subgroups()]
[72, 144, 144, 144, 144, 288, 288, 288, 288, 144, 288, 288, 576, 576, 144, 288, 288, 576, 57
```

is_even()

Return True precisely if this subgroup contains the matrix -1.

EXAMPLES:

```
sage: GammaH(10, [3]).is_even()
True
sage: GammaH(14, [1]).is_even()
False
```

is_subgroup(*other*)

Return True if self is a subgroup of right, and False otherwise.

EXAMPLES:

```
sage: GammaH(24,[7]).is_subgroup(SL2Z)
True
sage: GammaH(24,[7]).is_subgroup(Gamma0(8))
True
sage: GammaH(24, []).is_subgroup(GammaH(24, [7]))
True
sage: GammaH(24, []).is_subgroup(Gamma1(24))
True
sage: GammaH(24, [17]).is_subgroup(GammaH(24, [7]))
False
sage: GammaH(1371, [169]).is_subgroup(GammaH(457, [169]))
True
```

ncusps()

Return the number of orbits of cusps (regular or otherwise) for this subgroup.

EXAMPLE:

```
sage: GammaH(33,[2]).ncusps()
8
sage: GammaH(32079, [21676]).ncusps()
28800
```

AUTHORS:

- Jordi Quer

nirregcusps()

Return the number of irregular cusps for this subgroup.

EXAMPLES:

```
sage: GammaH(3212, [2045, 2773]).nirregcusps()
720
```

nregcusps()

Return the number of orbits of regular cusps for this subgroup. A cusp is regular if we may find a parabolic element generating the stabiliser of that cusp whose eigenvalues are both +1 rather than -1. If G contains -1, all cusps are regular.

EXAMPLES:

```
sage: GammaH(20, [17]).nregcusps()
4
sage: GammaH(20, [17]).nirregcusps()
2
sage: GammaH(3212, [2045, 2773]).nregcusps()
1440
sage: GammaH(3212, [2045, 2773]).nirregcusps()
720
```

AUTHOR:

- Jordi Quer

nu2()

Return the number of orbits of elliptic points of order 2 for this group.

EXAMPLE:

```
sage: [H.nu2() for n in [1..10] for H in Gamma0(n).gamma_h_subgroups()]
[1, 1, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0]
sage: GammaH(33,[2]).nu2()
0
sage: GammaH(5,[2]).nu2()
2
```

AUTHORS:

•Jordi Quer

nu3()

Return the number of orbits of elliptic points of order 3 for this group.

EXAMPLE:

```
sage: [H.nu3() for n in [1..10] for H in Gamma0(n).gamma_h_subgroups()]
[1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
sage: GammaH(33,[2]).nu3()
0
sage: GammaH(7,[2]).nu3()
2
```

AUTHORS:

•Jordi Quer

reduce_cusp(c)

Compute a minimal representative for the given cusp c. Returns a cusp c' which is equivalent to the given cusp, and is in lowest terms with minimal positive denominator, and minimal positive numerator for that denominator.

Two cusps u_1/v_1 and u_2/v_2 are equivalent modulo $\Gamma_H(N)$ if and only if

$$v_1 = hv_2 \bmod N \quad \text{and} \quad u_1 = h^{-1}u_2 \bmod \gcd(v_1, N)$$

or

$$v_1 = -hv_2 \bmod N \quad \text{and} \quad u_1 = -h^{-1}u_2 \bmod \gcd(v_1, N)$$

for some $h \in H$.

EXAMPLES:

```
sage: GammaH(6,[5]).reduce_cusp(Cusp(5,3))
1/3
sage: GammaH(12,[5]).reduce_cusp(Cusp(8,9))
1/3
sage: GammaH(12,[5]).reduce_cusp(Cusp(5,12))
Infinity
sage: GammaH(12,[]).reduce_cusp(Cusp(5,12))
5/12
sage: GammaH(21,[5]).reduce_cusp(Cusp(-9/14))
1/7
```

restrict(M)

Return the subgroup of $\Gamma_0(M)$ obtained by taking H to be the image of the H at level N modulo M.

EXAMPLES:

```
sage: G = GammaH(33,[2])
sage: G.restrict(11)
Congruence Subgroup Gamma_H(11) with H generated by [2]
sage: G.restrict(1)
Modular Group SL(2,Z)
sage: G.restrict(15)
...
ValueError: M (=15) must be a divisor of the level (33) of self
```

GammaH_constructor(*level, H*)

Return the congruence subgroup $\Gamma_H(N)$, which is the subgroup of $SL_2(\mathbf{Z})$ consisting of matrices of the form $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ with $N|c$ and $a, b \in H$, for H a specified subgroup of $(\mathbf{Z}/N\mathbf{Z})^\times$.

INPUT:

- level – an integer
- **H – either 0, 1, or a list** – If H is a list, return $\Gamma_H(N)$, where H is the subgroup of $(\mathbf{Z}/N\mathbf{Z})^*$ **generated** by the elements of the list.
  - If H = 0, returns $\Gamma_0(N)$.
  - If H = 1, returns $\Gamma_1(N)$.

EXAMPLES:

```
sage: GammaH(11,0) # indirect doctest
Congruence Subgroup Gamma0(11)
sage: GammaH(11,1)
Congruence Subgroup Gamma1(11)
sage: GammaH(11,[2])
Congruence Subgroup Gamma_H(11) with H generated by [2]
sage: GammaH(11,[2,1])
Congruence Subgroup Gamma_H(11) with H generated by [2]
```

is_GammaH(x)

Return True if x is a congruence subgroup of type GammaH.

EXAMPLES:

```
sage: from sage.modular.arithgroup.all import is_GammaH
sage: is_GammaH(GammaH(13, [2]))
True
sage: is_GammaH(Gamma0(6))
True
sage: is_GammaH(sage.modular.arithgroup.congroup_generic.CongruenceSubgroup(5))
False
```

mumu(N)

Return 0 if any cube divides N. Otherwise return $(-2)^v$ where v is the number of primes that exactly divide N. This is similar to the Moebius function.

INPUT:

- N - an integer at least 1

OUTPUT: Integer

EXAMPLES:

```
sage: from sage.modular.arithgroup.congroup_gammaH import mumu
sage: mumu(27)
0
sage: mumu(6*25)
4
sage: mumu(7*9*25)
-2
sage: mumu(9*25)
1
```

41.6 Congruence Subgroup $\Gamma_1(N)$

class Gamma1_class(*level*)

The congruence subgroup $\Gamma_1(N)$.

TESTS:

```
sage: [Gamma1(n).genus() for n in prime_range(2,100)]
[0, 0, 0, 0, 1, 2, 5, 7, 12, 22, 26, 40, 51, 57, 70, 92, 117, 126, 155, 176, 187, 222, 247, 287,
sage: [Gamma1(n).index() for n in [1..10]]
[1, 3, 8, 12, 24, 24, 48, 48, 72, 72]

sage: [Gamma1(n).dimension_cusp_forms() for n in [1..20]]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 2, 1, 1, 2, 5, 2, 7, 3]
sage: [Gamma1(n).dimension_cusp_forms(1) for n in [1..20]]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
sage: [Gamma1(4).dimension_cusp_forms(k) for k in [1..20]]
[0, 0, 0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8]
sage: Gamma1(23).dimension_cusp_forms(1)
...
NotImplementedError: Computation of dimensions of weight 1 cusp forms spaces not implemented in
```

dimension_cusp_forms (*k=2, eps=None, algorithm='CohenOesterle'*)

Return the dimension of the space of cusp forms for self, or the dimension of the subspace corresponding to the given character if one is supplied.

INPUT:

- k - an integer (default: 2), the weight.
- eps - either None or a Dirichlet character modulo N, where N is the level of this group. If this is None, then the dimension of the whole space is returned; otherwise, the dimension of the subspace of forms of character eps.
- algorithm – either "CohenOesterle" (the default) or "Quer". This specifies the method to use in the case of nontrivial character: either the Cohen–Oesterle formula as described in Stein's book, or by Moebius inversion using the subgroups GammaH (a method due to Jordi Quer).

EXAMPLES:

We compute the same dimension in two different ways

```
sage: K = CyclotomicField(3)
sage: eps = DirichletGroup(7*43,K).0^2
sage: G = Gamma1(7*43)
```

Via Cohen–Oesterle:

```
sage: Gamma1(7*43).dimension_cusp_forms(2, eps)
28
```

Via Quer's method:

```
sage: Gamma1(7*43).dimension_cusp_forms(2, eps, algorithm="Quer")
28
```

Some more examples:

```
sage: G.<eps> = DirichletGroup(9)
sage: [Gamma1(9).dimension_cusp_forms(k, eps) for k in [1..10]]
[0, 0, 1, 0, 3, 0, 5, 0, 7, 0]
sage: [Gamma1(9).dimension_cusp_forms(k, eps^2) for k in [1..10]]
[0, 0, 0, 2, 0, 4, 0, 6, 0, 8]
```

dimension_eis (*k=2, eps=None, algorithm='CohenOesterle'*)

Return the dimension of the space of Eisenstein series forms for self, or the dimension of the subspace corresponding to the given character if one is supplied.

INPUT:

- k - an integer (default: 2), the weight.

- eps - either None or a Dirichlet character modulo N, where N is the level of this group. If this is None, then the dimension of the whole space is returned; otherwise, the dimension of the subspace of Eisenstein series of character eps.
- algorithm – either "CohenOesterle" (the default) or "Quer". This specifies the method to use in the case of nontrivial character: either the Cohen–Oesterle formula as described in Stein's book, or by Moebius inversion using the subgroups GammaH (a method due to Jordi Quer).

AUTHORS:

- William Stein - Cohen–Oesterle algorithm
- Jordi Quer - algorithm based on GammaH subgroups
- David Loeffler (2009) - code refactoring

EXAMPLES:

The following two computations use different algorithms:

```
sage: [Gamma1(36).dimension_eis(1,eps) for eps in DirichletGroup(36)]
[0, 4, 3, 0, 0, 2, 6, 0, 0, 2, 3, 0]
sage: [Gamma1(36).dimension_eis(1,eps,algorithm="Quer") for eps in DirichletGroup(36)]
[0, 4, 3, 0, 0, 2, 6, 0, 0, 2, 3, 0]
```

So do these:

```
sage: [Gamma1(48).dimension_eis(3,eps) for eps in DirichletGroup(48)]
[0, 12, 0, 4, 0, 8, 0, 4, 12, 0, 4, 0, 8, 0, 4, 0]
sage: [Gamma1(48).dimension_eis(3,eps,algorithm="Quer") for eps in DirichletGroup(48)]
[0, 12, 0, 4, 0, 8, 0, 4, 12, 0, 4, 0, 8, 0, 4, 0]
```

dimension_modular_forms (*k=2, eps=None, algorithm='CohenOesterle'*)

Return the dimension of the space of modular forms for self, or the dimension of the subspace corresponding to the given character if one is supplied.

INPUT:

- k - an integer (default: 2), the weight.
- eps - either None or a Dirichlet character modulo N, where N is the level of this group. If this is None, then the dimension of the whole space is returned; otherwise, the dimension of the subspace of forms of character eps.
- algorithm – either "CohenOesterle" (the default) or "Quer". This specifies the method to use in the case of nontrivial character: either the Cohen–Oesterle formula as described in Stein's book, or by Moebius inversion using the subgroups GammaH (a method due to Jordi Quer).

EXAMPLES:

```
sage: K = CyclotomicField(3)
sage: eps = DirichletGroup(7*43,K).0^2
sage: G = Gamma1(7*43)

sage: G.dimension_modular_forms(2, eps)
32
sage: G.dimension_modular_forms(2, eps, algorithm="Quer")
32
```

dimension_new_cusp_forms (*k=2, eps=None, p=0, algorithm='CohenOesterle'*)

Dimension of the new subspace (or p-new subspace) of cusp forms of weight k and character ε.

INPUT:

- k - an integer (default: 2)
- eps - a Dirichlet character
- p - a prime (default: 0); just the p-new subspace if given

- `algorithm` - either "CohenOesterle" (the default) or "Quer". This specifies the method to use in the case of nontrivial character: either the Cohen–Oesterle formula as described in Stein's book, or by Moebius inversion using the subgroups GammaH (a method due to Jordi Quer).

EXAMPLES:

```
sage: G = DirichletGroup(9)
sage: eps = G.0^3
sage: eps.conductor()
3
sage: [Gamma1(9).dimension_new_cusp_forms(k, eps) for k in [2..10]]
[0, 0, 0, 2, 0, 2, 0, 2, 0]
sage: [Gamma1(9).dimension_cusp_forms(k, eps) for k in [2..10]]
[0, 0, 0, 2, 0, 4, 0, 6, 0]
sage: [Gamma1(9).dimension_new_cusp_forms(k, eps, 3) for k in [2..10]]
[0, 0, 0, 2, 0, 2, 0, 2, 0]
```

Double check using modular symbols (independent calculation):

```
sage: [ModularSymbols(eps,k,sign=1).cuspidal_subspace().new_subspace().dimension()  for k in
[0, 0, 0, 2, 0, 2, 0, 2, 0]
sage: [ModularSymbols(eps,k,sign=1).cuspidal_subspace().new_subspace(3).dimension()  for k i
[0, 0, 0, 2, 0, 2, 0, 2, 0]
```

Another example at level 33:

```
sage: G = DirichletGroup(33)
sage: eps = G.1
sage: eps.conductor()
11
sage: [Gamma1(33).dimension_new_cusp_forms(k, G.1) for k in [2..4]]
[0, 4, 0]
sage: [Gamma1(33).dimension_new_cusp_forms(k, G.1, algorithm="Quer") for k in [2..4]]
[0, 4, 0]
sage: [Gamma1(33).dimension_new_cusp_forms(k, G.1^2) for k in [2..4]]
[2, 0, 6]
sage: [Gamma1(33).dimension_new_cusp_forms(k, G.1^2, p=3) for k in [2..4]]
[2, 0, 6]
```

generators()

Return generators for this congruence subgroup.

The result is cached.

EXAMPLE:

```
sage: for g in Gamma1(3).generators():
...       print g
...       print '---'
[1 1]
[0 1]
---
[-20   9]
[ 51 -23]
---
[ 4  1]
[-9 -2]
---
...
---
[ 4 -1]
[ 9 -2]
---
```

```
[ -5   3]
[-12   7]
---
```

index()

Return the index of self in the full modular group. This is given by the formula

$$N^2 \prod_{\substack{p|N \\ p \text{ prime}}} \left(1 - \frac{1}{p^2}\right).$$

EXAMPLE:

```
sage: Gamma1(180).index()
20736
sage: [Gamma1(n).projective_index() for n in [1..16]]
[1, 3, 4, 6, 12, 12, 24, 24, 36, 36, 60, 48, 84, 72, 96, 96]
```

is_even()

Return True precisely if this subgroup contains the matrix -1.

EXAMPLES:

```
sage: Gamma1(1).is_even()
True
sage: Gamma1(2).is_even()
True
sage: Gamma1(15).is_even()
False
```

is_subgroup(*right*)

Return True if self is a subgroup of right.

EXAMPLES:

```
sage: Gamma1(3).is_subgroup(SL2Z)
True
sage: Gamma1(3).is_subgroup(Gamma1(5))
False
sage: Gamma1(3).is_subgroup(Gamma1(6))
False
sage: Gamma1(6).is_subgroup(Gamma1(3))
True
sage: Gamma1(6).is_subgroup(Gamma0(2))
True
sage: Gamma1(80).is_subgroup(GammaH(40, []))
True
sage: Gamma1(80).is_subgroup(GammaH(40, [21]))
True
```

ncusps()

Return the number of cusps of this subgroup $\Gamma_1(N)$.

EXAMPLES:

```
sage: [Gamma1(n).ncusps() for n in [1..15]]
[1, 2, 2, 3, 4, 4, 6, 6, 8, 8, 10, 10, 12, 12, 16]
sage: [Gamma1(n).ncusps() for n in prime_range(2, 100)]
[2, 2, 4, 6, 10, 12, 16, 18, 22, 28, 30, 36, 40, 42, 46, 52, 58, 60, 66, 70, 72, 78, 82, 88,
```

nu2()

Calculate the number of orbits of elliptic points of order 2 for this subgroup $\Gamma_1(N)$. This is known to be 0 if N > 2.

EXAMPLE:

```
sage: Gamma1(2).nu2()
1
sage: Gamma1(457).nu2()
0
sage: [Gamma1(n).nu2() for n in [1..16]]
[1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

nu3()

Calculate the number of orbits of elliptic points of order 3 for this subgroup $\Gamma_1(N)$. This is known to be 0 if N > 3.

EXAMPLE:

```
sage: Gamma1(2).nu3()
0
sage: Gamma1(3).nu3()
1
sage: Gamma1(457).nu3()
0
sage: [Gamma1(n).nu3() for n in [1..10]]
[1, 0, 1, 0, 0, 0, 0, 0, 0, 0]
```

Gamma1_constructor(N)

Return the congruence subgroup $\Gamma_1(N)$.

EXAMPLES:

```
sage: Gamma1(5) # indirect doctest
Congruence Subgroup Gamma1(5)
sage: G = Gamma1(23)
sage: G is Gamma1(23)
True
sage: G == loads(dumps(G))
True
sage: G is loads(dumps(G))
True
```

is_Gamma1(x)

Return True if x is a congruence subgroup of type Gamma1.

EXAMPLES:

```
sage: from sage.modular.arithgroup.all import is_Gamma1
sage: is_Gamma1(SL2Z)
True
sage: is_Gamma1(Gamma1(13))
True
sage: is_Gamma1(Gamma0(6))
False
sage: is_Gamma1(GammaH(12, []))
False
```

41.7 Congruence Subgroup $\Gamma_0(N)$

class Gamma0_class(*level*)

The congruence subgroup $\Gamma_0(N)$.

TESTS:

```
sage: Gamma0(11).dimension_cusp_forms(2)
1
sage: a = Gamma0(1).dimension_cusp_forms(2); a
0
sage: type(a)
<type 'sage.rings.integer.Integer'>
sage: Gamma0(5).dimension_cusp_forms(0)
0
sage: Gamma0(20).dimension_cusp_forms(1)
0
sage: Gamma0(20).dimension_cusp_forms(4)
6

sage: Gamma0(23).dimension_cusp_forms(2)
2
sage: Gamma0(1).dimension_cusp_forms(24)
2
sage: Gamma0(3).dimension_cusp_forms(3)
0
sage: Gamma0(11).dimension_cusp_forms(-1)
0

sage: Gamma0(22).dimension_new_cusp_forms()
0
sage: Gamma0(100).dimension_new_cusp_forms(2, 5)
5
```

Independently compute the dimension 5 above:

```
sage: m = ModularSymbols(100, 2,sign=1).cuspidal_subspace()
sage: m.new_subspace(5)
Modular Symbols subspace of dimension 5 of Modular Symbols space of dimension 18 for Gamma_0(100
```

coset_reps()

Return representatives for the right cosets of this congruence subgroup in $\mathrm{SL}_2(\mathbf{Z})$ as a generator object.

Use `list(self.coset_reps())` to obtain coset reps as a list.

EXAMPLES:

```
sage: list(Gamma0(5).coset_reps())
[[1 0]
[0 1], [ 0 -1]
[ 1  0], [1 0]
[1 1], [ 0 -1]
[ 1  2], [ 0 -1]
[ 1  3], [ 0 -1]
[ 1  4]]
sage: list(Gamma0(4).coset_reps())
 [[1 0] [0 1],
 [ 0 -1] [ 1  0],
 [1 0] [1 1],
 [ 0 -1] [ 1  2],
 [ 0 -1] [ 1  3],
 [1 0] [2 1]]
sage: list(Gamma0(1).coset_reps())
[[1 0] [0 1]]
```

divisor_subgroups()

Return the subgroups of SL2Z of the form Gamma0(M) that contain this subgroup, i.e. those for M a divisor of N.

EXAMPLE:

```
sage: Gamma0(24).divisor_subgroups()
[Modular Group SL(2,Z),
Congruence Subgroup Gamma0(2),
Congruence Subgroup Gamma0(3),
Congruence Subgroup Gamma0(4),
Congruence Subgroup Gamma0(6),
Congruence Subgroup Gamma0(8),
Congruence Subgroup Gamma0(12),
Congruence Subgroup Gamma0(24)]
```

gamma_h_subgroups()

Return the subgroups of the form $\Gamma_H(N)$ contained in self, where N is the level of self.

EXAMPLES:

```
sage: G = Gamma0(11)
sage: G.gamma_h_subgroups()
[Congruence Subgroup Gamma_H(11) with H generated by [2], Congruence Subgroup Gamma_H(11) wi
sage: G = Gamma0(12)
sage: G.gamma_h_subgroups()
[Congruence Subgroup Gamma_H(12) with H generated by [5, 7], Congruence Subgroup Gamma_H(12)
```

generators()

Return generators for this congruence subgroup.

The result is cached.

EXAMPLE:

```
sage: for g in Gamma0(3).generators():
...       print g
...       print '---'
[1 1]
[0 1]
---
[-1  0]
[ 0 -1]
---
...
---
[ 1  0]
[-3  1]
---
```

index()

Return the index of self in the full modular group. This is given by

$$N \prod_{\substack{p|N \\ p \text{ prime}}} \left(1 + \frac{1}{p}\right).$$

EXAMPLE:: sage: [Gamma0(n).index() for n in [1..19]] [1, 3, 4, 6, 6, 12, 8, 12, 12, 18, 12, 24, 14, 24, 24, 24, 18, 36, 20] sage: Gamma0(32041).index() 32220

is_even()

Return True precisely if this subgroup contains the matrix -1.

Since $\Gamma 0(N)$ always contains the matrix -1, this always returns True.

EXAMPLES:

```
sage: Gamma0(12).is_even()
True
sage: SL2Z.is_even()
True
```

is_subgroup(*right*)

Return True if self is a subgroup of right.

EXAMPLES:

```
sage: G = Gamma0(20)
sage: G.is_subgroup(SL2Z)
True
sage: G.is_subgroup(Gamma0(4))
True
sage: G.is_subgroup(Gamma0(20))
True
sage: G.is_subgroup(Gamma0(7))
False
sage: G.is_subgroup(Gamma1(20))
False
sage: G.is_subgroup(GammaH(40, []))
False
sage: Gamma0(80).is_subgroup(GammaH(40, [31, 21, 17]))
True
sage: Gamma0(2).is_subgroup(Gamma1(2))
True
```

ncusps()

Return the number of cusps of this subgroup $\Gamma_0(N)$.

EXAMPLES:

```
sage: [Gamma0(n).ncusps() for n in [1..19]]
[1, 2, 2, 3, 2, 4, 2, 4, 4, 4, 2, 6, 2, 4, 4, 6, 2, 8, 2]
sage: [Gamma0(n).ncusps() for n in prime_range(2,100)]
[2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2]
```

nu2()

Return the number of elliptic points of order 2 for this congruence subgroup $\Gamma_0(N)$. The number of these is given by a standard formula: 0 if N is divisible by 4 or any prime congruent to -1 mod 4, and otherwise 2^d where d is the number of odd primes dividing N.

EXAMPLE:

```
sage: Gamma0(2).nu2()
1
sage: Gamma0(4).nu2()
0
sage: Gamma0(21).nu2()
0
sage: Gamma0(1105).nu2()
8
sage: [Gamma0(n).nu2() for n in [1..19]]
[1, 1, 0, 0, 2, 0, 0, 0, 0, 2, 0, 0, 2, 0, 0, 0, 2, 0, 0]
```

nu3()

Return the number of elliptic points of order 3 for this congruence subgroup $\Gamma_0(N)$. The number of these is given by a standard formula: 0 if N is divisible by 9 or any prime congruent to -1 mod 3, and otherwise 2^d where d is the number of primes other than 3 dividing N.

EXAMPLE:

```
sage: Gamma0(2).nu3()
0
sage: Gamma0(3).nu3()
1
sage: Gamma0(9).nu3()
0
sage: Gamma0(7).nu3()
2
sage: Gamma0(21).nu3()
2
sage: Gamma0(1729).nu3()
8
```

Gamma0_constructor(*N*)

Return the congruence subgroup Gamma0(N).

EXAMPLES:

```
sage: G = Gamma0(51) ; G # indirect doctest
Congruence Subgroup Gamma0(51)
sage: G == Gamma0(51)
True
sage: G is Gamma0(51)
True
```

is_Gamma0(*x*)

Return True if x is a congruence subgroup of type Gamma0.

EXAMPLES:

```
sage: from sage.modular.arithgroup.all import is_Gamma0
sage: is_Gamma0(SL2Z)
True
sage: is_Gamma0(Gamma0(13))
True
sage: is_Gamma0(Gamma1(6))
False
```

41.8 Congruence Subgroup $\Gamma(N)$

class Gamma_class(*level*)

The principal congruence subgroup $\Gamma(N)$.

Gamma_constructor(*N*)

Return the congruence subgroup $\Gamma(N)$.

EXAMPLES:

```
sage: Gamma(5) # indirect doctest
Congruence Subgroup Gamma(5)
sage: G = Gamma(23)
sage: G is Gamma(23)
True
sage: G == loads(dumps(G))
True
sage: G is loads(dumps(G))
True
```

is_Gamma(*x*)
Return True if x is a congruence subgroup of type Gamma.

EXAMPLES:

```
sage: from sage.modular.arithgroup.all import is_Gamma
sage: is_Gamma(Gamma0(13))
False
sage: is_Gamma(Gamma(4))
True
```

41.9 The modular group $\mathrm{SL}_2(\mathbf{Z})$

class SL2Z_class()
The full modular group $\mathrm{SL}_2(\mathbf{Z})$, regarded as a congruence subgroup of itself.

is_subgroup(*right*)
Return True if self is a subgroup of right.

EXAMPLES:

```
sage: SL2Z.is_subgroup(SL2Z)
True
sage: SL2Z.is_subgroup(Gamma1(1))
True
sage: SL2Z.is_subgroup(Gamma0(6))
False
```

random_element(*bound=100*)
Return a random element of $\mathrm{SL}_2(\mathbf{Z})$ with entries whose absolute value is strictly less than bound (default 100).

(Algorithm: Generate a random pair of integers at most bound. If they are not coprime, throw them away and start again. If they are, find an element of $\mathrm{SL}_2(\mathbf{Z})$ whose bottom row is that, and left-multiply it by $\begin{pmatrix} 1 & w \\ 0 & 1 \end{pmatrix}$ for an integer w randomly chosen from a small enough range that the answer still has entries at most bound.)

It is, unfortunately, not true that all elements of SL2Z with entries < bound appear with equal probability; those with larger bottom rows are favoured, because there are fewer valid possibilities for w.

EXAMPLES:

```
sage: SL2Z.random_element() # random
sage: SL2Z.random_element(5) # still random
```

reduce_cusp(*c*)
Return the unique reduced cusp equivalent to c under the action of self. Always returns Infinity, since there is only one equivalence class of cusps for $SL_2(Z)$.

EXAMPLES:

```
sage: SL2Z.reduce_cusp(Cusps(-1/4))
Infinity
```

is_SL2Z(*x*)
Return True if x is the modular group $\mathrm{SL}_2(\mathbf{Z})$.

EXAMPLES:

```
sage: from sage.modular.arithgroup.all import is_SL2Z
sage: is_SL2Z(SL2Z)
True
sage: is_SL2Z(Gamma0(6))
False
```

CHAPTER

FORTYTWO

GENERAL HECKE ALGEBRAS AND HECKE MODULES

This chapter describes the basic functionality for modules over Hecke algebras, including decompositions, degeneracy maps and so on. For specific examples of Hecke algebras that use this functionality see *Modular Symbols* and *Modular Forms*.

42.1 Hecke modules

class HeckeModule_free_module(*base_ring, level, weight*)

A Hecke module modeled on a free module over a commutative ring.

T(*n*)

Returns the n^{th} Hecke operator T_n. This function is a synonym for `hecke_operator()`.

EXAMPLE:

```
sage: M = ModularSymbols(11,2)
sage: M.T(3)
Hecke operator T_3 on Modular Symbols ...
```

ambient()

Synonym for ambient_hecke_module. Return the ambient module associated to this module.

EXAMPLE:

```
sage: CuspForms(1, 12).ambient()
Modular Forms space of dimension 2 for Modular Group SL(2,Z) of weight 12 over Rational Fiel
```

ambient_hecke_module()

Return the ambient module associated to this module. As this is an abstract base class, return NotImplementedError.

EXAMPLE:

```
sage: sage.modular.hecke.module.HeckeModule_free_module(QQ, 10, 3).ambient_hecke_module()
...
NotImplementedError
```

ambient_module()

Synonym for ambient_hecke_module. Return the ambient module associated to this module.

EXAMPLE:

```
sage: CuspForms(1, 12).ambient_module()
Modular Forms space of dimension 2 for Modular Group SL(2,Z) of weight 12 over Rational Fiel
```

```
sage: sage.modular.hecke.module.HeckeModule_free_module(QQ, 10, 3).ambient_module()
...
NotImplementedError
```

atkin_lehner_operator(*d=None*)

Return the Atkin-Lehner operator W_d on this space, if defined, where d is a divisor of the level N such that N/d and d are coprime.

EXAMPLES:

```
sage: M = ModularSymbols(11)
sage: w = M.atkin_lehner_operator()
sage: w
Hecke module morphism Atkin-Lehner operator W_11 defined by the matrix
[-1  0  0]
[ 0 -1  0]
[ 0  0 -1]
Domain: Modular Symbols space of dimension 3 for Gamma_0(11) of weight ...
Codomain: Modular Symbols space of dimension 3 for Gamma_0(11) of weight ...
sage: M = ModularSymbols(Gamma1(13))
sage: w = M.atkin_lehner_operator()
sage: w.fcp('x')
(x - 1)^7 * (x + 1)^8

sage: M = ModularSymbols(33)
sage: S = M.cuspidal_submodule()
sage: S.atkin_lehner_operator()
Hecke module morphism Atkin-Lehner operator W_33 defined by the matrix
[ 0 -1  0  1 -1  0]
[ 0 -1  0  0  0  0]
[ 0 -1  0  0 -1  1]
[ 1 -1  0  0 -1  0]
[ 0  0  0  0 -1  0]
[ 0 -1  1  0 -1  0]
Domain: Modular Symbols subspace of dimension 6 of Modular Symbols space ...
Codomain: Modular Symbols subspace of dimension 6 of Modular Symbols space ...

sage: S.atkin_lehner_operator(3)
Hecke module morphism Atkin-Lehner operator W_3 defined by the matrix
[ 0  1  0 -1  1  0]
[ 0  1  0  0  0  0]
[ 0  1  0  0  1 -1]
[-1  1  0  0  1  0]
[ 0  0  0  0  1  0]
[ 0  1 -1  0  1  0]
Domain: Modular Symbols subspace of dimension 6 of Modular Symbols space ...
Codomain: Modular Symbols subspace of dimension 6 of Modular Symbols space ...

sage: N = M.new_submodule()
sage: N.atkin_lehner_operator()
Hecke module morphism Atkin-Lehner operator W_33 defined by the matrix
[  1 2/5 4/5]
[  0  -1   0]
[  0   0  -1]
Domain: Modular Symbols subspace of dimension 3 of Modular Symbols space ...
Codomain: Modular Symbols subspace of dimension 3 of Modular Symbols space ...
```

basis()

Returns a basis for self.

EXAMPLES:

```
sage: m = ModularSymbols(43)
sage: m.basis()
((1,0), (1,31), (1,32), (1,38), (1,39), (1,40), (1,41))
```

basis_matrix()

Return the matrix of the basis vectors of self (as vectors in some ambient module)

EXAMPLE:

```
sage: CuspForms(1, 12).basis_matrix()
[1 0]
```

coordinate_vector(*x*)

Write x as a vector with respect to the basis given by self.basis().

EXAMPLES:

```
sage: S = ModularSymbols(11,2).cuspidal_submodule()
sage: S.0
(1,8)
sage: S.basis()
((1,8), (1,9))
sage: S.coordinate_vector(S.0)
(1, 0)
```

decomposition(*bound=None, anemic=True, height_guess=1, proof=None*)

Returns the maximal decomposition of this Hecke module under the action of Hecke operators of index coprime to the level. This is the finest decomposition of self that we can obtain using factors obtained by taking kernels of Hecke operators.

Each factor in the decomposition is a Hecke submodule obtained as the kernel of $f(T_n)^r$ acting on self, where n is coprime to the level and $r = 1$. If anemic is False, instead choose r so that $f(X)^r$ exactly divides the characteristic polynomial.

INPUT:

- `anemic` - bool (default: True), if True, use only Hecke operators of index coprime to the level.
- `bound` - int or None, (default: None). If None, use all Hecke operators up to the Sturm bound, and hence obtain the same result as one would obtain by using every element of the Hecke ring. If a fixed integer, decompose using only Hecke operators T_p, with p prime, up to bound.

OUTPUT:

- `list` - a list of subspaces of self.

EXAMPLES:

```
sage: ModularSymbols(17,2).decomposition()
[
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0(
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 3 for Gamma_0(
]
sage: ModularSymbols(Gamma1(10),4).decomposition()
[
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 18 for Gamma_1
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 18 for Gamma_1
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 18 for Gamma_1
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 18 for Gamma_1
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 18 for Gamma_1
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 18 for Gamma_1
]
sage: ModularSymbols(GammaH(12, [11])).decomposition()
[
```

```
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 9 for Congruen
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 9 for Congruen
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 9 for Congruen
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 9 for Congruen
Modular Symbols subspace of dimension 5 of Modular Symbols space of dimension 9 for Congruen
]
```

degree()

The degree of this Hecke module (i.e. the rank of the ambient free module)

EXAMPLE:

```
sage: CuspForms(1, 12).degree()
2
```

dual_eigenvector(*names='alpha', lift=True, nz=None*)

Return an eigenvector for the Hecke operators acting on the linear dual of this space. This eigenvector will have entries in an extension of the base ring of degree equal to the dimension of this space.

INPUT:

- `name` - print name of generator for eigenvalue field.
- `lift` - bool (default: True)
- `nz` - if not None, then normalize vector so dot product with this basis vector of ambient space is 1.

OUTPUT: A vector with entries possibly in an extension of the base ring. This vector is an eigenvector for all Hecke operators acting via their transpose.

If lift = False, instead return an eigenvector in the subspace for the Hecke operators on the dual space. I.e., this is an eigenvector for the restrictions of Hecke operators to the dual space.

Note:

1. The answer is cached so subsequent calls always return the same vector. However, the algorithm is randomized, so calls during another session may yield a different eigenvector. This function is used mainly for computing systems of Hecke eigenvalues.
2. One can also view a dual eigenvector as defining (via dot product) a functional phi from the ambient space of modular symbols to a field. This functional phi is an eigenvector for the dual action of Hecke operators on functionals.

EXAMPLE:

```
sage: ModularSymbols(14).cuspidal_subspace().simple_factors()[1].dual_eigenvector()
(0, 1, 0, 0, 0)
```

dual_hecke_matrix(*n*)

The matrix of the n-th Hecke operator acting on the dual embedded representation of self.

EXAMPLE:

```
sage: CuspForms(1, 24).dual_hecke_matrix(5)
[     44656110        -15040]
[-307849789440      28412910]
```

eigenvalue(*n, name='alpha'*)

Assuming that self is a simple space, return the eigenvalue of the n^{th} Hecke operator on self.

INPUT:

- `n` - index of Hecke operator
- `name` - print representation of generator of eigenvalue field

EXAMPLES:

```
sage: A = ModularSymbols(125,sign=1).new_subspace()[0]
sage: A.eigenvalue(7)
-3
sage: A.eigenvalue(3)
-alpha - 2
sage: A.eigenvalue(3,'w')
-w - 2
sage: A.eigenvalue(3,'z').charpoly('x')
x^2 + 3*x + 1
sage: A.hecke_polynomial(3)
x^2 + 3*x + 1

sage: M = ModularSymbols(Gamma1(17)).decomposition()[8].plus_submodule()
sage: M.eigenvalue(2,'a')
a
sage: M.eigenvalue(4,'a')
4/3*a^3 + 17/3*a^2 + 28/3*a + 8/3
```

Note:

1. In fact there are d systems of eigenvalues associated to self, where d is the rank of self. Each of the systems of eigenvalues is conjugate over the base field. This function chooses one of the systems and consistently returns eigenvalues from that system. Thus these are the coefficients a_n for $n \geq 1$ of a modular eigenform attached to self.
2. This function works even for Eisenstein subspaces, though it will not give the constant coefficient of one of the corresponding Eisenstein series (i.e., the generalized Bernoulli number).

factor_number()

If this Hecke module was computed via a decomposition of another Hecke module, this is the corresponding number. Otherwise return -1.

EXAMPLES:

```
sage: ModularSymbols(23)[0].factor_number()
0
sage: ModularSymbols(23).factor_number()
-1
```

gen(n)

Return the nth basis vector of the space.

EXAMPLE:

```
sage: ModularSymbols(23).gen(1)
(1,17)
```

hecke_matrix(n)

The matrix of the n-th Hecke operator acting on given basis.

EXAMPLE:

```
sage: C = CuspForms(1, 16)
sage: C.hecke_matrix(3)
[-3348]
```

hecke_operator(n)

Returns the n-th Hecke operator T_n.

INPUT:

- `ModularSymbols self` - Hecke equivariant space of modular symbols
- `int n` - an integer at least 1.

EXAMPLES:

```
sage: M = ModularSymbols(11,2)
sage: T = M.hecke_operator(3) ; T
Hecke operator T_3 on Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with
sage: T.matrix()
[ 4  0 -1]
[ 0 -1  0]
[ 0  0 -1]
sage: T(M.0)
4*(1,0) - (1,9)
sage: S = M.cuspidal_submodule()
sage: T = S.hecke_operator(3) ; T
Hecke operator T_3 on Modular Symbols subspace of dimension 2 of Modular Symbols space of di
sage: T.matrix()
[-1  0]
[ 0 -1]
sage: T(S.0)
-(1,8)
```

hecke_polynomial(*n, var='x'*)

Return the characteristic polynomial of the n-th Hecke operator acting on this space.

INPUT:

- n - integer

OUTPUT: a polynomial

EXAMPLE:

```
sage: ModularSymbols(11,2).hecke_polynomial(3)
x^3 - 2*x^2 - 7*x - 4
```

is_simple()

Return True if this space is simple as a module for the corresponding Hecke algebra. Raises NotImplementedError, as this is an abstract base class.

EXAMPLE:

```
sage: sage.modular.hecke.module.HeckeModule_free_module(QQ, 10, 3).is_simple()
...
NotImplementedError
```

is_splittable()

Returns True if and only if only it is possible to split off a nontrivial generalized eigenspace of self as the kernel of some Hecke operator (not necessarily prime to the level). Note that the direct sum of several copies of the same simple module is not splittable in this sense.

EXAMPLE:

```
sage: M = ModularSymbols(Gamma0(64)).cuspidal_subspace()
sage: M.is_splittable()
True
sage: M.simple_factors()[0].is_splittable()
False
```

is_splittable_anemic()

Returns true if and only if only it is possible to split off a nontrivial generalized eigenspace of self as the kernel of some Hecke operator of index coprime to the level. Note that the direct sum of several copies of the same simple module is not splittable in this sense.

EXAMPLE:

```
sage: M = ModularSymbols(Gamma0(64)).cuspidal_subspace()
sage: M.is_splittable_anemic()
```

```
True
sage: M.simple_factors()[0].is_splittable_anemic()
False
```

is_submodule(*other*)

Return True if self is a submodule of other.

EXAMPLES:

```
sage: M = ModularSymbols(Gamma0(64))
sage: M[0].is_submodule(M)
True
sage: CuspForms(1, 24).is_submodule(ModularForms(1, 24))
True
sage: CuspForms(1, 12).is_submodule(CuspForms(3, 12))
False
```

ngens()

Number of generators of self (equal to the rank).

EXAMPLE:

```
sage: ModularForms(1, 12).ngens()
2
```

projection()

Return the projection map from the ambient space to self.

ALGORITHM: Let B be the matrix whose columns are obtained by concatenating together a basis for the factors of the ambient space. Then the projection matrix onto self is the submatrix of B^{-1} obtained from the rows corresponding to self, i.e., if the basis vectors for self appear as columns n through m of B, then the projection matrix is got from rows n through m of B^{-1}. This is because projection with respect to the B basis is just given by an $m - n + 1$ row slice P of a diagonal matrix D with 1's in the n through m positions, so projection with respect to the standard basis is given by $P \cdot B^{-1}$, which is just rows n through m of B^{-1}.

EXAMPLES:

```
sage: e = EllipticCurve('34a')
sage: m = ModularSymbols(34); s = m.cuspidal_submodule()
sage: d = s.decomposition(7)
sage: d
[
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 9 for Gamma_0(
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 9 for Gamma_0(
]
sage: a = d[0]; a
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 9 for Gamma_0(
sage: pi = a.projection()
sage: pi(m([0,oo]))
-1/6*(2,7) + 1/6*(2,13) - 1/6*(2,31) + 1/6*(2,33)
sage: M = ModularSymbols(53,sign=1)
sage: S = M.cuspidal_subspace()[1] ; S
Modular Symbols subspace of dimension 3 of Modular Symbols space of dimension 5 for Gamma_0(
sage: p = S.projection()
sage: S.basis()
((1,33) - (1,37), (1,35), (1,49))
sage: [ p(x) for x in S.basis() ]
[(1,33) - (1,37), (1,35), (1,49)]
```

system_of_eigenvalues(*n, name='alpha'*)

Assuming that self is a simple space of modular symbols, return the eigenvalues $[a_1, \ldots, a_n max]$ of the Hecke operators on self. See `self.eigenvalue(n)` for more details.

INPUT:

- `n` - number of eigenvalues
- `alpha` - name of generate for eigenvalue field

EXAMPLES: These computations use pseudo-random numbers, so we set the seed for reproducible testing.

```
sage: set_random_seed(0)
```

The computations also use cached results from other computations, so we clear the caches for reproducible testing.

```
sage: ModularSymbols_clear_cache()
```

We compute eigenvalues for newforms of level 62.

```
sage: M = ModularSymbols(62,2,sign=-1)
sage: S = M.cuspidal_submodule().new_submodule()
sage: [A.system_of_eigenvalues(3) for A in S.decomposition()]
[[1, 1, 0], [1, -1, -alpha - 1]]
```

Next we define a function that does the above:

```
sage: def b(N,k=2):
...     t=cputime()
...     S = ModularSymbols(N,k,sign=-1).cuspidal_submodule().new_submodule()
...     for A in S.decomposition():
...         print N, A.system_of_eigenvalues(5)

sage: b(63)
63 [1, 1, 0, -1, 2]
63 [1, alpha, 0, 1, -2*alpha]
```

This example illustrates finding field over which the eigenvalues are defined:

```
sage: M = ModularSymbols(23,2,sign=1).cuspidal_submodule().new_submodule()
sage: v = M.system_of_eigenvalues(10); v
[1, alpha, -2*alpha - 1, -alpha - 1, 2*alpha, alpha - 2, 2*alpha + 2, -2*alpha - 1, 2, -2*al
sage: v[0].parent()
Number Field in alpha with defining polynomial x^2 + x - 1
```

This example illustrates setting the print name of the eigenvalue field.

```
sage: A = ModularSymbols(125,sign=1).new_subspace()[0]
sage: A.system_of_eigenvalues(10)
[1, alpha, -alpha - 2, -alpha - 1, 0, -alpha - 1, -3, -2*alpha - 1, 3*alpha + 2, 0]
sage: A.system_of_eigenvalues(10,'x')
[1, x, -x - 2, -x - 1, 0, -x - 1, -3, -2*x - 1, 3*x + 2, 0]
```

weight()

Returns the weight of this Hecke module.

INPUT:

- `self` - an arbitrary Hecke module

OUTPUT:

- `int` - the weight

EXAMPLES:

```
sage: m = ModularSymbols(20, weight=2)
sage: m.weight()
2
```

zero_submodule()
Return the zero submodule of self.

EXAMPLES:

```
sage: ModularSymbols(11,4).zero_submodule()
Modular Symbols subspace of dimension 0 of Modular Symbols space of dimension 6 for Gamma_0(
sage: CuspForms(11,4).zero_submodule()
Modular Forms subspace of dimension 0 of Modular Forms space of dimension 4 for Congruence S
```

class HeckeModule_generic(*base_ring, level*)
A very general base class for Hecke modules.

We define a Hecke module of weight k to be a module over a commutative ring equipped with an action of operators T_m for all positive integers m coprime to some integer n (the level), which satisfy $T_r T_s = T_{rs}$ for r,s coprime, and for powers of a prime p, $T_{p^r} = T_p T_{p^{r-1}} - \varepsilon(p) p^{k-1} T_{p^{r-2}}$, where $\varepsilon(p)$ is some endomorphism of the module which commutes with the T_m.

We distinguish between *full* Hecke modules, which also have an action of operators T_m for m not assumed to be coprime to the level, and *anemic* Hecke modules, for which this does not hold.

anemic_hecke_algebra()
Return the Hecke algebra associated to this Hecke module.

EXAMPLES:

```
sage: T = ModularSymbols(1,12).hecke_algebra()
sage: A = ModularSymbols(1,12).anemic_hecke_algebra()
sage: T == A
False
sage: A
Anemic Hecke algebra acting on Modular Symbols space of dimension 3 for Gamma_0(1) of weight
sage: A.is_anemic()
True
```

category()
Return the category to which this module belongs, i.e. the category of Hecke modules over the given base ring.

EXAMPLE:

```
sage: ModularForms(3, 3).category()
Category of Hecke modules over Rational Field
```

character()
The character of this space. As this is an abstract base class, return None.

EXAMPLE:

```
sage: sage.modular.hecke.module.HeckeModule_generic(QQ, 10).character() is None
True
```

dimension()
Synonym for rank.

EXAMPLE:

```
sage: M = sage.modular.hecke.module.HeckeModule_generic(QQ, 10).dimension()
...
NotImplementedError: Derived subclasses must implement rank
```

hecke_algebra()
Return the Hecke algebra associated to this Hecke module.

EXAMPLES:

```
sage: T = ModularSymbols(Gamma1(5),3).hecke_algebra()
sage: T
Full Hecke algebra acting on Modular Symbols space of dimension 4 for Gamma_1(5) of weight 3
sage: T.is_anemic()
False

sage: M = ModularSymbols(37,sign=1)
sage: E, A, B = M.decomposition()
sage: A.hecke_algebra() == B.hecke_algebra()
False
```

is_full_hecke_module()

Return True if this space is invariant under all Hecke operators.

Since self is guaranteed to be an anemic Hecke module, the significance of this function is that it also ensures invariance under Hecke operators of index that divide the level.

EXAMPLES:

```
sage: M = ModularSymbols(22); M.is_full_hecke_module()
True
sage: M.submodule(M.free_module().span([M.0.list()]), check=False).is_full_hecke_module()
False
```

is_hecke_invariant(n)

Return True if self is invariant under the Hecke operator T_n.

Since self is guaranteed to be an anemic Hecke module it is only interesting to call this function when n is not coprime to the level.

EXAMPLES:

```
sage: M = ModularSymbols(22).cuspidal_subspace()
sage: M.is_hecke_invariant(2)
True
```

We use check=False to create a nasty "module" that is not invariant under T_2:

```
sage: S = M.submodule(M.free_module().span([M.0.list()]), check=False); S
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 7 for Gamma_0(
sage: S.is_hecke_invariant(2)
False
sage: [n for n in range(1,12) if S.is_hecke_invariant(n)]
[1, 3, 5, 7, 9, 11]
```

is_zero()

Return True if this Hecke module has dimension 0.

EXAMPLES:

```
sage: ModularSymbols(11).is_zero()
False
sage: ModularSymbols(11).old_submodule().is_zero()
True
sage: CuspForms(10).is_zero()
True
sage: CuspForms(1,12).is_zero()
False
```

level()

Returns the level of this modular symbols space.

INPUT:

- `ModularSymbols self` - an arbitrary space of modular symbols

OUTPUT:

- int - the level

EXAMPLES:

```
sage: m = ModularSymbols(20)
sage: m.level()
20
```

rank()

Return the rank of this module over its base ring. Returns NotImplementedError, since this is an abstract base class.

EXAMPLES:

```
sage: sage.modular.hecke.module.HeckeModule_generic(QQ, 10).rank()
...
NotImplementedError: Derived subclasses must implement rank
```

submodule(*X*)

Return the submodule of self corresponding to X. As this is an abstract base class, this raises a NotImplementedError.

EXAMPLES:

```
sage: sage.modular.hecke.module.HeckeModule_generic(QQ, 10).submodule(0)
...
NotImplementedError: Derived subclasses should implement submodule
```

is_HeckeModule(*x*)

Return True if x is a Hecke module.

EXAMPLES:

```
sage: from sage.modular.hecke.module import is_HeckeModule
sage: is_HeckeModule(ModularForms(Gamma0(7), 4))
True
sage: is_HeckeModule(QQ^3)
False
sage: is_HeckeModule(J0(37).homology())
True
```

42.2 Ambient Hecke modules

class AmbientHeckeModule(*base_ring, rank, level, weight*)

An ambient Hecke module, i.e. a Hecke module that is isomorphic as a module over its base ring R to the standard free module R^k for some k. This is the base class for ambient spaces of modular forms and modular symbols, and for Brandt modules.

ambient_hecke_module()

Return the ambient space that contains this ambient space. This is, of course, just this space again.

EXAMPLE:

```
sage: M = ModularForms(11, 4); M.ambient_hecke_module() is M
True
```

complement()

Return the largest Hecke-stable complement of this space.

EXAMPLES:

```
sage: M=ModularSymbols(11,2,1)
sage: M
Modular Symbols space of dimension 2 for Gamma_0(11) of weight 2 with sign 1 over Rational F
sage: M.complement()
Modular Symbols subspace of dimension 0 of Modular Symbols space of dimension 2 for Gamma_0(
sage: C=M.cuspidal_subspace()
sage: C
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 for Gamma_0(
sage: C.complement()
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 for Gamma_0(
```

decomposition_matrix()

Returns the matrix whose columns form a basis for the canonical sorted decomposition of self coming from the Hecke operators.

If the simple factors are $D_0, \ldots, D_n$, then the first few columns are an echelonized basis for D_0, the next an echelonized basis for D_1, the next for D_2, etc.

EXAMPLE:

```
sage: S = ModularSymbols(37, 2)
sage: S.decomposition_matrix()
[   1    0    0    0 -1/3]
[   0    1   -1    0  1/2]
[   0    0    0    1 -1/2]
[   0    1    1    1    0]
[   0    0    0    0    1]
```

decomposition_matrix_inverse()

Returns the inverse of the matrix returned by decomposition_matrix().

EXAMPLE:

```
sage: S = ModularSymbols(37, 2)
sage: t = S.decomposition_matrix_inverse(); t
[   1    0    0    0  1/3]
[   0  1/2 -1/2  1/2 -1/2]
[   0 -1/2 -1/2  1/2    0]
[   0    0    1    0  1/2]
[   0    0    0    0    1]
sage: t * S.decomposition_matrix() == 1
True
```

degeneracy_map(*level, t=1*)

The t-th degeneracy map from self to the corresponding Hecke module of the given level. The level of self must be a divisor or multiple of level, and t must be a divisor of the quotient.

INPUT:

- `level` - int, the level of the codomain of the map (positive int).
- `t` - int, the parameter of the degeneracy map, i.e., the map is related to $f(q)$ - $f(q^t)$.

OUTPUT: A morphism from self to corresponding the Hecke module of given level.

EXAMPLES:

```
sage: M = ModularSymbols(11,sign=1)
sage: d1 = M.degeneracy_map(33); d1
Hecke module morphism degeneracy map corresponding to f(q) |--> f(q) defined by the matrix
[ 1  0  0  0 -2 -1]
[ 0  0 -2  2  0  0]
Domain: Modular Symbols space of dimension 2 for Gamma_0(11) of weight ...
Codomain: Modular Symbols space of dimension 6 for Gamma_0(33) of weight ...
sage: M.degeneracy_map(33,3).matrix()
```

```
[ 3  2  2  0 -2  1]
[ 0  2  0 -2  0  0]
sage: M = ModularSymbols(33,sign=1)
sage: d2 = M.degeneracy_map(11); d2.matrix()
[  1   0]
[  0 1/2]
[  0  -1]
[  0   1]
[ -1   0]
[ -1   0]
sage: (d2*d1).matrix()
[4 0]
[0 4]

sage: M = ModularSymbols(3,12,sign=1)
sage: M.degeneracy_map(1)
Hecke module morphism degeneracy map corresponding to f(q) |--> f(q) defined by the matrix
[1 0]
[0 0]
[0 1]
[0 1]
[0 1]
Domain: Modular Symbols space of dimension 5 for Gamma_0(3) of weight ...
Codomain: Modular Symbols space of dimension 2 for Gamma_0(1) of weight ...

sage: S = M.cuspidal_submodule()
sage: S.degeneracy_map(1)
Hecke module morphism defined by the matrix
[1 0]
[0 0]
[0 0]
Domain: Modular Symbols subspace of dimension 3 of Modular Symbols space ...
Codomain: Modular Symbols space of dimension 2 for Gamma_0(1) of weight ...

sage: D = ModularSymbols(10,4).cuspidal_submodule().decomposition()
sage: D
[
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 10 for Gamma_0
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 10 for Gamma_0
]
sage: D[1].degeneracy_map(5)
Hecke module morphism defined by the matrix
[   0    0   -1    1]
[   0  1/2  3/2   -2]
[   0   -1    1    0]
[   0 -3/4 -1/4    1]
Domain: Modular Symbols subspace of dimension 4 of Modular Symbols space ...
Codomain: Modular Symbols space of dimension 4 for Gamma_0(5) of weight ...
```

dual_free_module()

The free module dual to self, as a submodule of the dual module of the ambient space. As this space is ambient anyway, this just returns self.free_module().

EXAMPLE:

```
sage: M = ModularForms(2,8); M.dual_free_module()
Vector space of dimension 3 over Rational Field
sage: M.dual_free_module() is M.free_module()
True
```

fcp(*n, var='x'*)

Returns the factorization of the characteristic polynomial of the Hecke operator T_n of index n acting on this space.

INPUT:

- `self` - Hecke module invariant under the Hecke operator of index n.
- `int n` - a positive integer.
- `var` - variable of polynomial (default x)

OUTPUT:

- `list` - list of the pairs (g,e), where g is an irreducible factor of the characteristic polynomial of T_n, and e is its multiplicity.

EXAMPLES:

```
sage: m = ModularSymbols(23, 2, sign=1)
sage: m.fcp(2)
(x - 3) * (x^2 + x - 1)
sage: m.hecke_operator(2).charpoly('x').factor()
(x - 3) * (x^2 + x - 1)
```

free_module()

Return the free module underlying this ambient Hecke module (the forgetful functor from Hecke modules to modules over the base ring)

EXAMPLE:

```
sage: ModularForms(59, 2).free_module()
Vector space of dimension 6 over Rational Field
```

hecke_bound()

Return an integer B such that the Hecke operators T_n, for $n \leq B$, generate the full Hecke algebra as a module over the base ring. Note that we include the n with n not coprime to the level.

At present this returns an unproven guess which appears to be valid for $M_k(\Gamma_0(N))$, where k and N are the weight and level of self. (It is clearly valid for *cuspidal* spaces of any fixed character, as a consequence of the Sturm bound theorem.) It returns a hopelessly wrong answer for spaces of full level Γ_1.

TODO: Get rid of this dreadful bit of code.

EXAMPLE:

```
sage: ModularSymbols(17, 4).hecke_bound()
15
sage: ModularSymbols(Gamma1(17), 4).hecke_bound() # wrong!
15
```

hecke_images(*i, v*)

Return images of the i-th standard basis vector under the Hecke operators T_p for all integers in v.

INPUT:

- `i` - nonnegative integer
- `v` - a list of positive integer

OUTPUT:

- `matrix` - whose rows are the Hecke images

EXAMPLES:

```
sage: M = ModularSymbols(DirichletGroup(13).0, 3)
sage: M.T(2)(M.0).element()
(zeta12 + 4, 0, -1, 1)
sage: M.hecke_images(0, [1,2])
[         1          0          0          0]
[zeta12 + 4          0         -1          1]
```

hecke_module_of_level(*level*)

Return the Hecke module corresponding to self at the given level, which should be either a divisor or a multiple of the level of self. This raises NotImplementedError, and should be overridden in derived classes.

EXAMPLE:

```
sage: sage.modular.hecke.ambient_module.AmbientHeckeModule.hecke_module_of_level(ModularForm
...
NotImplementedError
```

intersection(*other*)

Returns the intersection of self and other, which must both lie in a common ambient space of modular symbols.

EXAMPLES:

```
sage: M = ModularSymbols(43, sign=1)
sage: A = M[0] + M[1]
sage: B = M[1] + M[2]
sage: A.rank(), B.rank()
(2, 3)
sage: C = A.intersection(B); C.rank()  # TODO
1
```

is_ambient()

Returns True if and only if self is an ambient Hecke module.

Warning: self can only be ambient by being of type AmbientHeckeModule.
For example, decomposing a simple ambient space yields a single factor, and that factor is *not* considered an ambient space.

EXAMPLES:

```
sage: m = ModularSymbols(10)
sage: m.is_ambient()
True

sage: a = m[0]  # the unique simple factor
sage: a == m
True
sage: a.is_ambient()
False
```

is_full_hecke_module(*compute=True*)

Returns True if this space is invariant under the action of all Hecke operators, even those that divide the level. This is always true for ambient Hecke modules, so return True.

EXAMPLE:

```
sage: ModularSymbols(11, 4).is_full_hecke_module()
True
```

is_new(*p=None*)

Return True if this module is entirely new.

EXAMPLE:

```
sage: ModularSymbols(11, 4).is_new()
False
sage: ModularSymbols(1, 12).is_new()
True
```

is_old(*p=None*)

Return True if this module is entirely old.

EXAMPLE:

```
sage: ModularSymbols(22).is_old()
True
sage: ModularSymbols(3, 12).is_old()
False
```

is_submodule(*V*)

Returns True if and only if self is a submodule of V. Since this is an ambient space, this returns True if and only if V is equal to self.

EXAMPLE:

```
sage: ModularSymbols(1, 4).is_submodule(ModularSymbols(11,4))
False
sage: ModularSymbols(11, 4).is_submodule(ModularSymbols(11,4))
True
```

linear_combination_of_basis(*v*)

Given a list or vector of length equal to the dimension of self, construct the appropriate linear combination of the basis vectors of self.

EXAMPLE:

```
sage: ModularForms(3, 12).linear_combination_of_basis([1,0,0,0,1])
2*q + 2049*q^2 + 177147*q^3 + 4196177*q^4 + 48830556*q^5 + O(q^6)
```

new_submodule(*p=None*)

Returns the new or p-new submodule of self.

INPUT:

- p - (default: None); if not None, return only the p-new submodule.

OUTPUT: the new or p-new submodule of self

EXAMPLES:

```
sage: m = ModularSymbols(33); m.rank()
9
sage: m.new_submodule().rank()
3
sage: m.new_submodule(3).rank()
4
sage: m.new_submodule(11).rank()
8
```

nonembedded_free_module()

Return the free module corresponding to self as an abstract free module (rather than as a submodule of an ambient free module). As this module is ambient anyway, this just returns `self.free_module()`.

EXAMPLES:

```
sage: M = ModularSymbols(11, 2)
sage: M.nonembedded_free_module() is M.free_module()
True
```

old_submodule(*p=None*)

Returns the old or p-old submodule of self.

INPUT:

- p - (default: None); if not None, return only the p-old submodule.

OUTPUT: the old or p-old submodule of self

EXAMPLES:

```
sage: m = ModularSymbols(33); m.rank()
9
sage: m.old_submodule().rank()
7
sage: m.old_submodule(3).rank()
6
sage: m.new_submodule(11).rank()
8

sage: e = DirichletGroup(16)([-1, 1])
sage: M = ModularSymbols(e, 3, sign=1); M
Modular Symbols space of dimension 4 and level 16, weight 3, character [-1, 1], sign 1, over
sage: M.old_submodule()
Modular Symbols subspace of dimension 3 of Modular Symbols space of dimension 4 and level 16
```

rank()

Return the rank of this ambient Hecke module.

OUTPUT:

Integer

EXAMPLES:

```
sage: M = sage.modular.hecke.ambient_module.AmbientHeckeModule(QQ, 3, 11, 2); M
Generic ambient Hecke module of rank 3, level 11 and weight 2 over Rational Field
sage: M.rank()
3
```

submodule(*M, Mdual=None, check=True*)

Return the Hecke submodule of self generated by M, which may be a submodule of the free module of self, or a list of elements of self.

EXAMPLE:

```
sage: M = ModularForms(37, 2)
sage: A = M.submodule([M.newforms()[0].element(), M.newforms()[1].element()]); A
Modular Forms subspace of dimension 2 of Modular Forms space of dimension 3 for Congruence S
```

submodule_from_nonembedded_module(*V, Vdual=None, check=True*)

Create a submodule of this module, from a submodule of an ambient free module of the same rank as the rank of self.

INPUT:

- `V` - submodule of ambient free module of the same rank as the rank of self.
- `Vdual` - used to pass in dual submodule (may be None)
- `check` - whether to check that submodule is Hecke equivariant

OUTPUT: Hecke submodule of self

EXAMPLE:

```
sage: V = QQ^8
sage: ModularForms(24, 2).submodule_from_nonembedded_module(V.submodule([0]))
Modular Forms subspace of dimension 0 of Modular Forms space of dimension 8 for Congruence S
```

submodule_generated_by_images(*M*)

Return the submodule of this ambient modular symbols space generated by the images under all degeneracy maps of M. The space M must have the same weight, sign, and group or character as this ambient space.

EXAMPLES:

```
sage: ModularSymbols(6, 12).submodule_generated_by_images(ModularSymbols(1,12))
Modular Symbols subspace of dimension 12 of Modular Symbols space of dimension 22 for Gamma_
```

is_AmbientHeckeModule(*x*)

Return True if x is of type AmbientHeckeModule.

EXAMPLES:

```
sage: from sage.modular.hecke.ambient_module import is_AmbientHeckeModule
sage: is_AmbientHeckeModule(ModularSymbols(6))
True
sage: is_AmbientHeckeModule(ModularSymbols(6).cuspidal_subspace())
False
sage: is_AmbientHeckeModule(ModularForms(11))
True
sage: is_AmbientHeckeModule(BrandtModule(2, 3))
True
```

42.3 Submodules of Hecke modules

class HeckeSubmodule(*ambient, submodule, dual_free_module=None, check=True*)

Submodule of a Hecke module.

ambient()

Synonym for ambient_hecke_module.

EXAMPLES:

```
sage: CuspForms(2, 12).ambient()
Modular Forms space of dimension 4 for Congruence Subgroup Gamma0(2) of weight 12 over Ratic
```

ambient_hecke_module()

Return the ambient Hecke module of which this is a submodule.

EXAMPLES:

```
sage: CuspForms(2, 12).ambient_hecke_module()
Modular Forms space of dimension 4 for Congruence Subgroup Gamma0(2) of weight 12 over Ratic
```

complement(*bound=None*)

Return the largest Hecke-stable complement of this space.

EXAMPLES:

```
sage: M = ModularSymbols(15, 6).cuspidal_subspace()
sage: M.complement()
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 20 for Gamma_0
sage: E = EllipticCurve("128a")
sage: ME = E.modular_symbol_space()
sage: ME.complement()
Modular Symbols subspace of dimension 17 of Modular Symbols space of dimension 18 for Gamma_
```

degeneracy_map(*level, t=1*)

The t-th degeneracy map from self to the space of ambient modular symbols of the given level. The level of self must be a divisor or multiple of level, and t must be a divisor of the quotient.

INPUT:

- `level` - int, the level of the codomain of the map (positive int).
- `t` - int, the parameter of the degeneracy map, i.e., the map is related to $f(q)$ - $f(q^t)$.

OUTPUT: A linear function from self to the space of modular symbols of given level with the same weight, character, sign, etc., as this space.

EXAMPLES:

```
sage: D = ModularSymbols(10,4).cuspidal_submodule().decomposition(); D
[
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 10 for Gamma_0
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 10 for Gamma_0
]
sage: d = D[1].degeneracy_map(5); d
Hecke module morphism defined by the matrix
[   0    0   -1    1]
[   0  1/2  3/2   -2]
[   0   -1    1    0]
[   0 -3/4 -1/4    1]
Domain: Modular Symbols subspace of dimension 4 of Modular Symbols space ...
Codomain: Modular Symbols space of dimension 4 for Gamma_0(5) of weight ...

sage: d.rank()
2
sage: d.kernel()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 10 for Gamma_0
sage: d.image()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 4 for Gamma_0(
```

dual_free_module(*bound=None, anemic=True, use_star=True*)

Compute embedded dual free module if possible. In general this won't be possible, e.g., if this space is not Hecke equivariant, possibly if it is not cuspidal, or if the characteristic is not 0. In all these cases we raise a RuntimeError exception.

If use_star is True (which is the default), we also use the +/- eigenspaces for the star operator to find the dual free module of self. If the self does not have a star involution, use_star will automatically be set to True.

EXAMPLES:

```
sage: M = ModularSymbols(11, 2)
sage: M.dual_free_module()
Vector space of dimension 3 over Rational Field
sage: Mpc = M.plus_submodule().cuspidal_submodule()
sage: Mcp = M.cuspidal_submodule().plus_submodule()
sage: Mcp.dual_free_module() == Mpc.dual_free_module()
True
sage: Mpc.dual_free_module()
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[  1 5/2   5]

sage: M = ModularSymbols(35,2).cuspidal_submodule()
sage: M.dual_free_module(use_star=False)
Vector space of degree 9 and dimension 6 over Rational Field
Basis matrix:
[   1    0    0    0   -1    0    0    4   -2]
[   0    1    0    0    0    0    0 -1/2  1/2]
[   0    0    1    0    0    0    0 -1/2  1/2]
[   0    0    0    1   -1    0    0    1    0]
[   0    0    0    0    0    1    0   -2    1]
[   0    0    0    0    0    0    1   -2    1]

sage: M = ModularSymbols(40,2)
```

```
sage: Mmc = M.minus_submodule().cuspidal_submodule()
sage: Mcm = M.cuspidal_submodule().minus_submodule()
sage: Mcm.dual_free_module() == Mmc.dual_free_module()
True
sage: Mcm.dual_free_module()
Vector space of degree 13 and dimension 3 over Rational Field
Basis matrix:
[ 0  1  0  0  0  0  1  0 -1 -1  1 -1  0]
[ 0  0  1  0 -1  0 -1  0  1  0  0  0  0]
[ 0  0  0  0  0  1  1  0 -1  0  0  0  0]

sage: M = ModularSymbols(43).cuspidal_submodule()
sage: S = M[0].plus_submodule() + M[1].minus_submodule()
sage: S.dual_free_module(use_star=False)
...
RuntimeError: Computation of embedded dual vector space failed (cut down to rank 6, but shou
sage: S.dual_free_module().dimension() == S.dimension()
True
```

free_module()

Return the free module corresponding to self.

EXAMPLES:

```
sage: M = ModularSymbols(33,2).cuspidal_subspace() ; M
Modular Symbols subspace of dimension 6 of Modular Symbols space of dimension 9 for Gamma_0(
sage: M.free_module()
Vector space of degree 9 and dimension 6 over Rational Field
Basis matrix:
[ 0  1  0  0  0  0  0 -1  1]
[ 0  0  1  0  0  0  0 -1  1]
[ 0  0  0  1  0  0  0 -1  1]
[ 0  0  0  0  1  0  0 -1  1]
[ 0  0  0  0  0  1  0 -1  1]
[ 0  0  0  0  0  0  1 -1  0]
```

intersection(*other*)

Returns the intersection of self and other, which must both lie in a common ambient space of modular symbols.

EXAMPLES:

```
sage: M = ModularSymbols(43, sign=1)
sage: A = M[0] + M[1]
sage: B = M[1] + M[2]
sage: A.dimension(), B.dimension()
(2, 3)
sage: C = A.intersection(B); C.dimension()
1
```

TESTS:

```
sage: M = ModularSymbols(1,80)
sage: M.plus_submodule().cuspidal_submodule().sign() # indirect doctest
1
```

is_ambient()

Return `True` if self is an ambient space of modular symbols.

EXAMPLES:

```
sage: M = ModularSymbols(17,4)
sage: M.cuspidal_subspace().is_ambient()
False
sage: A = M.ambient_hecke_module()
sage: S = A.submodule(A.basis())
sage: sage.modular.hecke.submodule.HeckeSubmodule.is_ambient(S)
True
```

is_new(*p=None*)

Returns True if this Hecke module is p-new. If p is None, returns True if it is new.

EXAMPLES:

```
sage: M = ModularSymbols(1,16)
sage: S = sage.modular.hecke.submodule.HeckeSubmodule(M, M.cuspidal_submodule().free_module(
sage: S.is_new()
True
```

is_old(*p=None*)

Returns True if this Hecke module is p-old. If p is None, returns True if it is old.

EXAMPLES:

```
sage: M = ModularSymbols(50,2)
sage: S = sage.modular.hecke.submodule.HeckeSubmodule(M, M.old_submodule().free_module())
sage: S.is_old()
True
sage: S = sage.modular.hecke.submodule.HeckeSubmodule(M, M.new_submodule().free_module())
sage: S.is_old()
False
```

is_submodule(*V*)

Returns True if and only if self is a submodule of V.

EXAMPLES:

```
sage: M = ModularSymbols(30,4)
sage: S = sage.modular.hecke.submodule.HeckeSubmodule(M, M.cuspidal_submodule().free_module(
sage: S.is_submodule(M)
True
sage: SS = sage.modular.hecke.submodule.HeckeSubmodule(M, M.old_submodule().free_module())
sage: S.is_submodule(SS)
False
```

linear_combination_of_basis(*v*)

Return the linear combination of the basis of self given by the entries of v.

EXAMPLES:

```
sage: M = ModularForms(Gamma0(2),12)
sage: S = sage.modular.hecke.submodule.HeckeSubmodule(M, M.cuspidal_submodule().free_module(
sage: S.basis()
(q + 252*q^3 - 2048*q^4 + 4830*q^5 + O(q^6), q^2 - 24*q^4 + O(q^6))
sage: S.linear_combination_of_basis([3,10])
3*q + 10*q^2 + 756*q^3 - 6384*q^4 + 14490*q^5 + O(q^6)
```

module()

Alias for code{self.free_module()}.

EXAMPLES:

```
sage: M = ModularSymbols(17,4).cuspidal_subspace()
sage: M.free_module() is M.module()
True
```

new_submodule(*p=None*)

Return the new or p-new submodule of this space of modular symbols.

EXAMPLES:

```
sage: M = ModularSymbols(20,4)
sage: M.new_submodule()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 18 for Gamma_0
sage: S = sage.modular.hecke.submodule.HeckeSubmodule(M, M.cuspidal_submodule().free_module(
sage: S
Rank 12 submodule of a Hecke module of level 20
sage: S.new_submodule()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 18 for Gamma_0
```

nonembedded_free_module()

Return the free module corresponding to self as an abstract free module, i.e. not as an embedded vector space.

EXAMPLES:

```
sage: M = ModularSymbols(12,6)
sage: S = sage.modular.hecke.submodule.HeckeSubmodule(M, M.cuspidal_submodule().free_module(
sage: S
Rank 14 submodule of a Hecke module of level 12
sage: S.nonembedded_free_module()
Vector space of dimension 14 over Rational Field
```

old_submodule(*p=None*)

Return the old or p-old submodule of this space of modular symbols.

EXAMPLES: We compute the old and new submodules of $\mathbf{S}_2(\Gamma_0(33))$.

```
sage: M = ModularSymbols(33); S = M.cuspidal_submodule(); S
Modular Symbols subspace of dimension 6 of Modular Symbols space of dimension 9 for Gamma_0(
sage: S.old_submodule()
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 9 for Gamma_0(
sage: S.new_submodule()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 9 for Gamma_0(
```

rank()

Return the rank of self as a free module over the base ring.

EXAMPLE:

```
sage: ModularSymbols(6, 4).cuspidal_subspace().rank()
2
sage: ModularSymbols(6, 4).cuspidal_subspace().dimension()
2
```

submodule(*M, Mdual=None, check=True*)

Construct a submodule of self from the free module M, which must be a subspace of self.

EXAMPLES:

```
sage: M = ModularSymbols(18,4)
sage: S = sage.modular.hecke.submodule.HeckeSubmodule(M, M.cuspidal_submodule().free_module(
sage: S[0]
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 18 for Gamma_0
sage: S.submodule(S[0].free_module())
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 18 for Gamma_0
```

submodule_from_nonembedded_module(*V, Vdual=None, check=True*)

Construct a submodule of self from V. Here V should be a subspace of a vector space whose dimension is the same as that of self.

INPUT:

- V - submodule of ambient free module of the same rank as the rank of self.
- check - whether to check that V is Hecke equivariant.

OUTPUT: Hecke submodule of self

EXAMPLES:

```
sage: M = ModularSymbols(37,2)
sage: S = sage.modular.hecke.submodule.HeckeSubmodule(M, M.cuspidal_submodule().free_module(
sage: V = (QQ**4).subspace([[1,-1,0,1/2],[0,0,1,-1/2]])
sage: S.submodule_from_nonembedded_module(V)
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 5 for Gamma_0(
```

is_HeckeSubmodule(*x*)

Return True if x is of type HeckeSubmodule.

EXAMPLES:

```
sage: sage.modular.hecke.submodule.is_HeckeSubmodule(ModularForms(1, 12))
False
sage: sage.modular.hecke.submodule.is_HeckeSubmodule(CuspForms(1, 12))
True
```

42.4 Elements of Hecke modules

AUTHORS:

- William Stein

class HeckeModuleElement(*parent, x=None*)

Element of a Hecke module.

ambient_module()

Return the ambient Hecke module that contains this element.

EXAMPLES:

```
sage: BrandtModule(37)([0,1,-1]).ambient_module()
Brandt module of dimension 3 of level 37 of weight 2 over Rational Field
```

element()

Return underlying vector space element that defines this Hecke module element.

EXAMPLES:

```
sage: z = BrandtModule(37)([0,1,-1]).element(); z
(0, 1, -1)
sage: type(z)
<type 'sage.modules.vector_rational_dense.Vector_rational_dense'>
```

is_cuspidal()

Return True if this element is cuspidal.

EXAMPLES:

```
sage: M = ModularForms(2, 22); M.0.is_cuspidal()
True
sage: (M.0 + M.4).is_cuspidal()
False
sage: EllipticCurve('37a1').newform().is_cuspidal()
True
```

It works for modular symbols too:

```
sage: M = ModularSymbols(19,2)
sage: M.0.is_cuspidal()
False
sage: M.1.is_cuspidal()
True
```

is_eisenstein()

Return True if this element is Eisenstein. This makes sense for both modular forms and modular symbols.

EXAMPLES:

```
sage: CuspForms(2,8).0.is_eisenstein()
False
sage: M = ModularForms(2,8);(M.0  + M.1).is_eisenstein()
False
sage: M.1.is_eisenstein()
True
sage: ModularSymbols(19,4).0.is_eisenstein()
False
sage: EllipticCurve('37a1').newform().is_eisenstein()
False
```

is_new(*p=None*)

Return True if this element is p-new. If p is None, return True if the element is new.

EXAMPLE:

```
sage: CuspForms(22, 2).0.is_new(2)
False
sage: CuspForms(22, 2).0.is_new(11)
True
sage: CuspForms(22, 2).0.is_new()
False
```

is_old(*p=None*)

Return True if this element is p-old. If p is None, return True if the element is old.

EXAMPLE:

```
sage: CuspForms(22, 2).0.is_old(11)
False
sage: CuspForms(22, 2).0.is_old(2)
True
sage: CuspForms(22, 2).0.is_old()
True
sage: EisensteinForms(144, 2).1.is_old()
False
sage: EisensteinForms(144, 2).1.is_old(2) # not implemented
False
```

is_HeckeModuleElement(*x*)

Return True if x is a Hecke module element, i.e., of type HeckeModuleElement.

EXAMPLES:

```
sage: sage.modular.hecke.all.is_HeckeModuleElement(0)
False
sage: sage.modular.hecke.all.is_HeckeModuleElement(BrandtModule(37)([1,2,3]))
True
```

42.5 Hom spaces between Hecke modules

class HeckeModuleHomspace(*X, Y*)
A space of homomorphisms between two objects in the category of Hecke modules over a given base ring.

is_HeckeModuleHomspace(*x*)
Return True if x is a space of homomorphisms in the category of Hecke modules.

EXAMPLES:

```
sage: M = ModularForms(Gamma0(7), 4)
sage: sage.modular.hecke.homspace.is_HeckeModuleHomspace(Hom(M, M))
True
sage: sage.modular.hecke.homspace.is_HeckeModuleHomspace(Hom(M, QQ))
False
```

42.6 Morphisms of Hecke modules

AUTHORS:

- William Stein

class HeckeModuleMorphism()
Abstract base class for morphisms of Hecke modules.

class HeckeModuleMorphism_matrix(*parent, A, name=''*)
Morphisms of Hecke modules when the morphism is given by a matrix.

Note that care is needed when composing morphisms, because morphisms in Sage act on the left, but their matrices act on the right (!). So if F: A -> B and G : B -> C are morphisms, the composition A -> C is G*F, but its matrix is F.matrix() * G.matrix().

EXAMPLE:

```
sage: A = ModularForms(1, 4)
sage: B = ModularForms(1, 16)
sage: C = ModularForms(1, 28)
sage: F = A.Hom(B)(matrix(QQ,1,2,srange(1, 3)))
sage: G = B.Hom(C)(matrix(QQ,2,3,srange(1, 7)))
sage: G * F
Hecke module morphism defined by the matrix
[ 9 12 15]
Domain: Modular Forms space of dimension 1 for Modular Group SL(2,Z) ...
Codomain: Modular Forms space of dimension 3 for Modular Group SL(2,Z) ...
sage: F * G
...
TypeError: Incompatible composition of morphisms: domain of left morphism must be codomain of ri
```

name(*new=None*)
Return the name of this operator, or set it to a new name.

EXAMPLES:

```
sage: M = ModularSymbols(6)
sage: t = M.Hom(M)(matrix(QQ,3,3,srange(9)), name="spam"); t
Hecke module morphism spam defined by ...
sage: t.name()
```

```
'spam'
sage: t.name("eggs"); t
Hecke module morphism eggs defined by ...
```

is_HeckeModuleMorphism(*x*)

Return True if x is of type HeckeModuleMorphism.

EXAMPLES:

```
sage: sage.modular.hecke.morphism.is_HeckeModuleMorphism(ModularSymbols(6).hecke_operator(7).hec
True
```

is_HeckeModuleMorphism_matrix(*x*)

EXAMPLES:

```
sage: sage.modular.hecke.morphism.is_HeckeModuleMorphism_matrix(ModularSymbols(6).hecke_operator
True
```

42.7 Degeneracy maps

class DegeneracyMap(*matrix, domain, codomain, t*)

A degeneracy map between Hecke modules of different levels.

EXAMPLES: We construct a number of degeneracy maps:

```
sage: M = ModularSymbols(33)
sage: d = M.degeneracy_map(11)
sage: d
Hecke module morphism degeneracy map corresponding to f(q) |--> f(q) defined by the matrix
[ 1  0  0]
[ 0  0  1]
[ 0  0 -1]
[ 0  1 -1]
[ 0  0  1]
[ 0 -1  1]
[-1  0  0]
[-1  0  0]
[-1  0  0]
Domain: Modular Symbols space of dimension 9 for Gamma_0(33) of weight ...
Codomain: Modular Symbols space of dimension 3 for Gamma_0(11) of weight ...
sage: d.t()
1
sage: d = M.degeneracy_map(11,3)
sage: d.t()
3
```

The parameter d must be a divisor of the quotient of the two levels:

```
sage: d = M.degeneracy_map(11,2)
...
ValueError: The level of self (=33) must be a divisor or multiple of level (=11), and t (=2) mus
```

Degeneracy maps can also go from lower level to higher level:

```
sage: M.degeneracy_map(66,2)
Hecke module morphism degeneracy map corresponding to f(q) |--> f(q^2) defined by the matrix
[ 2  0  0  0  0  0  1  0  0  0  1 -1  0  0  0 -1  1  0  0  0  0  0  0 -1]
[ 0  0  1 -1  0 -1  1  0 -1  2  0  0  0 -1  0  0 -1  1  2 -2  0  0  0 -1  1]
[ 0  0  1  0  0  0  0  0  1  0  0  0  1  0  0  0 -1  1  0  0 -1  1  0  0  0]
[ 0  0  0  0  0  0  0  0  0  2 -1  0  0  1  0  0 -1  1  0  0  1  0 -1 -1  1]
[ 0 -1  0  0  1  0  0  0  0  0  0  1  0  0  1  1 -1  0  0 -1  0  0  0  0  0]
[ 0  0  0  0  0  0  0  1 -1  0  0  2 -1  0  0  1  0  0  0 -1  0 -1  1 -1  1]
[ 0  0  0  0  1 -1  0  1 -1  0  0  0  0  0 -1  2  0  0  0  0  1  0  1  0  0]
[ 0  0  1  0  0  0  0  0  0  0  0  0  0  0  0  0  0  1  0  0  0  1  1  0  0]
[ 0  0  0  0  0  0  0  0  0  0 -1  0  0  0  0  0  0  0  0  1  1  1  0  0  0]
Domain: Modular Symbols space of dimension 9 for Gamma_0(33) of weight ...
Codomain: Modular Symbols space of dimension 25 for Gamma_0(66) of weight ...
```

t()

Return the divisor of the quotient of the two levels associated to the degeneracy map.

EXAMPLES:

```
sage: M = ModularSymbols(33)
sage: d = M.degeneracy_map(11,3)
sage: d.t()
3
sage: d = M.degeneracy_map(11,1)
sage: d.t()
1
```

42.8 Hecke algebras

In Sage a "Hecke algebra" always refers to an algebra of endomorphisms of some explicit module, rather than the abstract Hecke algebra of double cosets attached to a subgroup of the modular group.

We distinguish between "anemic Hecke algebras", which are algebras of Hecke operators whose indices do not divide some integer N (the level), and "full Hecke algebras", which include Hecke operators coprime to the level. Morphisms in the category of Hecke modules are not required to commute with the action of the full Hecke algebra, only with the anemic algebra.

AnemicHeckeAlgebra(*M*)

Return the anemic Hecke algebra associated to the Hecke module M. This checks whether or not the object already exists in memory, and if so, returns the existing object rather than a new one.

EXAMPLES:

```
sage: CuspForms(1, 12).anemic_hecke_algebra() # indirect doctest
Anemic Hecke algebra acting on Cuspidal subspace of dimension 1 of Modular Forms space of dimens
```

We test uniqueness:

```
sage: CuspForms(1, 12).anemic_hecke_algebra() is CuspForms(1, 12).anemic_hecke_algebra()
True
```

We can't ensure uniqueness when loading and saving objects from files, but we can ensure equality:

```
sage: CuspForms(1, 12).anemic_hecke_algebra() is loads(dumps(CuspForms(1, 12).anemic_hecke_algeb
False
sage: CuspForms(1, 12).anemic_hecke_algebra() == loads(dumps(CuspForms(1, 12).anemic_hecke_algeb
True
```

HeckeAlgebra(*M*)
Return the full Hecke algebra associated to the Hecke module M. This checks whether or not the object already exists in memory, and if so, returns the existing object rather than a new one.

EXAMPLES:

```
sage: CuspForms(1, 12).hecke_algebra() # indirect doctest
Full Hecke algebra acting on Cuspidal subspace of dimension 1 of Modular Forms space of dimensic
```

We test uniqueness:

```
sage: CuspForms(1, 12).hecke_algebra() is CuspForms(1, 12).hecke_algebra()
True
```

We can't ensure uniqueness when loading and saving objects from files, but we can ensure equality:

```
sage: CuspForms(1, 12).hecke_algebra() is loads(dumps(CuspForms(1, 12).hecke_algebra()))
False
sage: CuspForms(1, 12).hecke_algebra() == loads(dumps(CuspForms(1, 12).hecke_algebra()))
True
```

class **HeckeAlgebra_anemic**(*M*)
An anemic Hecke algebra, generated by Hecke operators with index coprime to the level.

gens()
Return a generator over all Hecke operator T_n for $n = 1, 2, 3, \ldots$, with n coprime to the level. This is an infinite sequence.

EXAMPLES:

```
sage: T = ModularSymbols(12,2).anemic_hecke_algebra()
sage: g = T.gens()
sage: g.next()
Hecke operator T_1 on Modular Symbols space of dimension 5 for Gamma_0(12) of weight 2 with
sage: g.next()
Hecke operator T_5 on Modular Symbols space of dimension 5 for Gamma_0(12) of weight 2 with
```

hecke_operator(*n*)
Return the n-th Hecke operator, for n any positive integer coprime to the level.

EXAMPLES:

```
sage: T = ModularSymbols(Gamma1(5),3).anemic_hecke_algebra()
sage: T.hecke_operator(2)
Hecke operator T_2 on Modular Symbols space of dimension 4 for Gamma_1(5) of weight 3 with s
sage: T.hecke_operator(5)
...
IndexError: Hecke operator T_5 not defined in the anemic Hecke algebra
```

is_anemic()
Return True, since this the anemic Hecke algebra.

EXAMPLES:

```
sage: H = CuspForms(3, 12).anemic_hecke_algebra()
sage: H.is_anemic()
True
```

class **HeckeAlgebra_base**(*M*)
Base class for algebras of Hecke operators on a fixed Hecke module.

basis()
Return a basis for this Hecke algebra as a free module over its base ring. Not implemented at present.

EXAMPLE:

```
sage: ModularSymbols(Gamma1(3), 3).hecke_algebra().basis()
...
NotImplementedError
```

discriminant()

Return the discriminant of this Hecke algebra, i.e. the determinant of the matrix $\mathrm{Tr}(x_i x_j)$ where $x_1, \ldots, x_d$ is a basis for self, and $\mathrm{Tr}(x)$ signifies the trace (in the sense of linear algebra) of left multiplication by x on the algebra (*not* the trace of the operator x acting on the underlying Hecke module!). For further discussion and conjectures see Calegari + Stein, *Conjectures about discriminants of Hecke algebras of prime level*, Springer LNCS 3076.

Not implemented at present.

EXAMPLE:

```
sage: BrandtModule(3, 4).hecke_algebra().discriminant()
...
NotImplementedError
```

gen(n)

Return the n-th Hecke operator.

EXAMPLES:

```
sage: T = ModularSymbols(11).hecke_algebra()
sage: T.gen(2)
Hecke operator T_2 on Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with
```

gens()

Return a generator over all Hecke operator T_n for $n = 1, 2, 3, \ldots$. This is infinite.

EXAMPLES:

```
sage: T = ModularSymbols(1,12).hecke_algebra()
sage: g = T.gens()
sage: g.next()
Hecke operator T_1 on Modular Symbols space of dimension 3 for Gamma_0(1) of weight 12 with
sage: g.next()
Hecke operator T_2 on Modular Symbols space of dimension 3 for Gamma_0(1) of weight 12 with
```

hecke_matrix(n)

Return the matrix of the n-th Hecke operator T_n.

EXAMPLES:

```
sage: T = ModularSymbols(1,12).hecke_algebra()
sage: T.hecke_matrix(2)
[ -24    0    0]
[   0  -24    0]
[4860    0 2049]
```

hecke_operator(n)

Return the n-th Hecke operator T_n.

EXAMPLES:

```
sage: T = ModularSymbols(1,12).hecke_algebra()
sage: T.hecke_operator(2)
Hecke operator T_2 on Modular Symbols space of dimension 3 for Gamma_0(1) of weight 12 with
```

is_noetherian()

Return True if this Hecke algebra is Noetherian as a ring. This is true if and only if the base ring is Noetherian.

EXAMPLES:

```
sage: CuspForms(1, 12).anemic_hecke_algebra().is_noetherian()
True
```

level()
Return the level of this Hecke algebra, which is (by definition) the level of the Hecke module on which it acts.
EXAMPLE:

```
sage: ModularSymbols(37).hecke_algebra().level()
37
```

matrix_space()
Return the underlying matrix space of this module.
EXAMPLES:

```
sage: CuspForms(3, 24, base_ring=Qp(5)).anemic_hecke_algebra().matrix_space()
Full MatrixSpace of 7 by 7 dense matrices over 5-adic Field with capped relative precision 2
```

module()
The Hecke module on which this algebra is acting.
EXAMPLES:

```
sage: T = ModularSymbols(1,12).hecke_algebra()
sage: T.module()
Modular Symbols space of dimension 3 for Gamma_0(1) of weight 12 with sign 0 over Rational F
```

ngens()
The size of the set of generators returned by gens(), which is clearly infinity. (This is not necessarily a minimal set of generators.)
EXAMPLES:

```
sage: CuspForms(1, 12).anemic_hecke_algebra().ngens()
+Infinity
```

rank()
The rank of this Hecke algebra as a module over its base ring. Not implemented at present.
EXAMPLE:

```
sage: ModularSymbols(Gamma1(3), 3).hecke_algebra().rank()
...
NotImplementedError
```

class **HeckeAlgebra_full**(*M*)
A full Hecke algebra (including the operators T_n where n is not assumed to be coprime to the level).

anemic_subalgebra()
The subalgebra of self generated by the Hecke operators of index coprime to the level.
EXAMPLE:

```
sage: H = CuspForms(3, 12).hecke_algebra()
sage: H.anemic_subalgebra()
Anemic Hecke algebra acting on Cuspidal subspace of dimension 3 of Modular Forms space of di
```

is_anemic()
Return False, since this the full Hecke algebra.
EXAMPLES:

```
sage: H = CuspForms(3, 12).hecke_algebra()
sage: H.is_anemic()
False
```

is_HeckeAlgebra(*x*)
Return True if x is of type HeckeAlgebra.

EXAMPLES:

```
sage: from sage.modular.hecke.algebra import is_HeckeAlgebra
sage: is_HeckeAlgebra(CuspForms(1, 12).anemic_hecke_algebra())
True
sage: is_HeckeAlgebra(ZZ)
False
```

42.9 Hecke operators

class HeckeAlgebraElement(*parent*)
Base class for elements of Hecke algebras.

apply_sparse(*x*)
Apply this Hecke operator to x, where we avoid computing the matrix of x if possible.

EXAMPLES:

```
sage: M = ModularSymbols(11)
sage: T = M.hecke_operator(23)
sage: T.apply_sparse(M.gen(0))
24*(1,0) - 5*(1,9)
```

charpoly(*var='x'*)
Return the characteristic polynomial of this Hecke operator.

INPUT:

- `var` - string (default: 'x')

OUTPUT: a monic polynomial in the given variable.

EXAMPLES:

```
sage: M = ModularSymbols(Gamma1(6),4)
sage: M.hecke_operator(2).charpoly('x')
x^6 - 14*x^5 + 29*x^4 + 172*x^3 - 124*x^2 - 320*x + 256
```

codomain()
The codomain of this operator. This is the Hecke module associated to the parent Hecke algebra.

EXAMPLE:

```
sage: R = ModularForms(Gamma0(7), 4).hecke_algebra()
sage: sage.modular.hecke.hecke_operator.HeckeAlgebraElement(R).codomain()
Modular Forms space of dimension 3 for Congruence Subgroup Gamma0(7) of weight 4 over Ration
```

decomposition()
Decompose the Hecke module under the action of this Hecke operator.

EXAMPLES:

```
sage: M = ModularSymbols(11)
sage: t2 = M.hecke_operator(2)
sage: t2.decomposition()
[
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0(
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 3 for Gamma_0(
]
```

```
sage: M = ModularSymbols(33, sign=1).new_submodule()
sage: T = M.hecke_operator(2)
sage: T.decomposition()
[
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 6 for Gamma_0(
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 6 for Gamma_0(
]
```

det()

Return the determinant of this Hecke operator.

EXAMPLES:

```
sage: M = ModularSymbols(23)
sage: T = M.hecke_operator(3)
sage: T.det()
100
```

domain()

The domain of this operator. This is the Hecke module associated to the parent Hecke algebra.

EXAMPLE:

```
sage: R = ModularForms(Gamma0(7), 4).hecke_algebra()
sage: sage.modular.hecke.hecke_operator.HeckeAlgebraElement(R).domain()
 Modular Forms space of dimension 3 for Congruence Subgroup Gamma0(7) of weight 4 over Ratio
```

fcp(*var='x'*)

Return the factorization of the characteristic polynomial of this Hecke operator.

EXAMPLES:

```
sage: M = ModularSymbols(23)
sage: T = M.hecke_operator(3)
sage: T.fcp('x')
(x - 4) * (x^2 - 5)^2
```

hecke_module_morphism()

Return the endomorphism of Hecke modules defined by the matrix attached to this Hecke operator.

EXAMPLES:

```
sage: M = ModularSymbols(Gamma1(13))
sage: t = M.hecke_operator(2)
sage: t
Hecke operator T_2 on Modular Symbols space of dimension 15 for Gamma_1(13) of weight 2 with
sage: t.hecke_module_morphism()
Hecke module morphism T_2 defined by the matrix
[ 2  1  0  0  0  0  0  0  0  0  0  0  0  0 -1]
[ 0  2  0  1  0  0  0 -1  0  0  0  0  0  0  0]
[ 0  0  2  0  0  1 -1  1  0 -1  0  1 -1  0  0]
[ 0  0  0  2  1  0  1  0  0  0  1 -1  0  0  0]
[ 0  0  1  0  2  0  0  0  0  1 -1  0  0  0  1]
[ 1  0  0  0  0  2  0  0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  0  1 -1  1 -1  0 -1  1  1]
[ 0  0  0  0  0  0  0 -1  1  1  0  0 -1  1  0]
[ 0  0  0  0  0  0 -1 -1  0  1 -1 -1  1  0 -1]
[ 0  0  0  0  0  0 -2  0  2 -2  0  2 -2  1 -1]
[ 0  0  0  0  0  0  0  0  2 -1  1  0  0  1 -1]
[ 0  0  0  0  0  0 -1  1  2 -1  1  0 -2  2  0]
[ 0  0  0  0  0  0  0  0  1  1  0 -1  0  0  0]
[ 0  0  0  0  0  0 -1  1  1  0  1  1 -1  0  0]
[ 0  0  0  0  0  0  2  0  0  0  2 -1  0  1 -1]
```

```
Domain: Modular Symbols space of dimension 15 for Gamma_1(13) of weight ...
Codomain: Modular Symbols space of dimension 15 for Gamma_1(13) of weight ...
```

image()

Return the image of this Hecke operator.

EXAMPLES:

```
sage: M = ModularSymbols(23)
sage: T = M.hecke_operator(3)
sage: T.fcp('x')
(x - 4) * (x^2 - 5)^2
sage: T.image()
Modular Symbols subspace of dimension 5 of Modular Symbols space of dimension 5 for Gamma_0(
sage: (T-4).image()
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 5 for Gamma_0(
sage: (T**2-5).image()
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 5 for Gamma_0(
```

kernel()

Return the kernel of this Hecke operator.

EXAMPLES:

```
sage: M = ModularSymbols(23)
sage: T = M.hecke_operator(3)
sage: T.fcp('x')
(x - 4) * (x^2 - 5)^2
sage: T.kernel()
Modular Symbols subspace of dimension 0 of Modular Symbols space of dimension 5 for Gamma_0(
sage: (T-4).kernel()
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 5 for Gamma_0(
sage: (T**2-5).kernel()
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 5 for Gamma_0(
```

trace()

Return the trace of this Hecke operator.

```
sage: M = ModularSymbols(1,12)
sage: T = M.hecke_operator(2)
sage: T.trace()
2001
```

class **HeckeAlgebraElement_matrix**(*parent, A*)

An element of the Hecke algebra represented by a matrix.

matrix()

Return the matrix that defines this Hecke algebra element.

EXAMPLES:

```
sage: M = ModularSymbols(1,12)
sage: T = M.hecke_operator(2).matrix_form()
sage: T.matrix()
[ -24    0    0]
[   0  -24    0]
[4860    0 2049]
```

class **HeckeOperator**(*parent, n*)

The Hecke operator T_n for some n (which need not be coprime to the level). The matrix is not computed until it is needed.

index()

Return the index of this Hecke operator, i.e., if this Hecke operator is T_n, return the int n.

EXAMPLES:

```
sage: T = ModularSymbols(11).hecke_operator(17)
sage: T.index()
17
```

matrix()

Return the matrix underlying this Hecke operator.

EXAMPLES:

```
sage: T = ModularSymbols(11).hecke_operator(17)
sage: T.matrix()
[18  0 -4]
[ 0 -2  0]
[ 0  0 -2]
```

matrix_form()

Return the matrix form of this element of a Hecke algebra.

```
sage: T = ModularSymbols(11).hecke_operator(17)
sage: T.matrix_form()
Hecke operator on Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with sign
[18  0 -4]
[ 0 -2  0]
[ 0  0 -2]
```

is_HeckeAlgebraElement(*x*)

Return True if x is of type HeckeAlgebraElement.

EXAMPLES:

```
sage: from sage.modular.hecke.hecke_operator import is_HeckeAlgebraElement
sage: M = ModularSymbols(Gamma0(7), 4)
sage: is_HeckeAlgebraElement(M.T(3))
True
sage: is_HeckeAlgebraElement(M.T(3) + M.T(5))
True
```

is_HeckeOperator(*x*)

Return True if x is of type HeckeOperator.

EXAMPLES:

```
sage: from sage.modular.hecke.hecke_operator import is_HeckeOperator
sage: M = ModularSymbols(Gamma0(7), 4)
sage: is_HeckeOperator(M.T(3))
True
sage: is_HeckeOperator(M.T(3) + M.T(5))
False
```

CHAPTER

FORTYTHREE

MODULAR SYMBOLS

43.1 Creation of modular symbols spaces

EXAMPLES: We create a space and output its category.

```
sage: C = HeckeModules(RationalField()); C
Category of Hecke modules over Rational Field
sage: M = ModularSymbols(11)
sage: M.category()
Category of Hecke modules over Rational Field
sage: M in C
True
```

We create a space compute the charpoly, then compute the same but over a bigger field. In each case we also decompose the space using T_2.

```
sage: M = ModularSymbols(23,2,base_ring=QQ)
sage: print M.T(2).charpoly('x').factor()
(x - 3) * (x^2 + x - 1)^2
sage: print M.decomposition(2)
[
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 5 for Gamma_0(23) of we
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 5 for Gamma_0(23) of we
]


sage: M = ModularSymbols(23,2,base_ring=QuadraticField(5, 'sqrt5'))
sage: print M.T(2).charpoly('x').factor()
(x - 3) * (x - 1/2*sqrt5 + 1/2)^2 * (x + 1/2*sqrt5 + 1/2)^2
sage: print M.decomposition(2)
[
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 5 for Gamma_0(23) of we
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 5 for Gamma_0(23) of we
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 5 for Gamma_0(23) of we
]
```

We compute some Hecke operators and do a consistency check:

```
sage: m = ModularSymbols(39, 2)
sage: t2 = m.T(2); t5 = m.T(5)
sage: t2*t5 - t5*t2 == 0
True
```

This tests the bug reported in trac #1220:

```
sage: G = GammaH(36, [13, 19])
sage: G.modular_symbols()
Modular Symbols space of dimension 13 for Congruence Subgroup Gamma_H(36) with H generated by [13, 19
sage: G.modular_symbols().cuspidal_subspace()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 13 for Congruence Subgr
```

This test catches a tricky corner case for spaces with character:

```
sage: ModularSymbols(DirichletGroup(20).1**3, weight=3, sign=1).cuspidal_subspace()
Modular Symbols subspace of dimension 3 of Modular Symbols space of dimension 6 and level 20, weight
```

ModularSymbols(*group=1, weight=2, sign=0, base_ring=Rational Field, use_cache=True*)

Create an ambient space of modular symbols.

INPUT:

- `group` - A congruence subgroup or a Dirichlet character eps.

weight - int, the weight, which must be = 2.

sign - int, The sign of the involution on modular symbols induced by complex conjugation. The default is 0, which means"no sign", i.e., take the whole space.

base_ring - the base ring. This is ignored if group is a Dirichlet character.

EXAMPLES: First we create some spaces with trivial character:

```
sage: ModularSymbols(Gamma0(11),2).dimension()
3
sage: ModularSymbols(Gamma0(1),12).dimension()
3
```

If we give an integer N for the congruence subgroup, it defaults to $\Gamma_0(N)$:

```
sage: ModularSymbols(1,12,-1).dimension()
1
sage: ModularSymbols(11,4, sign=1)
Modular Symbols space of dimension 4 for Gamma_0(11) of weight 4 with sign 1 over Rational Field
```

We create some spaces for $\Gamma_1(N)$.

```
sage: ModularSymbols(Gamma1(13),2)
Modular Symbols space of dimension 15 for Gamma_1(13) of weight 2 with sign 0 and over Rational
sage: ModularSymbols(Gamma1(13),2, sign=1).dimension()
13
sage: ModularSymbols(Gamma1(13),2, sign=-1).dimension()
2
sage: [ModularSymbols(Gamma1(7),k).dimension() for k in [2,3,4,5]]
[5, 8, 12, 16]
sage: ModularSymbols(Gamma1(5),11).dimension()
20
```

We create a space for $\Gamma_H(N)$:

```
sage: G = GammaH(15,[4,13])
sage: M = ModularSymbols(G,2)
sage: M.decomposition()
[
```

```
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 5 for Congruence S
Modular Symbols subspace of dimension 3 of Modular Symbols space of dimension 5 for Congruence S
]
```

We create a space with character:

```
sage: e = (DirichletGroup(13).0)^2
sage: e.order()
6
sage: M = ModularSymbols(e, 2); M
Modular Symbols space of dimension 4 and level 13, weight 2, character [zeta6], sign 0, over Cyc
sage: f = M.T(2).charpoly('x'); f
x^4 + (-zeta6 - 1)*x^3 - 8*zeta6*x^2 + (10*zeta6 - 5)*x + 21*zeta6 - 21
sage: f.factor()
(x - 2*zeta6 - 1) * (x - zeta6 - 2) * (x + zeta6 + 1)^2
```

More examples of spaces with character:

```
sage: e = DirichletGroup(5, RationalField()).gen(); e
[-1]
sage: m = ModularSymbols(e, 2); m
Modular Symbols space of dimension 2 and level 5, weight 2, character [-1], sign 0, over Rationa


sage: m.T(2).charpoly('x')
x^2 - 1
sage: m = ModularSymbols(e, 6); m.dimension()
6
sage: m.T(2).charpoly('x')
x^6 - 873*x^4 - 82632*x^2 - 1860496
```

We create a space of modular symbols with nontrivial character in characteristic 2.

```
sage: G = DirichletGroup(13,GF(4,'a')); G
Group of Dirichlet characters of modulus 13 over Finite Field in a of size 2^2
sage: e = G.list()[2]; e
[a + 1]
sage: M = ModularSymbols(e,4); M
Modular Symbols space of dimension 8 and level 13, weight 4, character [a + 1], sign 0, over Fin
sage: M.basis()
([X*Y,(1,0)], [X*Y,(1,5)], [X*Y,(1,10)], [X*Y,(1,11)], [X^2,(0,1)], [X^2,(1,10)], [X^2,(1,11)],
sage: M.T(2).matrix()
[    0     0     0     0     0     0     1     1]
[    0     0     0     0     0     0     0     0]
[    0     0     0     0     0 a + 1     1     a]
[    0     0     0     0     0     1 a + 1     a]
[    0     0     0     0 a + 1     0     1     1]
[    0     0     0     0     0     a     1     a]
[    0     0     0     0     0     0 a + 1     a]
[    0     0     0     0     0     0     1     0]
```

TESTS: We test use_cache:

```
sage: ModularSymbols_clear_cache()
sage: M = ModularSymbols(11,use_cache=False)
sage: sage.modular.modsym.modsym._cache
{}
sage: M = ModularSymbols(11,use_cache=True)
sage: sage.modular.modsym.modsym._cache
```

```
{(Congruence Subgroup Gamma0(11), 2, 0, Rational Field): <weakref at ...; to 'ModularSymbolsAmbi
sage: M is ModularSymbols(11,use_cache=True)
True
sage: M is ModularSymbols(11,use_cache=False)
False
```

ModularSymbols_clear_cache()

Clear the global cache of modular symbols spaces.

EXAMPLES:

```
sage: sage.modular.modsym.modsym.ModularSymbols_clear_cache()
sage: sage.modular.modsym.modsym._cache.keys()
[]
sage: M = ModularSymbols(6,2)
sage: sage.modular.modsym.modsym._cache.keys()
[(Congruence Subgroup Gamma0(6), 2, 0, Rational Field)]
sage: sage.modular.modsym.modsym.ModularSymbols_clear_cache()
sage: sage.modular.modsym.modsym._cache.keys()
[]
```

canonical_parameters(*group, weight, sign, base_ring*)

Return the canonically normalized parameters associated to a choice of group, weight, sign, and base_ring. That is, normalize each of these to be of the correct type, perform all appropriate type checking, etc.

EXAMPLES:

```
sage: p1 = sage.modular.modsym.modsym.canonical_parameters(5,int(2),1,QQ) ; p1
(Congruence Subgroup Gamma0(5), 2, 1, Rational Field)
sage: p2 = sage.modular.modsym.modsym.canonical_parameters(Gamma0(5),2,1,QQ) ; p2
(Congruence Subgroup Gamma0(5), 2, 1, Rational Field)
sage: p1 == p2
True
sage: type(p1[1])
<type 'sage.rings.integer.Integer'>
```

43.2 Space of modular symbols (base class)

All the spaces of modular symbols derive from this class. This class is an abstract base class.

class IntegralPeriodMapping(*modsym, A*)

class ModularSymbolsSpace(*group, weight, character, sign, base_ring*)

Base class for spaces of modular symbols.

abelian_variety()

Return the corresponding abelian variety.

INPUT:

- `self` - modular symbols space of weight 2 for a congruence subgroup such as Gamma0, Gamma1 or GammaH.

EXAMPLES:

```
sage: ModularSymbols(Gamma0(11)).cuspidal_submodule().abelian_variety()
Abelian variety J0(11) of dimension 1
sage: ModularSymbols(Gamma1(11)).cuspidal_submodule().abelian_variety()
Abelian variety J1(11) of dimension 1
```

```
sage: ModularSymbols(GammaH(11,[3])).cuspidal_submodule().abelian_variety()
Abelian variety JH(11,[3]) of dimension 1
```

The abelian variety command only works on cuspidal modular symbols spaces:

```
sage: M = ModularSymbols(37)
sage: M[0].abelian_variety()
...
ValueError: self must be cuspidal
sage: M[1].abelian_variety()
Abelian subvariety of dimension 1 of J0(37)
sage: M[2].abelian_variety()
Abelian subvariety of dimension 1 of J0(37)
```

character()

Return the character associated to self.

EXAMPLES:

```
sage: ModularSymbols(12,8).character()
[1, 1]
sage: ModularSymbols(DirichletGroup(25).0, 4).character()
[zeta20]
```

compact_system_of_eigenvalues(*v, names='alpha', nz=None*)

Return a compact system of eigenvalues a_n for $n \in v$. This should only be called on simple factors of modular symbols spaces.

INPUT:

- v - a list of positive integers
- nz - (default: None); if given specifies a column index such that the dual module has that column nonzero.

OUTPUT:

- E - matrix such that E*v is a vector with components the eigenvalues a_n for $n \in v$.
- v - a vector over a number field

EXAMPLES:

```
sage: M = ModularSymbols(43,2,1)[2]; M
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 4 for Gamma_0(
sage: E, v = M.compact_system_of_eigenvalues(prime_range(10))
sage: E
[ 3 -2]
[-3  2]
[-1  2]
[ 1 -2]
sage: v
(1, -1/2*alpha + 3/2)
sage: E*v
(alpha, -alpha, -alpha + 2, alpha - 2)
```

congruence_number(*other, prec=None*)

Given two cuspidal spaces of modular symbols, compute the congruence number, using prec terms of the q-expansions.

The congruence number is defined as follows. If V is the submodule of integral cusp forms corresponding to self (satured in $\mathbf{Z}[[q]]$, by definition) and W is the submodule corresponding to other, each computed to precision prec, the congruence number is the index of $V + W$ in its saturation in $\mathbf{Z}[[q]]$.

If prec is not given it is set equal to the max of the hecke_bound function called on each space.

EXAMPLES:

sage: A, B = ModularSymbols(48, 2).cuspidal_submodule().decomposition() sage: A.congruence_number(B) 2

cuspidal_submodule()

Return the cuspidal submodule of self.

Note: This should be overridden by all derived classes.

EXAMPLES:

```
sage: sage.modular.modsym.space.ModularSymbolsSpace(Gamma0(11),2,DirichletGroup(11).gens()[0
...
NotImplementedError: computation of cuspidal submodule not yet implemented for this class
sage: ModularSymbols(Gamma0(11),2).cuspidal_submodule()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 3 for Gamma_0(
```

cuspidal_subspace()

Synonym for cuspidal_submodule.

EXAMPLES:

```
sage: m = ModularSymbols(Gamma1(3),12); m.dimension()
8
sage: m.cuspidal_subspace().new_subspace().dimension()
2
```

default_prec()

Get the default precision for computation of q-expansion associated to the ambient space of this space of modular symbols (and all subspaces). Use `set_default_prec` to change the default precision.

EXAMPLES:

```
sage: M = ModularSymbols(15)
sage: M.cuspidal_submodule().q_expansion_basis()
[
q - q^2 - q^3 - q^4 + q^5 + q^6 + O(q^8)
]
sage: M.set_default_prec(20)
```

Notice that setting the default precision of the ambient space affects the subspaces.

```
sage: M.cuspidal_submodule().q_expansion_basis()
[
q - q^2 - q^3 - q^4 + q^5 + q^6 + 3*q^8 + q^9 - q^10 - 4*q^11 + q^12 - 2*q^13 - q^15 - q^16
]
sage: M.cuspidal_submodule().default_prec()
20
```

dimension_of_associated_cuspform_space()

Return the dimension of the corresponding space of cusp forms.

The input space must be cuspidal, otherwise there is no corresponding space of cusp forms.

EXAMPLES:

```
sage: m = ModularSymbols(Gamma0(389),2).cuspidal_subspace(); m.dimension()
64
sage: m.dimension_of_associated_cuspform_space()
32
sage: m = ModularSymbols(Gamma0(389),2,sign=1).cuspidal_subspace(); m.dimension()
32
sage: m.dimension_of_associated_cuspform_space()
32
```

dual_star_involution_matrix()

Return the matrix of the dual star involution, which is induced by complex conjugation on the linear dual of modular symbols.

Note: This should be overridden in all derived classes.

EXAMPLES:

```
sage: sage.modular.modsym.space.ModularSymbolsSpace(Gamma0(11),2,DirichletGroup(11).gens()[0
...
NotImplementedError: computation of dual star involution matrix not yet implemented for this
sage: ModularSymbols(Gamma0(11),2).dual_star_involution_matrix()
[ 1  0  0]
[ 0 -1  0]
[ 0  1  1]
```

eisenstein_subspace()

Synonym for eisenstein_submodule.

EXAMPLES:

```
sage: m = ModularSymbols(Gamma1(3),12); m.dimension()
8
sage: m.eisenstein_subspace().dimension()
2
sage: m.cuspidal_subspace().dimension()
6
```

group()

Returns the group of this modular symbols space.

INPUT:

- `ModularSymbols self` - an arbitrary space of modular symbols

OUTPUT:

- `CongruenceSubgroup` - the congruence subgroup that this is a space of modular symbols for.

ALGORITHM: The group is recorded when this space is created.

EXAMPLES:

```
sage: m = ModularSymbols(20)
sage: m.group()
Congruence Subgroup Gamma0(20)
```

hecke_module_of_level(*level*)

Alias for `self.modular_symbols_of_level(level)`.

EXAMPLE:

```
sage: ModularSymbols(11, 2).hecke_module_of_level(22)
Modular Symbols space of dimension 7 for Gamma_0(22) of weight 2 with sign 0 over Rational F
```

integral_basis()

Return a basis for the **Z**-submodule of this modular symbols space spanned by the generators.

Modular symbols spaces for congruence subgroups have a **Z**-structure. Computing this **Z**-structure is expensive, so by default modular symbols spaces for congruence subgroups in Sage are defined over **Q**. This function returns a tuple of independent elements in this modular symbols space whose **Z**-span is the corresponding space of modular symbols over **Z**.

EXAMPLES:

```
sage: M = ModularSymbols(11)
sage: M.basis()
((1,0), (1,8), (1,9))
sage: M.integral_basis()
((1,0), (1,8), (1,9))
sage: S = M.cuspidal_submodule()
sage: S.basis()
```

```
((1,8), (1,9))
sage: S.integral_basis()
((1,8), (1,9))

sage: M = ModularSymbols(13,4)
sage: M.basis()
([X^2,(0,1)], [X^2,(1,4)], [X^2,(1,5)], [X^2,(1,7)], [X^2,(1,9)], [X^2,(1,10)], [X^2,(1,11)]
sage: M.integral_basis()
([X^2,(0,1)], 1/28*[X^2,(1,4)] + 2/7*[X^2,(1,5)] + 3/28*[X^2,(1,7)] + 11/14*[X^2,(1,9)] + 2/
sage: S = M.cuspidal_submodule()
sage: S.basis()
([X^2,(1,4)] - [X^2,(1,12)], [X^2,(1,5)] - [X^2,(1,12)], [X^2,(1,7)] - [X^2,(1,12)], [X^2,(1
sage: S.integral_basis()
(1/28*[X^2,(1,4)] + 2/7*[X^2,(1,5)] + 3/28*[X^2,(1,7)] + 11/14*[X^2,(1,9)] + 2/7*[X^2,(1,10)
```

This function currently raises a NotImplementedError on modular symbols spaces with character of order bigger than 2:

EXAMPLES:

```
sage: M = ModularSymbols(DirichletGroup(13).0^2, 2); M
Modular Symbols space of dimension 4 and level 13, weight 2, character [zeta6], sign 0, over
sage: M.basis()
((1,0), (1,5), (1,10), (1,11))
sage: M.integral_basis()
...
NotImplementedError
```

integral_hecke_matrix(*n*)

Return the matrix of the $n`thHeckeoperatoractingontheintegralstructureonself(asreturnedby"self.integral_structu$ This is often (but not always) different from the matrix returned by `self.hecke_matrix`, even if the latter has integral entries.

EXAMPLE:

```
sage: M = ModularSymbols(6,4)
sage: M.hecke_matrix(3)
[27  0  0  0  6 -6]
[ 0  1 -4  4  8 10]
[18  0  1  0  6 -6]
[18  0  4 -3  6 -6]
[ 0  0  0  0  9 18]
[ 0  0  0  0 12 15]
sage: M.integral_hecke_matrix(3)
[ 27   0   0   0   6  -6]
[  0   1  -8   8  12  14]
[ 18   0   5  -4  14   8]
[ 18   0   8  -7   2 -10]
[  0   0   0   0   9  18]
[  0   0   0   0  12  15]
```

integral_period_mapping()

Return the integral period mapping associated to self.

This is a homomorphism to a vector space whose kernel is the same as the kernel of the period mapping associated to self, normalized so the image of integral modular symbols is exactly $\mathbf{Z}^n$.

EXAMPLES:

```
sage: m = ModularSymbols(23).cuspidal_submodule()
sage: i = m.integral_period_mapping()
sage: i
```

```
Integral period mapping associated to Modular Symbols subspace of dimension 4 of Modular Sym
sage: i.matrix()
[-1/11  1/11     0  3/11]
[    1     0     0     0]
[    0     1     0     0]
[    0     0     1     0]
[    0     0     0     1]
sage: [i(b) for b in m.integral_structure().basis()]
[(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1)]
sage: [i(b) for b in m.ambient_module().basis()]
[(-1/11, 1/11, 0, 3/11),
 (1, 0, 0, 0),
 (0, 1, 0, 0),
 (0, 0, 1, 0),
 (0, 0, 0, 1)]
```

We compute the image of the winding element:

```
sage: m = ModularSymbols(37,sign=1)
sage: a = m[1]
sage: f = a.integral_period_mapping()
sage: e = m([0,oo])
sage: f(e)
(-2/3)
```

The input space must be cuspidal:

```
sage: m = ModularSymbols(37,2,sign=1)
sage: m.integral_period_mapping()
...
ValueError: integral mapping only defined for cuspidal spaces
```

integral_structure()

Return the $\mathbf{Z}$-structure of this modular symbols spaces generated by all integral modular symbols.

EXAMPLES:

```
sage: M = ModularSymbols(11,4)
sage: M.integral_structure()
Free module of degree 6 and rank 6 over Integer Ring
Echelon basis matrix:
[    1     0     0     0     0     0]
[    0  1/14   1/7  5/14   1/2 13/14]
[    0     0   1/2     0     0   1/2]
[    0     0     0     1     0     0]
[    0     0     0     0     1     0]
[    0     0     0     0     0     1]
sage: M.cuspidal_submodule().integral_structure()
Free module of degree 6 and rank 4 over Integer Ring
Echelon basis matrix:
[     0   1/14    1/7   5/14    1/2 -15/14]
[     0      0    1/2      0      0   -1/2]
[     0      0      0      1      0     -1]
[     0      0      0      0      1     -1]
```

intersection_number(M)

Given modular symbols spaces self and M in some common ambient space, returns the intersection number of these two spaces. This is the index in their saturation of the sum of their underlying integral structures.

If self and M are of weight two and defined over QQ, and correspond to newforms f and g, then this number equals the order of the intersection of the modular abelian varieties attached to f and g.

EXAMPLES:

```
sage: m = ModularSymbols(389,2)
sage: d = m.decomposition(2)
sage: eis = d[0]
sage: ell = d[1]
sage: af = d[-1]
sage: af.intersection_number(eis)
97
sage: af.intersection_number(ell)
400
```

is_ambient()

Return True if self is an ambient space of modular symbols.

EXAMPLES:

```
sage: ModularSymbols(21,4).is_ambient()
True
sage: ModularSymbols(21,4).cuspidal_submodule().is_ambient()
False
```

is_cuspidal()

Return True if self is a cuspidal space of modular symbols.

Note: This should be overridden in all derived classes.

EXAMPLES:

```
sage: sage.modular.modsym.space.ModularSymbolsSpace(Gamma0(11),2,DirichletGroup(11).gens()[0
...
NotImplementedError: computation of cuspidal subspace not yet implemented for this class
sage: ModularSymbols(Gamma0(11),2).is_cuspidal()
False
```

is_simple()

Return whether not this modular symbols space is simple as a module over the anemic Hecke algebra adjoin *.

EXAMPLES:

```
sage: m = ModularSymbols(Gamma0(33),2,sign=1)
sage: m.is_simple()
False
sage: o = m.old_subspace()
sage: o.decomposition()
[
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 6 for Gamma_0(
Modular Symbols subspace of dimension 3 of Modular Symbols space of dimension 6 for Gamma_0(
]
sage: C=ModularSymbols(1,14,0,GF(5)).cuspidal_submodule()
sage: C
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 for Gamma_0(
sage: C.is_simple()
True
```

minus_submodule(*compute_dual=True*)

Return the subspace of self on which the star involution acts as -1.

INPUT:

- `compute_dual` - bool (default: True) also compute dual subspace. This are useful for many algorithms.

OUTPUT: subspace of modular symbols

EXAMPLES:

```
sage: ModularSymbols(14,4)
Modular Symbols space of dimension 12 for Gamma_0(14) of weight 4 with sign 0 over Rational
sage: ModularSymbols(14,4).minus_submodule()
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 12 for Gamma_0
```

modular_symbols_of_sign(*sign, bound=None*)

Returns a space of modular symbols with the same defining properties (weight, level, etc.) and Hecke eigenvalues as this space except with given sign.

INPUT:

- `self` - a cuspidal space of modular symbols
- `sign` - an integer, one of -1, 0, or 1
- `bound` - integer (default: None); if specified only use Hecke operators up to the given bound.

EXAMPLES:

```
sage: S = ModularSymbols(Gamma0(11),2,sign=0).cuspidal_subspace()
sage: S
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 3 for Gamma_0(
sage: S.modular_symbols_of_sign(-1)
Modular Symbols space of dimension 1 for Gamma_0(11) of weight 2 with sign -1 over Rational

sage: S = ModularSymbols(43,2,sign=1)[2]; S
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 4 for Gamma_0(
sage: S.modular_symbols_of_sign(-1)
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 3 for Gamma_0(

sage: S.modular_symbols_of_sign(0)
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 7 for Gamma_0(

sage: S = ModularSymbols(389,sign=1)[3]; S
Modular Symbols subspace of dimension 3 of Modular Symbols space of dimension 33 for Gamma_0
sage: S.modular_symbols_of_sign(-1)
Modular Symbols subspace of dimension 3 of Modular Symbols space of dimension 32 for Gamma_0
sage: S.modular_symbols_of_sign(0)
Modular Symbols subspace of dimension 6 of Modular Symbols space of dimension 65 for Gamma_0

sage: S = ModularSymbols(23,sign=1,weight=4)[2]; S
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 7 for Gamma_0(
sage: S.modular_symbols_of_sign(1) is S
True
sage: S.modular_symbols_of_sign(0)
Modular Symbols subspace of dimension 8 of Modular Symbols space of dimension 12 for Gamma_0
sage: S.modular_symbols_of_sign(-1)
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 5 for Gamma_0(
```

multiplicity(*S, check_simple=True*)

Return the multiplicity of the simple modular symbols space S in self. S must be a simple anemic Hecke module.

ASSUMPTION: self is an anemic Hecke module with the same weight and group as S, and S is simple.

EXAMPLES:

```
sage: M = ModularSymbols(11,2,sign=1)
sage: N1, N2 = M.decomposition()
sage: N1.multiplicity(N2)
0
sage: M.multiplicity(N1)
1
```

```
sage: M.multiplicity(ModularSymbols(14,2))
0
```

new_subspace(*p=None*)

Synonym for new_submodule.

EXAMPLES:

```
sage: m = ModularSymbols(Gamma0(5),12); m.dimension()
12
sage: m.new_subspace().dimension()
6
sage: m = ModularSymbols(Gamma1(3),12); m.dimension()
8
sage: m.new_subspace().dimension()
2
```

ngens()

The number of generators of self.

INPUT:

- `ModularSymbols self` - arbitrary space of modular symbols.

OUTPUT:

- `int` - the number of generators, which is the same as the dimension of self.

ALGORITHM: Call the dimension function.

EXAMPLES:

```
sage: m = ModularSymbols(33)
sage: m.ngens()
9
sage: m.rank()
9
sage: ModularSymbols(100, weight=2, sign=1).ngens()
18
```

old_subspace(*p=None*)

Synonym for old_submodule.

EXAMPLES:

```
sage: m = ModularSymbols(Gamma1(3),12); m.dimension()
8
sage: m.old_subspace().dimension()
6
```

plus_submodule(*compute_dual=True*)

Return the subspace of self on which the star involution acts as +1.

INPUT:

- `compute_dual` - bool (default: True) also compute dual subspace. This are useful for many algorithms.

OUTPUT: subspace of modular symbols

EXAMPLES:

```
sage: ModularSymbols(17,2)
Modular Symbols space of dimension 3 for Gamma_0(17) of weight 2 with sign 0 over Rational F
sage: ModularSymbols(17,2).plus_submodule()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 3 for Gamma_0(
```

q_eigenform(*prec, names=None*)

Returns the q-expansion to precision prec of a new eigenform associated to self, where self must be new, cuspidal, and simple.

EXAMPLES:

```
sage: ModularSymbols(2, 8)[1].q_eigenform(5, 'a')
q - 8*q^2 + 12*q^3 + 64*q^4 + O(q^5)
sage: ModularSymbols(2, 8)[0].q_eigenform(5,'a')
...
ArithmeticError: self must be cuspidal.
```

q_expansion_basis(*prec=None, algorithm='default'*)

Returns a basis of q-expansions (as power series) to precision prec of the space of modular forms associated to self. The q-expansions are defined over the same base ring as self, and a put in echelon form.

INPUT:

- `self` - a space of CUSPIDAL modular symbols
- `prec` - an integer
- `algorithm` - string:
- `'default' (default)` - decide which algorithm to use based on heuristics
- `'hecke'` - compute basis by computing homomorphisms T - K, where T is the Hecke algebra
- `'eigen'` - compute basis using eigenvectors for the Hecke action and Atkin-Lehner-Li theory to patch them together
- `'all'` - compute using hecke_dual and eigen algorithms and verify that the results are the same.

The computed basis is *not* cached, though of course Hecke operators used in computing the basis are cached.

EXAMPLES:

```
sage: M = ModularSymbols(1, 12).cuspidal_submodule()
sage: M.q_expansion_basis(8)
[
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7 + O(q^8)
]

sage: M.q_expansion_basis(8, algorithm='eigen')
[
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7 + O(q^8)
]

sage: M = ModularSymbols(1, 24).cuspidal_submodule()
sage: M.q_expansion_basis(8, algorithm='eigen')
[
q + 195660*q^3 + 12080128*q^4 + 44656110*q^5 - 982499328*q^6 - 147247240*q^7 + O(q^8),
q^2 - 48*q^3 + 1080*q^4 - 15040*q^5 + 143820*q^6 - 985824*q^7 + O(q^8)
]

sage: M = ModularSymbols(11, 2, sign=-1).cuspidal_submodule()
sage: M.q_expansion_basis(8, algorithm='eigen')
[
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)
]

sage: M = ModularSymbols(Gamma1(13), 2, sign=1).cuspidal_submodule()
sage: M.q_expansion_basis(8, algorithm='eigen')
[
q - 4*q^3 - q^4 + 3*q^5 + 6*q^6 + O(q^8),
```

```
q^2 - 2*q^3 - q^4 + 2*q^5 + 2*q^6 + O(q^8)
]

sage: M = ModularSymbols(Gamma1(5), 3, sign=-1).cuspidal_submodule()
sage: M.q_expansion_basis(8, algorithm='eigen')    # dimension is 0
[]

sage: M = ModularSymbols(Gamma1(7), 3, sign=-1).cuspidal_submodule()
sage: M.q_expansion_basis(8)
[
q - 3*q^2 + 5*q^4 - 7*q^7 + O(q^8)
]

sage: M = ModularSymbols(43, 2, sign=0).cuspidal_submodule()
sage: M[0]
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 7 for Gamma_0(
sage: M[0].q_expansion_basis()
[
q - 2*q^2 - 2*q^3 + 2*q^4 - 4*q^5 + 4*q^6 + O(q^8)
]
sage: M[1]
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 7 for Gamma_0(
sage: M[1].q_expansion_basis()
[
q + 2*q^5 - 2*q^6 - 2*q^7 + O(q^8),
q^2 - q^3 - q^5 + q^7 + O(q^8)
]
```

q_expansion_cuspforms (*prec=None*)

Returns a function f(i,j) such that each value f(i,j) is the q-expansion, to the given precision, of an element of the corresponding space S of cusp forms. Together these functions span S. Here i, j are integers with $0 \leq i, j < d$, where d is the dimension of self.

For a reduced echelon basis, use the function `q_expansion_basis` instead.

More precisely, this function returns the q-expansions obtained by taking the ij entry of the matrices of the Hecke operators T_n acting on the subspace of the linear dual of modular symbols corresponding to self.

EXAMPLES:

```
sage: S = ModularSymbols(11,2, sign=1).cuspidal_submodule()
sage: f = S.q_expansion_cuspforms(8)
sage: f(0,0)
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 + O(q^8)

sage: S = ModularSymbols(37,2).cuspidal_submodule()
sage: f = S.q_expansion_cuspforms(8)
sage: f(0,0)
q + q^3 - 2*q^4 - q^7 + O(q^8)
sage: f(3,3)
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + 6*q^6 - q^7 + O(q^8)
sage: f(1,2)
q^2 + 2*q^3 - 2*q^4 + q^5 - 3*q^6 + O(q^8)

sage: S = ModularSymbols(Gamma1(13),2,sign=-1).cuspidal_submodule()
sage: f = S.q_expansion_cuspforms(8)
sage: f(0,0)
q - 2*q^2 + q^4 - q^5 + 2*q^6 + O(q^8)
sage: f(0,1)
q^2 - 2*q^3 - q^4 + 2*q^5 + 2*q^6 + O(q^8)
```

```
sage: S = ModularSymbols(1,12,sign=-1).cuspidal_submodule()
sage: f = S.q_expansion_cuspforms(8)
sage: f(0,0)
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7 + O(q^8)
```

q_expansion_module (*prec=None, R=None*)

Return a basis over R for the space spanned by the coefficient vectors of the q-expansions corresponding to self. If R is not the base ring of self, returns the restriction of scalars down to R (for this, self must have base ring $\mathbf{Q}$ or a number field).

INPUT:

- `self` - must be cuspidal
- `prec` - an integer (default: self.default_prec())
- `R` - either ZZ, QQ, or the base_ring of self (which is the default)

OUTPUT: A free module over R.

TODO - extend to more general R (though that is fairly easy for the user to get by just doing base_extend or change_ring on the output of this function).

Note that the prec needed to distinguish elements of the restricted-down-to-R basis may be bigger than `self.hecke_bound()`, since one must use the Sturm bound for modular forms on $\Gamma_H(N)$.

EXAMPLES WITH SIGN 1 and R=QQ:

Basic example with sign 1:

```
sage: M = ModularSymbols(11, sign=1).cuspidal_submodule()
sage: M.q_expansion_module(5, QQ)
Vector space of degree 5 and dimension 1 over Rational Field
Basis matrix:
[ 0  1 -2 -1  2]
```

Same example with sign -1:

```
sage: M = ModularSymbols(11, sign=-1).cuspidal_submodule()
sage: M.q_expansion_module(5, QQ)
Vector space of degree 5 and dimension 1 over Rational Field
Basis matrix:
[ 0  1 -2 -1  2]
```

An example involving old forms:

```
sage: M = ModularSymbols(22, sign=1).cuspidal_submodule()
sage: M.q_expansion_module(5, QQ)
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[ 0  1  0 -1 -2]
[ 0  0  1  0 -2]
```

An example that (somewhat spuriously) is over a number field:

```
sage: x = polygen(QQ)
sage: k = NumberField(x^2+1, 'a')
sage: M = ModularSymbols(11, base_ring=k, sign=1).cuspidal_submodule()
sage: M.q_expansion_module(5, QQ)
Vector space of degree 5 and dimension 1 over Rational Field
Basis matrix:
[ 0  1 -2 -1  2]
```

An example that involves an eigenform with coefficients in a number field:

```
sage: M = ModularSymbols(23, sign=1).cuspidal_submodule()
sage: M.q_eigenform(4, 'gamma')
```

```
q + gamma*q^2 + (-2*gamma - 1)*q^3 + O(q^4)
sage: M.q_expansion_module(11, QQ)
Vector space of degree 11 and dimension 2 over Rational Field
Basis matrix:
[ 0  1  0 -1 -1  0 -2  2 -1  2  2]
[ 0  0  1 -2 -1  2  1  2 -2  0 -2]
```

An example that is genuinely over a base field besides QQ.

```
sage: eps = DirichletGroup(11).0
sage: M = ModularSymbols(eps,3,sign=1).cuspidal_submodule(); M
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 and level 11
sage: M.q_eigenform(4, 'beta')
q + (-zeta10^3 + 2*zeta10^2 - 2*zeta10)*q^2 + (2*zeta10^3 - 3*zeta10^2 + 3*zeta10 - 2)*q^3 +
sage: M.q_expansion_module(7, QQ)
Vector space of degree 7 and dimension 4 over Rational Field
Basis matrix:
[  0   1   0   0   0 -40  64]
[  0   0   1   0   0 -24  41]
[  0   0   0   1   0 -12  21]
[  0   0   0   0   1  -4   4]
```

An example involving an eigenform rational over the base, but the base is not QQ.

```
sage: k.<a> = NumberField(x^2-5)
sage: M = ModularSymbols(23, base_ring=k, sign=1).cuspidal_submodule()
sage: D = M.decomposition(); D
[
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0(
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0(
]
sage: M.q_expansion_module(8, QQ)
Vector space of degree 8 and dimension 2 over Rational Field
Basis matrix:
[ 0  1  0 -1 -1  0 -2  2]
[ 0  0  1 -2 -1  2  1  2]
```

An example involving an eigenform not rational over the base and for which the base is not QQ.

```
sage: eps = DirichletGroup(25).0^2
sage: M = ModularSymbols(eps,2,sign=1).cuspidal_submodule(); M
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 4 and level 25
sage: D = M.decomposition(); D
[
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 4 and level 25
]
sage: D[0].q_eigenform(4, 'mu')
q + mu*q^2 + ((zeta10^3 + zeta10 - 1)*mu + zeta10^2 - 1)*q^3 + O(q^4)
sage: D[0].q_expansion_module(11, QQ)
Vector space of degree 11 and dimension 8 over Rational Field
Basis matrix:
[  0   1   0   0   0   0   0   0 -20  -3   0]
[  0   0   1   0   0   0   0   0 -16  -1   0]
[  0   0   0   1   0   0   0   0 -11  -2   0]
[  0   0   0   0   1   0   0   0  -8  -1   0]
[  0   0   0   0   0   1   0   0  -5  -1   0]
[  0   0   0   0   0   0   1   0  -3  -1   0]
[  0   0   0   0   0   0   0   1  -2   0   0]
[  0   0   0   0   0   0   0   0   0   0   1]
sage: D[0].q_expansion_module(11)
```

```
Vector space of degree 11 and dimension 2 over Cyclotomic Field of order 10 and degree 4
Basis matrix:
[                                  0                                   1
[                                  0                                   0
```

EXAMPLES WITH SIGN 0 and R=QQ:

TODO - this doesn't work yet as it's not implemented!!

```
sage: M = ModularSymbols(11,2).cuspidal_submodule() #not tested
sage: M.q_expansion_module() #not tested
... boom ...
```

EXAMPLES WITH SIGN 1 and R=ZZ (computes saturation):

```
sage: M = ModularSymbols(43,2, sign=1).cuspidal_submodule()
sage: M.q_expansion_module(8, QQ)
Vector space of degree 8 and dimension 3 over Rational Field
Basis matrix:
[   0    1    0    0    0    2   -2   -2]
[   0    0    1    0 -1/2    1 -3/2    0]
[   0    0    0    1 -1/2    2 -3/2   -1]
sage: M.q_expansion_module(8, ZZ)
Free module of degree 8 and rank 3 over Integer Ring
Echelon basis matrix:
[ 0  1  0  0  0  2 -2 -2]
[ 0  0  1  1 -1  3 -3 -1]
[ 0  0  0  2 -1  4 -3 -2]
```

rational_period_mapping()

Return the rational period mapping associated to self.

This is a homomorphism to a vector space whose kernel is the same as the kernel of the period mapping associated to self. For this to exist, self must be Hecke equivariant.

Use `integral_period_mapping` to obtain a homomorphism to a $\mathbf{Z}$-module, normalized so the image of integral modular symbols is exactly $\mathbf{Z}^n$.

EXAMPLES:

```
sage: M = ModularSymbols(37)
sage: A = M[1]; A
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 5 for Gamma_0(
sage: r = A.rational_period_mapping(); r
Rational period mapping associated to Modular Symbols subspace of dimension 2 of Modular Sym
sage: r(M.0)
(0, 0)
sage: r(M.1)
(1, 0)
sage: r.matrix()
[ 0  0]
[ 1  0]
[ 0  1]
[-1 -1]
[ 0  0]
sage: r.domain()
Modular Symbols space of dimension 5 for Gamma_0(37) of weight 2 with sign 0 over Rational F
sage: r.codomain()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
```

set_default_prec(*prec*)

Set the default precision for computation of q-expansion associated to the ambient space of this space of modular symbols (and all subspaces).

EXAMPLES:

```
sage: M = ModularSymbols(Gamma1(13),2)
sage: M.set_default_prec(5)
sage: M.cuspidal_submodule().q_expansion_basis()
[
q - 4*q^3 - q^4 + O(q^5),
q^2 - 2*q^3 - q^4 + O(q^5)
]
```

set_precision(*prec*)

Same as self.set_default_prec(prec).

EXAMPLES:

```
sage: M = ModularSymbols(17,2)
sage: M.cuspidal_submodule().q_expansion_basis()
[
q - q^2 - q^4 - 2*q^5 + 4*q^7 + O(q^8)
]
sage: M.set_precision(10)
sage: M.cuspidal_submodule().q_expansion_basis()
[
q - q^2 - q^4 - 2*q^5 + 4*q^7 + 3*q^8 - 3*q^9 + O(q^10)
]
```

sign()

Returns the sign of self.

For efficiency reasons, it is often useful to compute in the (largest) quotient of modular symbols where the * involution acts as +1, or where it acts as -1.

INPUT:

- `ModularSymbols self` - arbitrary space of modular symbols.

OUTPUT:

- `int` - the sign of self, either -1, 0, or 1.
- `-1` - if this is factor of quotient where * acts as -1,
- `+1` - if this is factor of quotient where * acts as +1,
- `0` - if this is full space of modular symbols (no quotient).

EXAMPLES:

```
sage: m = ModularSymbols(33)
sage: m.rank()
9
sage: m.sign()
0
sage: m = ModularSymbols(33, sign=0)
sage: m.sign()
0
sage: m.rank()
9
sage: m = ModularSymbols(33, sign=-1)
sage: m.sign()
-1
sage: m.rank()
3
```

sign_submodule(*sign, compute_dual=True*)

Return the subspace of self that is fixed under the star involution.

INPUT:

- `sign` - int (either -1, 0 or +1)
- `compute_dual` - bool (default: True) also compute dual subspace. This are useful for many algorithms.

OUTPUT: subspace of modular symbols

EXAMPLES:

```
sage: M = ModularSymbols(29,2)
sage: M.sign_submodule(1)
Modular Symbols subspace of dimension 3 of Modular Symbols space of dimension 5 for Gamma_0(
sage: M.sign_submodule(-1)
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 5 for Gamma_0(
sage: M.sign_submodule(-1).sign()
-1
```

simple_factors()

Returns a list modular symbols spaces S where S is simple spaces of modular symbols (for the anemic Hecke algebra) and self is isomorphic to the direct sum of the S with some multiplicities, as a module over the *anemic* Hecke algebra. For the multiplicities use factorization() instead.

ASSUMPTION: self is a module over the anemic Hecke algebra.

EXAMPLES:

```
sage: ModularSymbols(1,100,sign=-1).simple_factors()
[Modular Symbols subspace of dimension 8 of Modular Symbols space of dimension 8 for Gamma_0
sage: ModularSymbols(1,16,0,GF(5)).simple_factors()
[Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0(
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0(
```

star_decomposition()

Decompose self into subspaces which are eigenspaces for the star involution.

EXAMPLE:

```
sage: ModularSymbols(Gamma1(19), 2).cuspidal_submodule().star_decomposition()
[
Modular Symbols subspace of dimension 7 of Modular Symbols space of dimension 31 for Gamma_1
Modular Symbols subspace of dimension 7 of Modular Symbols space of dimension 31 for Gamma_1
]
```

star_eigenvalues()

Returns the eigenvalues of the star involution acting on self.

EXAMPLES:

```
sage: M = ModularSymbols(11)
sage: D = M.decomposition()
sage: M.star_eigenvalues()
[1, -1]
sage: D[0].star_eigenvalues()
[1]
sage: D[1].star_eigenvalues()
[1, -1]
sage: D[1].plus_submodule().star_eigenvalues()
[1]
sage: D[1].minus_submodule().star_eigenvalues()
[-1]
```

star_involution()
Return the star involution on self, which is induced by complex conjugation on modular symbols. Not implemented in this abstract base class.

EXAMPLES:

```
sage: M = ModularSymbols(11, 2); sage.modular.modsym.space.ModularSymbolsSpace.star_involuti
...
NotImplementedError
```

sturm_bound()
Returns the Sturm bound for this space of modular symbols.

Type `sturm_bound?` for more details.

EXAMPLES:

```
sage: ModularSymbols(11,2).sturm_bound()
2
sage: ModularSymbols(389,2).sturm_bound()
65
sage: ModularSymbols(1,12).sturm_bound()
1
sage: ModularSymbols(1,36).sturm_bound()
3
sage: ModularSymbols(DirichletGroup(31).0^2).sturm_bound()
6
sage: ModularSymbols(Gamma1(31)).sturm_bound()
160
```

class **PeriodMapping**(*modsym, A*)
Base class for representing a period mapping attached to a space of modular symbols. To be used via the derived classes RationalPeriodMapping and IntegralPeriodMapping.

codomain()
Return the codomain of this mapping.

EXAMPLE:

Note that this presently returns the wrong answer, as a consequence of various bugs in the free module routines:

```
sage: ModularSymbols(11, 2).cuspidal_submodule().integral_period_mapping().codomain()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
```

domain()
Return the domain of this mapping (which is the ambient space of the corresponding modular symbols space).

EXAMPLE:

```
sage: ModularSymbols(17, 2).cuspidal_submodule().integral_period_mapping().domain()
Modular Symbols space of dimension 3 for Gamma_0(17) of weight 2 with sign 0 over Rational F
```

matrix()
Return the matrix of this period mapping.

EXAMPLE:

```
sage: ModularSymbols(11, 2).cuspidal_submodule().integral_period_mapping().matrix()
[  0 1/5]
[  1   0]
[  0   1]
```

modular_symbols_space()
Return the space of modular symbols to which this period mapping corresponds.
EXAMPLES:

```
sage: ModularSymbols(17, 2).rational_period_mapping().modular_symbols_space()
Modular Symbols space of dimension 3 for Gamma_0(17) of weight 2 with sign 0 over Rational F
```

class RationalPeriodMapping(*modsym, A*)

is_ModularSymbolsSpace(*x*)
Return True if x is a space of modular symbols.
EXAMPLES:

```
sage: M = ModularForms(3, 2)
sage: sage.modular.modsym.space.is_ModularSymbolsSpace(M)
False
sage: sage.modular.modsym.space.is_ModularSymbolsSpace(M.modular_symbols(sign=1))
True
```

43.3 Ambient spaces of modular symbols.

This module defines the following classes. There is an abstract base class `ModularSymbolsAmbient`, derived from `space.ModularSymbolsSpace` and `hecke.AmbientHeckeModule`. As this is an abstract base class, only derived classes should be instantiated. There are five derived classes:

- `ModularSymbolsAmbient_wtk_g0`, for modular symbols of general weight k for $\Gamma_0(N)$;
- `ModularSymbolsAmbient_wt2_g0` (derived from `ModularSymbolsAmbient_wtk_g0`), for modular symbols of weight 2 for $\Gamma_0(N)$;
- `ModularSymbolsAmbient_wtk_g1`, for modular symbols of general weight k for $\Gamma_1(N)$;
- `ModularSymbolsAmbient_wtk_gamma_h`, for modular symbols of general weight k for Γ_H, where H is a subgroup of $\mathbf{Z}/N\mathbf{Z}$;
- `ModularSymbolsAmbient_wtk_eps`, for modular symbols of general weight k and character ϵ.

EXAMPLES:

We compute a space of modular symbols modulo 2. The dimension is different from that of the corresponding space in characteristic 0:

```
sage: M = ModularSymbols(11,4,base_ring=GF(2)); M
Modular Symbols space of dimension 7 for Gamma_0(11) of weight 4
with sign 0 over Finite Field of size 2
sage: M.basis()
([X*Y,(1,0)], [X*Y,(1,8)], [X*Y,(1,9)], [X^2,(0,1)], [X^2,(1,8)], [X^2,(1,9)], [X^2,(1,10)])
sage: M0 =  ModularSymbols(11,4,base_ring=QQ); M0
Modular Symbols space of dimension 6 for Gamma_0(11) of weight 4
with sign 0 over Rational Field
sage: M0.basis()
([X^2,(0,1)], [X^2,(1,6)], [X^2,(1,7)], [X^2,(1,8)], [X^2,(1,9)], [X^2,(1,10)])
```

The characteristic polynomial of the Hecke operator T_2 has an extra factor x.

```
sage: M.T(2).matrix().fcp('x')
(x + 1)^2 * x^5
sage: M0.T(2).matrix().fcp('x')
(x - 9)^2 * (x^2 - 2*x - 2)^2
```

class ModularSymbolsAmbient(*group, weight, sign, base_ring, character=None*)

An ambient space of modular symbols for a congruence subgroup of $SL_2(\mathbf{Z})$.

This class is an abstract base class, so only derived classes should be instantiated.

INPUT:

- `weight` - an integer
- `group` - a congruence subgroup.
- `sign` - an integer, either -1, 0, or 1
- `base_ring` - a commutative ring

boundary_map()

Return the boundary map to the corresponding space of boundary modular symbols.

EXAMPLES:

```
sage: ModularSymbols(20,2).boundary_map()
Hecke module morphism boundary map defined by the matrix
[ 1 -1  0  0  0  0]
[ 0  1 -1  0  0  0]
[ 0  1  0 -1  0  0]
[ 0  0  0 -1  1  0]
[ 0  1  0 -1  0  0]
[ 0  0  1 -1  0  0]
[ 0  1  0  0  0 -1]
Domain: Modular Symbols space of dimension 7 for Gamma_0(20) of weight ...
Codomain: Space of Boundary Modular Symbols for Congruence Subgroup Gamma0(20) ...
sage: type(ModularSymbols(20,2).boundary_map())
<class 'sage.modular.hecke.morphism.HeckeModuleMorphism_matrix'>
```

boundary_space()

Return the subspace of boundary modular symbols of this modular symbols ambient space.

EXAMPLES:

```
sage: ModularSymbols(20,2).boundary_space()
Space of Boundary Modular Symbols for Congruence Subgroup Gamma0(20) of weight 2 and over Ra
sage: ModularSymbols(20,2).dimension()
7
sage: ModularSymbols(20,2).boundary_space().dimension()
6
```

compact_newform_eigenvalues(*v, names='alpha'*)

Return compact systems of eigenvalues for each Galois conjugacy class of cuspidal newforms in this ambient space.

INPUT:

- `v` - list of positive integers

OUTPUT:

- `list` - of pairs (E, x), where E*x is a vector with entries the eigenvalues a_n for $n \in v$.

EXAMPLES:

```
sage: M = ModularSymbols(43,2,1)
sage: X = M.compact_newform_eigenvalues(prime_range(10))
sage: X[0][0] * X[0][1]
(-2, -2, -4, 0)
sage: X[1][0] * X[1][1]
(alpha1, -alpha1, -alpha1 + 2, alpha1 - 2)

sage: M = ModularSymbols(DirichletGroup(24,QQ).1,2,sign=1)
sage: M.compact_newform_eigenvalues(prime_range(10),'a')
[([-1/2 -1/2]
[ 1/2 -1/2]
[  -1    1]
[  -2    0], (1, -2*a0 - 1))]
sage: a = M.compact_newform_eigenvalues([1..10],'a')[0]
sage: a[0]*a[1]
(1, a0, a0 + 1, -2*a0 - 2, -2*a0 - 2, -a0 - 2, -2, 2*a0 + 4, -1, 2*a0 + 4)
sage: M = ModularSymbols(DirichletGroup(13).0^2,2,sign=1)
sage: M.compact_newform_eigenvalues(prime_range(10),'a')
[([  -zeta6 - 1]
[ 2*zeta6 - 2]
[-2*zeta6 + 1]
[           0], (1))]
sage: a = M.compact_newform_eigenvalues([1..10],'a')[0]
sage: a[0]*a[1]
(1, -zeta6 - 1, 2*zeta6 - 2, zeta6, -2*zeta6 + 1, -2*zeta6 + 4, 0, 2*zeta6 - 1, -zeta6, 3*ze
```

compute_presentation()

Compute and cache the presentation of this space.

EXAMPLES:

```
sage: ModularSymbols(11,2).compute_presentation() # no output
```

cuspidal_submodule()

The cuspidal submodule of this modular symbols ambient space.

EXAMPLES:

```
sage: M = ModularSymbols(12,2,0,GF(5)) ; M
Modular Symbols space of dimension 5 for Gamma_0(12) of weight 2 with sign 0 over Finite Fie
sage: M.cuspidal_submodule()
Modular Symbols subspace of dimension 0 of Modular Symbols space of dimension 5 for Gamma_0(
sage: ModularSymbols(1,24,-1).cuspidal_submodule()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 2 for Gamma_0(
```

The cuspidal submodule of the cuspidal submodule is itself:

```
sage: M = ModularSymbols(389)
sage: S = M.cuspidal_submodule()
sage: S.cuspidal_submodule() is S
True
```

cusps()

Return the set of cusps for this modular symbols space.

EXAMPLES:

```
sage: ModularSymbols(20,2).cusps()
[Infinity, 0, -1/4, 1/5, -1/2, 1/10]
```

dual_star_involution_matrix()

Return the matrix of the dual star involution, which is induced by complex conjugation on the linear dual of modular symbols.

EXAMPLES:

```
sage: ModularSymbols(20,2).dual_star_involution_matrix()
[1 0 0 0 0 0 0]
[0 1 0 0 0 0 0]
[0 0 0 0 1 0 0]
[0 0 0 1 0 0 0]
[0 0 1 0 0 0 0]
[0 0 0 0 0 1 0]
[0 0 0 0 0 0 1]
```

eisenstein_submodule()

Return the Eisenstein submodule of this space of modular symbols.

EXAMPLES:

```
sage: ModularSymbols(20,2).eisenstein_submodule()
Modular Symbols subspace of dimension 5 of Modular Symbols space of dimension 7 for Gamma_0(
```

element(x)

Creates and returns an element of self from a modular symbol, if possible.

INPUT:

- x - an object of one of the following types: ModularSymbol, ManinSymbol.

OUTPUT:

ModularSymbol - a modular symbol with parent self.

EXAMPLES:

```
sage: M = ModularSymbols(11,4,1)
sage: M.T(3)
Hecke operator T_3 on Modular Symbols space of dimension 4 for Gamma_0(11) of weight 4 with
sage: M.T(3)(M.0)
28*[X^2,(0,1)] + 2*[X^2,(1,7)] - [X^2,(1,9)] - [X^2,(1,10)]
sage: M.T(3)(M.0).element()
(28, 2, -1, -1)
```

factor()

Returns a list of pairs (S, e) where S is spaces of modular symbols and self is isomorphic to the direct sum of the S^e as a module over the *anemic* Hecke algebra adjoin the star involution. The cuspidal S are all simple, but the Eisenstein factors need not be simple.

EXAMPLES:

```
sage: ModularSymbols(Gamma0(22), 2).factorization()
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0
(Modular Symbols subspace of dimension 3 of Modular Symbols space of dimension 7 for Gamma_0

sage: ModularSymbols(1,6,0,GF(2)).factorization()
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 for Gamma_0
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 for Gamma_0

sage: ModularSymbols(18,2).factorization()
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 7 for Gamma_0
(Modular Symbols subspace of dimension 5 of Modular Symbols space of dimension 7 for Gamma_0

sage: M = ModularSymbols(DirichletGroup(38,CyclotomicField(3)).0^2,  2, +1); M
Modular Symbols space of dimension 7 and level 38, weight 2, character [zeta3], sign 1, over
sage: M.factorization()                    # long time (about 8 seconds)
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 7 and level 3
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 7 and level 3
```

```
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 7 and level 3
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 7 and level 3
```

factorization()

Returns a list of pairs (S, e) where S is spaces of modular symbols and self is isomorphic to the direct sum of the S^e as a module over the *anemic* Hecke algebra adjoin the star involution. The cuspidal S are all simple, but the Eisenstein factors need not be simple.

EXAMPLES:

```
sage: ModularSymbols(Gamma0(22), 2).factorization()
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0
(Modular Symbols subspace of dimension 3 of Modular Symbols space of dimension 7 for Gamma_0

sage: ModularSymbols(1,6,0,GF(2)).factorization()
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 for Gamma_0
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 for Gamma_0

sage: ModularSymbols(18,2).factorization()
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 7 for Gamma_0
(Modular Symbols subspace of dimension 5 of Modular Symbols space of dimension 7 for Gamma_0

sage: M = ModularSymbols(DirichletGroup(38,CyclotomicField(3)).0^2,  2, +1); M
Modular Symbols space of dimension 7 and level 38, weight 2, character [zeta3], sign 1, over
sage: M.factorization()                    # long time (about 8 seconds)
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 7 and level 3
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 7 and level 3
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 7 and level 3
(Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 7 and level 3
```

integral_structure(*algorithm='default'*)

Return the $\mathbf{Z}$-structure of this modular symbols space, generated by all integral modular symbols.

INPUT:

- `algorithm` - string (default: 'default' - choose heuristically)
  - `'pari'` - use pari for the HNF computation
  - `'padic'` - use p-adic algorithm (only good for dense case)

ALGORITHM: It suffices to consider lattice generated by the free generating symbols $X^iY^{k-2-i}.(u, v)$ after quotienting out by the S (and I) relations, since the quotient by these relations is the same over any ring.

EXAMPLES: In weight 2 the rational basis is often integral.

```
sage: M = ModularSymbols(11,2)
sage: M.integral_structure()
Free module of degree 3 and rank 3 over Integer Ring
Echelon basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

This is rarely the case in higher weight:

```
sage: M = ModularSymbols(6,4)
sage: M.integral_structure()
Free module of degree 6 and rank 6 over Integer Ring
Echelon basis matrix:
[  1   0   0   0   0   0]
[  0   1   0   0   0   0]
```

```
[  0   0 1/2 1/2 1/2 1/2]
[  0   0   0   1   0   0]
[  0   0   0   0   1   0]
[  0   0   0   0   0   1]
```

Here is an example involving $\Gamma_1(N)$.

```
sage: M = ModularSymbols(Gamma1(5),6)
sage: M.integral_structure()
Free module of degree 10 and rank 10 over Integer Ring
Echelon basis matrix:
[     1      0      0      0      0      0      0      0      0      0]
[     0      1      0      0      0      0      0      0      0      0]
[     0      0  1/102      0  5/204  1/136  23/24   3/17 43/136 69/136]
[     0      0      0   1/48      0   1/48  23/24    1/6    1/8  17/24]
[     0      0      0      0   1/24      0  23/24    1/3    1/6    1/2]
[     0      0      0      0      0   1/24  23/24    1/3  11/24   5/24]
[     0      0      0      0      0      0      1      0      0      0]
[     0      0      0      0      0      0      0    1/2      0    1/2]
[     0      0      0      0      0      0      0      0    1/2    1/2]
[     0      0      0      0      0      0      0      0      0      1]
```

is_cuspidal()

Returns True if this space is cuspidal, else False.

EXAMPLES:

```
sage: M = ModularSymbols(20,2)
sage: M.is_cuspidal()
False
sage: S = M.cuspidal_subspace()
sage: S.is_cuspidal()
True
sage: S = M.eisenstein_subspace()
sage: S.is_cuspidal()
False
```

is_eisenstein()

Returns True if this space is Eisenstein, else False.

EXAMPLES:

```
sage: M = ModularSymbols(20,2)
sage: M.is_eisenstein()
False
sage: S = M.eisenstein_submodule()
sage: S.is_eisenstein()
True
sage: S = M.cuspidal_subspace()
sage: S.is_eisenstein()
False
```

manin_basis()

Return a list of indices into the list of Manin generators (see `self.manin_generators()`) such that those symbols form a basis for the quotient of the $\mathbf{Q}$-vector space spanned by Manin symbols modulo the relations.

EXAMPLES:

```
sage: M = ModularSymbols(2,2)
sage: M.manin_basis()
[1]
```

```
sage: [M.manin_generators()[i] for i in M.manin_basis()]
[(1,0)]
sage: M = ModularSymbols(6,2)
sage: M.manin_basis()
[1, 10, 11]
sage: [M.manin_generators()[i] for i in M.manin_basis()]
[(1,0), (3,1), (3,2)]
```

manin_generators()

Return list of all Manin symbols for this space. These are the generators in the presentation of this space by Manin symbols.

EXAMPLES:

```
sage: M = ModularSymbols(2,2)
sage: M.manin_generators()
[(0,1), (1,0), (1,1)]

sage: M = ModularSymbols(1,6)
sage: M.manin_generators()
[[Y^4,(0,0)], [X*Y^3,(0,0)], [X^2*Y^2,(0,0)], [X^3*Y,(0,0)], [X^4,(0,0)]]
```

manin_gens_to_basis()

Return the matrix expressing the manin symbol generators in terms of the basis.

EXAMPLES:

```
sage: ModularSymbols(11,2).manin_gens_to_basis()
[-1  0  0]
[ 1  0  0]
[ 0  0  0]
[ 0  0  1]
[ 0 -1  1]
[ 0 -1  0]
[ 0  0 -1]
[ 0  0 -1]
[ 0  1 -1]
[ 0  1  0]
[ 0  0  1]
[ 0  0  0]
```

manin_symbol(*x*, *check=True*)

Construct a Manin Symbol from the given data.

INPUT:

- x (list) – either $[u, v]$ or $[i, u, v]$, where $0 \leq i \leq k-2$ where k is the weight, and u,'v' are integers defining a valid element of $\mathbb{P}^1(N)$, where N is the level.

OUTPUT:

(ManinSymbol) the monomial Manin Symbol associated to $[i; (u, v)]$, with $i = 0$ if not supplied, corresponding to the symbol $[X^i * Y^{k-2-i}, (u, v)]$.

EXAMPLES:

```
sage: M = ModularSymbols(11,4,1)
sage: M.manin_symbol([2,5,6])
[X^2,(1,10)]
```

manin_symbols()

Return the list of Manin symbols for this modular symbols ambient space.

EXAMPLES:

```
sage: ModularSymbols(11,2).manin_symbols()
Manin Symbol List of weight 2 for Gamma0(11)
```

manin_symbols_basis()

A list of Manin symbols that form a basis for the ambient space self. INPUT:

- `ModularSymbols self` - an ambient space of modular symbols

OUTPUT:

- `list` - a list of 2-tuples (if the weight is 2) or 3-tuples, which represent the Manin symbols basis for self.

EXAMPLES:

```
sage: m = ModularSymbols(23)
sage: m.manin_symbols_basis()
[(1,0), (1,17), (1,19), (1,20), (1,21)]
sage: m = ModularSymbols(6, weight=4, sign=-1)
sage: m.manin_symbols_basis()
[[X^2,(2,1)]]
```

modular_symbol(*x, check=True*)

Create a modular symbol in this space.

INPUT:

- x (list) – a list of either 2 or 3 entries:
  - –2 entries: $[\alpha, \beta]$ where α and β are cusps;
  - –3 entries: $[i, \alpha, \beta]$ where $0 \leq i \leq k-2$ and α and β are cusps;
- `check` (bool, default True) – flag that determines whether the input x needs processing: use check=False for efficiency if the input x is a list of length 3 whose first entry is an Integer, and whose second and third entries are Cusps (se examples).

OUTPUT:

(Modular Symbol) The modular symbol $Y^{k-2}\{\alpha, \beta\}$. or $X^i Y^{k-2-i}\{\alpha, \beta\}$.

EXAMPLES:

```
sage: set_modsym_print_mode('modular')
sage: M = ModularSymbols(11)
sage: M.modular_symbol([2/11, oo])
-{-1/9, 0}
sage: M.1
{-1/8, 0}
sage: M.modular_symbol([-1/8, 0])
{-1/8, 0}
sage: M.modular_symbol([0, -1/8, 0])
{-1/8, 0}
sage: M.modular_symbol([10, -1/8, 0])
...
ValueError: The first entry of the tuple (=[10, -1/8, 0]) must be an integer between 0 and k

sage: N = ModularSymbols(6,4)
sage: set_modsym_print_mode('manin')
sage: N([1,Cusp(-1/4),Cusp(0)])
17/2*[X^2,(2,3)] - 9/2*[X^2,(2,5)] + 15/2*[X^2,(3,1)] - 15/2*[X^2,(3,2)]
sage: N([1,Cusp(-1/2),Cusp(0)])
1/2*[X^2,(2,3)] + 3/2*[X^2,(2,5)] + 3/2*[X^2,(3,1)] - 3/2*[X^2,(3,2)]
```

Use check=False for efficiency if the input x is a list of length 3 whose first entry is an Integer, and whose second and third entries are cusps:

```
sage: M.modular_symbol([0, Cusp(2/11), Cusp(oo)], check=False)
-(1,9)

sage: set_modsym_print_mode()    # return to default.
```

modular_symbol_sum(*x, check=True*)

Construct a modular symbol sum.

INPUT:

- x (list) – $[f, \alpha, \beta]$ where $f = \sum_{i=0}^{k-2} a_i X^i Y^{k-2-i}$ is a homogeneous polynomial over $\mathbf{Z}$ of degree k and α and β are cusps.
- check (bool, default True) – if True check the validity of the input tuple x

OUTPUT:

The sum $\sum_{i=0}^{k-2} a_i[i, \alpha, \beta]$ as an element of this modular symbol space.

EXAMPLES:

sage: M = ModularSymbols(11,4) sage: R.<X,Y>=QQ[] sage: M.modular_symbol_sum([X*Y,Cusp(0),Cusp(Infinity)]) -3/14*[X^2,(1,6)] + 1/14*[X^2,(1,7)] - 1/14*[X^2,(1,8)] + 1/2*[X^2,(1,9)] - 2/7*[X^2,(1,10)]

modular_symbols_of_sign(*sign*)

Returns a space of modular symbols with the same defining properties (weight, level, etc.) as this space except with given sign.

INPUT:

- sign (int) – A sign ($+1$, -1 or 0).

OUTPUT:

(ModularSymbolsAmbient) A space of modular symbols with the same defining properties (weight, level, etc.) as this space except with given sign.

EXAMPLES:

```
sage: M = ModularSymbols(Gamma0(11),2,sign=0)
sage: M
Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with sign 0 over Rational F
sage: M.modular_symbols_of_sign(-1)
Modular Symbols space of dimension 1 for Gamma_0(11) of weight 2 with sign -1 over Rational
sage: M = ModularSymbols(Gamma1(11),2,sign=0)
sage: M.modular_symbols_of_sign(-1)
Modular Symbols space of dimension 1 for Gamma_1(11) of weight 2 with sign -1 and over Ratio
```

modular_symbols_of_weight(*k*)

Returns a space of modular symbols with the same defining properties (weight, sign, etc.) as this space except with weight k.

INPUT:

- k (int) – A positive integer.

OUTPUT:

(ModularSymbolsAmbient) A space of modular symbols with the same defining properties (level, sign) as this space except with given weight.

EXAMPLES:

```
sage: M = ModularSymbols(Gamma1(6),2,sign=0)
sage: M.modular_symbols_of_weight(3)
Modular Symbols space of dimension 4 for Gamma_1(6) of weight 3 with sign 0 and over Rationa
```

new_submodule(*p=None*)

Returns the new or p-new submodule of this moduluar symbols ambient space.

INPUT:

- p - (default: None); if not None, return only the p-new submodule.

OUTPUT:

The new or p-new submodule of this modular symbols ambient space.

EXAMPLES:

```
sage: ModularSymbols(100).new_submodule()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 31 for Gamma_0
sage: ModularSymbols(389).new_submodule()
Modular Symbols space of dimension 65 for Gamma_0(389) of weight 2 with sign 0 over Rational
```

p1list()

Return a P1list of the level of this modular symbol space.

EXAMPLES:

```
sage: ModularSymbols(11,2).p1list()
The projective line over the integers modulo 11
```

rank()

Returns the rank of this modular symbols ambient space.

OUTPUT:

(int) The rank of this space of modular symbols.

EXAMPLES:

```
sage: M = ModularSymbols(389)
sage: M.rank()
65

sage: ModularSymbols(11,sign=0).rank()
3
sage: ModularSymbols(100,sign=0).rank()
31
sage: ModularSymbols(22,sign=1).rank()
5
sage: ModularSymbols(1,12).rank()
3
sage: ModularSymbols(3,4).rank()
2
sage: ModularSymbols(8,6,sign=-1).rank()
3
```

star_involution()

Return the star involution on this modular symbols space.

OUTPUT:

(matrix) The matrix of the star involution on this space, which is induced by complex conjugation on modular symbols, with respect to the standard basis.

EXAMPLES:

```
sage: ModularSymbols(20,2).star_involution()
Hecke module morphism Star involution on Modular Symbols space of dimension 7 for Gamma_0(20
[1 0 0 0 0 0 0]
[0 1 0 0 0 0 0]
[0 0 0 0 1 0 0]
[0 0 0 1 0 0 0]
[0 0 1 0 0 0 0]
[0 0 0 0 0 1 0]
[0 0 0 0 0 0 1]
Domain: Modular Symbols space of dimension 7 for Gamma_0(20) of weight ...
Codomain: Modular Symbols space of dimension 7 for Gamma_0(20) of weight ...
```

submodule(*M, dual_free_module=None, check=True*)

Return the submodule with given generators or free module M.

INPUT:

- M - either a submodule of this ambient free module, or generators for a submodule;
- **dual_free_module (bool, default None) – this may be** useful to speed up certain calculations; it is the corresponding submodule of the ambient dual module;
- **check (bool, default True) – if True, check that M is** a submodule, i.e. is invariant under all Hecke operators.

OUTPUT:

A subspace of this modular symbol space.

EXAMPLES:

```
sage: M = ModularSymbols(11)
sage: M.submodule([M.0])
...
ValueError: The submodule must be invariant under all Hecke operators.
sage: M.eisenstein_submodule().basis()
((1,0) - 1/5*(1,9),)
sage: M.basis()
((1,0), (1,8), (1,9))
sage: M.submodule([M.0 - 1/5*M.2])
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0(
```

Note: It would make more sense to only check that M is invariant under the Hecke operators with index coprime to the level. Unfortunately, I do not know a reasonable algorithm for determining whether a module is invariant under just the anemic Hecke algebra, since I do not know an analogue of the Sturm bound for the anemic Hecke algebra. - William Stein, 2007-07-27

twisted_winding_element(*i, eps*)

Return the twisted winding element of given degree and character.

INPUT:

- i (int) – an integer, $0 \leq i \leq k-2$ where k is the weight.
- eps (character) – a Dirichlet character

OUTPUT:

(modular symbol) The so-called 'twisted winding element':

$$\sum_{a \in (\mathbf{Z}/m\mathbf{Z})^\times} \varepsilon(a) * [i, 0, a/m].$$

Note: This will only work if the base ring of the modular symbol space contains the character values.

EXAMPLES:

```
sage: eps = DirichletGroup(5)[2]
sage: K = eps.base_ring()
sage: M = ModularSymbols(37,2,0,K)
sage: M.twisted_winding_element(0,eps)
2*(1,23) + (-2)*(1,32) + 2*(1,34)
```

class ModularSymbolsAmbient_wt2_g0(*N, sign, F*)

Modular symbols for $\Gamma_0(N)$ of integer weight 2 over the field F.

INPUT:

- N - int, the level
- sign - int, either -1, 0, or 1

OUTPUT:

The space of modular symbols of weight 2, trivial character, level N and given sign.

EXAMPLES:

```
sage: ModularSymbols(Gamma0(12),2)
Modular Symbols space of dimension 5 for Gamma_0(12) of weight 2 with sign 0 over Rational Field
```

boundary_space()

Return the space of boundary mdular symbols for this space.

EXAMPLES:

```
sage: M = ModularSymbols(100,2)
sage: M.boundary_space()
Space of Boundary Modular Symbols for Congruence Subgroup Gamma0(100) of weight 2 and over R
```

class ModularSymbolsAmbient_wtk_eps(*eps, weight, sign=0*)

boundary_space()

Return the space of boundary modular symbols for this space.

EXAMPLES:

```
sage: eps = DirichletGroup(5).gen(0)
sage: M = ModularSymbols(eps, 2)
sage: M.boundary_space()
Boundary Modular Symbols space of level 5, weight 2, character [zeta4] and dimension 0 over
```

manin_symbols()

Return the Manin symbol list of this modular symbol space.

EXAMPLES:

```
sage: eps = DirichletGroup(5).gen(0)
sage: M = ModularSymbols(eps, 2)
sage: M.manin_symbols()
Manin Symbol List of weight 2 for Gamma1(5) with character [zeta4]
sage: len(M.manin_symbols())
6
```

modular_symbols_of_level(*N*)

Returns a space of modular symbols with the same parameters as this space except with level N.

INPUT:

- N (int) – a positive integer.

OUTPUT:

(Modular Symbol space) A space of modular symbols with the same defining properties (weight, sign, etc.) as this space except with level N.

EXAMPLES:

```
sage: eps = DirichletGroup(5).gen(0)
sage: M = ModularSymbols(eps, 2); M
Modular Symbols space of dimension 0 and level 5, weight 2, character [zeta4], sign 0, over
sage: M.modular_symbols_of_level(15)
Modular Symbols space of dimension 0 and level 15, weight 2, character [1, zeta4], sign 0, o
```

modular_symbols_of_sign(*sign*)

Returns a space of modular symbols with the same defining properties (weight, level, etc.) as this space except with given sign.

INPUT:

- sign (int) – A sign ($+1$, -1 or 0).

OUTPUT:

(ModularSymbolsAmbient) A space of modular symbols with the same defining properties (weight, level, etc.) as this space except with given sign.

EXAMPLES:

```
sage: eps = DirichletGroup(5).gen(0)
sage: M = ModularSymbols(eps, 2); M
Modular Symbols space of dimension 0 and level 5, weight 2, character [zeta4], sign 0, over
sage: M.modular_symbols_of_sign(0) == M
True
sage: M.modular_symbols_of_sign(+1)
Modular Symbols space of dimension 0 and level 5, weight 2, character [zeta4], sign 1, over
sage: M.modular_symbols_of_sign(-1)
Modular Symbols space of dimension 0 and level 5, weight 2, character [zeta4], sign -1, over
```

modular_symbols_of_weight(*k*)

Returns a space of modular symbols with the same defining properties (weight, sign, etc.) as this space except with weight k.

INPUT:

- k (int) – A positive integer.

OUTPUT:

(ModularSymbolsAmbient) A space of modular symbols with the same defining properties (level, sign) as this space except with given weight.

EXAMPLES:

```
sage: eps = DirichletGroup(5).gen(0)
sage: M = ModularSymbols(eps, 2); M
Modular Symbols space of dimension 0 and level 5, weight 2, character [zeta4], sign 0, over
sage: M.modular_symbols_of_weight(3)
Modular Symbols space of dimension 2 and level 5, weight 3, character [zeta4], sign 0, over
sage: M.modular_symbols_of_weight(2) == M
True
```

class **ModularSymbolsAmbient_wtk_g0**(*N, k, sign, F*)

Modular symbols for $\Gamma_0(N)$ of integer weight $k > 2$ over the field F.

For weight 2, it is faster to use ModularSymbols_wt2_g0.

INPUT:

- N - int, the level
- k - integer weight = 2.
- sign - int, either -1, 0, or 1
- F - field

EXAMPLES:

```
sage: ModularSymbols(1,12)
Modular Symbols space of dimension 3 for Gamma_0(1) of weight 12 with sign 0 over Rational Field
sage: ModularSymbols(1,12, sign=1).dimension()
2
sage: ModularSymbols(15,4, sign=-1).dimension()
4
sage: ModularSymbols(6,6).dimension()
10
sage: ModularSymbols(36,4).dimension()
36
```

boundary_space()
Return the space of boundary modular symbols for this space.
EXAMPLES:

```
sage: M = ModularSymbols(100,2)
sage: M.boundary_space()
Space of Boundary Modular Symbols for Congruence Subgroup Gamma0(100) of weight 2 and over R
```

manin_symbols()
Return the Manin symbol list of this modular symbol space.
EXAMPLES:

```
sage: M = ModularSymbols(100,4)
sage: M.manin_symbols()
Manin Symbol List of weight 4 for Gamma0(100)
sage: len(M.manin_symbols())
540
```

modular_symbols_of_level(*N*)
Returns a space of modular symbols with the same parameters as this space except with level N.
INPUT:

- N (int) – a positive integer.

OUTPUT:
(Modular Symbol space) A space of modular symbols with the same defining properties (weight, sign, etc.) as this space except with level N.
For example, if self is the space of modular symbols of weight 2 for $\Gamma_0(22)$, and level is 11, then this function returns the modular symbol space of weight 2 for $\Gamma_0(11)$.
EXAMPLES:

```
sage: M = ModularSymbols(11)
sage: M.modular_symbols_of_level(22)
Modular Symbols space of dimension 7 for Gamma_0(22) of weight 2 with sign 0 over Rational F
sage: M = ModularSymbols(Gamma1(6))
sage: M.modular_symbols_of_level(12)
Modular Symbols space of dimension 9 for Gamma_1(12) of weight 2 with sign 0 and over Ration
```

class **ModularSymbolsAmbient_wtk_g1**(*level, weight, sign, F*)
INPUT:

- level - int, the level
- weight - int, the weight = 2
- sign - int, either -1, 0, or 1
- F - field

EXAMPLES:

```
sage: ModularSymbols(Gamma1(17),2)
Modular Symbols space of dimension 25 for Gamma_1(17) of weight 2 with sign 0 and over Rational
sage: [ModularSymbols(Gamma1(7),k).dimension() for k in [2,3,4,5]]
[5, 8, 12, 16]

sage: ModularSymbols(Gamma1(7),3)
Modular Symbols space of dimension 8 for Gamma_1(7) of weight 3 with sign 0 and over Rational Fi
```

boundary_space()
Return the space of boundary mdular symbols for this space.
EXAMPLES:

```
sage: M = ModularSymbols(100,2)
sage: M.boundary_space()
Space of Boundary Modular Symbols for Congruence Subgroup Gamma0(100) of weight 2 and over R
```

manin_symbols()
Return the Manin symbol list of this modular symbol space.
EXAMPLES:

```
sage: M = ModularSymbols(Gamma1(30),4)
sage: M.manin_symbols()
Manin Symbol List of weight 4 for Gamma1(30)
sage: len(M.manin_symbols())
1728
```

modular_symbols_of_level(N)
Returns a space of modular symbols with the same parameters as this space except with level N.
INPUT:

- N (int) – a positive integer.

OUTPUT:
(Modular Symbol space) A space of modular symbols with the same defining properties (weight, sign, etc.) as this space except with level N.
For example, if self is the space of modular symbols of weight 2 for $\Gamma_0(22)$, and level is 11, then this function returns the modular symbol space of weight 2 for $\Gamma_0(11)$.
EXAMPLES:

```
sage: M = ModularSymbols(Gamma1(30),4); M
Modular Symbols space of dimension 144 for Gamma_1(30) of weight 4 with sign 0 and over Rati
sage: M.modular_symbols_of_level(22)
Modular Symbols space of dimension 90 for Gamma_1(22) of weight 4 with sign 0 and over Ratio
```

class **ModularSymbolsAmbient_wtk_gamma_h**(*group, weight, sign, F*)

boundary_space()
Return the space of boundary modular symbols for this space.
EXAMPLES:

```
sage: M = ModularSymbols(GammaH(15,[4]),2)
sage: M.boundary_space()
Boundary Modular Symbols space for Congruence Subgroup Gamma_H(15) with H generated by [4] o
```

manin_symbols()
Return the Manin symbol list of this modular symbol space.
EXAMPLES:

```
sage: M = ModularSymbols(GammaH(15,[4]),2)
sage: M.manin_symbols()
Manin Symbol List of weight 2 for Congruence Subgroup Gamma_H(15) with H generated by [4]
sage: len(M.manin_symbols())
96
```

modular_symbols_of_level(N)
Returns a space of modular symbols with the same parameters as this space except with level N.
Note: Not implemented for this class.
TESTS:

```
sage: M = ModularSymbols(GammaH(15,[4]),2)
sage: M.modular_symbols_of_level(30)
...
NotImplementedError
```

43.4 Subspace of ambient spaces of modular symbols

class ModularSymbolsSubspace(*ambient_hecke_module, submodule, dual_free_module=None, check=False*)

Subspace of ambient space of modular symbols

boundary_map()

The boundary map to the corresponding space of boundary modular symbols. (This is the restriction of the map on the ambient space.)

EXAMPLES:

```
sage: M = ModularSymbols(1, 24, sign=1) ; M
Modular Symbols space of dimension 3 for Gamma_0(1) of weight 24 with sign 1 over Rational F
sage: M.basis()
([X^18*Y^4,(0,0)], [X^20*Y^2,(0,0)], [X^22,(0,0)])
sage: M.cuspidal_submodule().basis()
([X^18*Y^4,(0,0)], [X^20*Y^2,(0,0)])
sage: M.eisenstein_submodule().basis()
([X^18*Y^4,(0,0)] + 166747/324330*[X^20*Y^2,(0,0)] + 236364091/6742820700*[X^22,(0,0)],)
sage: M.boundary_map()
Hecke module morphism boundary map defined by the matrix
[ 0]
[ 0]
[-1]
Domain: Modular Symbols space of dimension 3 for Gamma_0(1) of weight ...
Codomain: Space of Boundary Modular Symbols for Modular Group SL(2,Z) ...
sage: M.cuspidal_subspace().boundary_map()
Hecke module morphism defined by the matrix
[0]
[0]
Domain: Modular Symbols subspace of dimension 2 of Modular Symbols space ...
Codomain: Space of Boundary Modular Symbols for Modular Group SL(2,Z) ...
sage: M.eisenstein_submodule().boundary_map()
Hecke module morphism defined by the matrix
[-236364091/6742820700]
Domain: Modular Symbols subspace of dimension 1 of Modular Symbols space ...
Codomain: Space of Boundary Modular Symbols for Modular Group SL(2,Z) ...
```

cuspidal_submodule()

Return the cuspidal subspace of this subspace of modular symbols.

EXAMPLES:

```
sage: S = ModularSymbols(42,4).cuspidal_submodule() ; S
Modular Symbols subspace of dimension 40 of Modular Symbols space of dimension 48 for Gamma_
sage: S.is_cuspidal()
True
sage: S.cuspidal_submodule()
Modular Symbols subspace of dimension 40 of Modular Symbols space of dimension 48 for Gamma_
```

The cuspidal submodule of the cuspidal submodule is just itself:

```
sage: S.cuspidal_submodule() is S
True
```

```
sage: S.cuspidal_submodule() == S
True
```

An example where we abuse the _set_is_cuspidal function:

```
sage: M = ModularSymbols(389)
sage: S = M.eisenstein_submodule()
sage: S._set_is_cuspidal(True)
sage: S.cuspidal_submodule()
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 65 for Gamma_0
```

dual_star_involution_matrix()

Return the matrix of the dual star involution, which is induced by complex conjugation on the linear dual of modular symbols.

EXAMPLES:

```
sage: S = ModularSymbols(6,4) ; S.dual_star_involution_matrix()
[ 1  0  0  0  0  0]
[ 0  1  0  0  0  0]
[ 0 -2  1  2  0  0]
[ 0  2  0 -1  0  0]
[ 0 -2  0  2  1  0]
[ 0  2  0 -2  0  1]
sage: S.star_involution().matrix().transpose() == S.dual_star_involution_matrix()
True
```

eisenstein_subspace()

Return the Eisenstein subspace of this space of modular symbols.

EXAMPLES:

```
sage: ModularSymbols(24,4).eisenstein_subspace()
Modular Symbols subspace of dimension 8 of Modular Symbols space of dimension 24 for Gamma_0
sage: ModularSymbols(20,2).cuspidal_subspace().eisenstein_subspace()
Modular Symbols subspace of dimension 0 of Modular Symbols space of dimension 7 for Gamma_0(
```

factorization()

Returns a list of pairs (S, e) where S is simple spaces of modular symbols and self is isomorphic to the direct sum of the S^e as a module over the *anemic* Hecke algebra adjoin the star involution.

The cuspidal S are all simple, but the Eisenstein factors need not be simple.

The factors are sorted by dimension - don't depend on much more for now.

ASSUMPTION: self is a module over the anemic Hecke algebra.

EXAMPLES: Note that if the sign is 1 then the cuspidal factors occur twice, one with each star eigenvalue.

```
sage: M = ModularSymbols(11)
sage: D = M.factorization(); D
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 for Gamma_0
sage: [A.T(2).matrix() for A, _ in D]
[[-2], [3], [-2]]
sage: [A.star_eigenvalues() for A, _ in D]
[[-1], [1], [1]]
```

In this example there is one old factor squared.

```
sage: M = ModularSymbols(22,sign=1)
sage: S = M.cuspidal_submodule()
sage: S.factorization()
(Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 for Gamma_0
```

```
sage: M = ModularSymbols(Gamma0(22), 2, sign=1)
sage: M1 = M.decomposition()[1]
sage: M1.factorization()
Modular Symbols subspace of dimension 3 of Modular Symbols space of dimension 5 for Gamma_0(
```

hecke_bound()

Compute the Hecke bound for self; that is, a number n such that the T_m for m = n generate the Hecke algebra.

EXAMPLES:

```
sage: M = ModularSymbols(24,8)
sage: M.hecke_bound()
53
sage: M.cuspidal_submodule().hecke_bound()
32
sage: M.eisenstein_submodule().hecke_bound()
53
```

is_cuspidal()

Return True if self is cuspidal.

EXAMPLES:

```
sage: ModularSymbols(42,4).cuspidal_submodule().is_cuspidal()
True
sage: ModularSymbols(12,6).eisenstein_submodule().is_cuspidal()
False
```

is_eisenstein()

Return True if self is an Eisenstein subspace.

EXAMPLES:

```
sage: ModularSymbols(22,6).cuspidal_submodule().is_eisenstein()
False
sage: ModularSymbols(22,6).eisenstein_submodule().is_eisenstein()
True
```

star_involution()

Return the star involution on self, which is induced by complex conjugation on modular symbols.

EXAMPLES:

```
sage: M = ModularSymbols(1,24)
sage: M.star_involution()
Hecke module morphism Star involution on Modular Symbols space of dimension 5 for Gamma_0(1)
[ 1  0  0  0  0]
[ 0 -1  0  0  0]
[ 0  0  1  0  0]
[ 0  0  0 -1  0]
[ 0  0  0  0  1]
Domain: Modular Symbols space of dimension 5 for Gamma_0(1) of weight ...
Codomain: Modular Symbols space of dimension 5 for Gamma_0(1) of weight ...
sage: M.cuspidal_subspace().star_involution()
Hecke module morphism defined by the matrix
[ 1  0  0  0]
[ 0 -1  0  0]
[ 0  0  1  0]
[ 0  0  0 -1]
Domain: Modular Symbols subspace of dimension 4 of Modular Symbols space ...
Codomain: Modular Symbols subspace of dimension 4 of Modular Symbols space ...
sage: M.plus_submodule().star_involution()
```

```
Hecke module morphism defined by the matrix
[1 0 0]
[0 1 0]
[0 0 1]
Domain: Modular Symbols subspace of dimension 3 of Modular Symbols space ...
Codomain: Modular Symbols subspace of dimension 3 of Modular Symbols space ...
sage: M.minus_submodule().star_involution()
Hecke module morphism defined by the matrix
[-1  0]
[ 0 -1]
Domain: Modular Symbols subspace of dimension 2 of Modular Symbols space ...
Codomain: Modular Symbols subspace of dimension 2 of Modular Symbols space ...
```

43.5 A single element of an ambient space of modular symbols.

class ModularSymbolsElement (*parent, x, check=True*)

An element of a space of modular symbols.

TESTS:

```
sage: x = ModularSymbols(3, 12).cuspidal_submodule().gen(0)
sage: x == loads(dumps(x))
True
```

list()

Return a list of the coordinates of self in terms of a basis for the ambient space.

EXAMPLE:

```
sage: ModularSymbols(37, 2).0.list()
[1, 0, 0, 0, 0]
```

manin_symbol_rep()

Returns a representation of self as a formal sum of Manin symbols.

(The result is cached for future use.)

EXAMPLE:

```
sage: ModularSymbols(37, 4).0.manin_symbol_rep()
[X^2,(0,1)]
```

modular_symbol_rep()

Returns a representation of self as a formal sum of modular symbols.

(The result is cached for future use.)

EXAMPLE:

```
sage: ModularSymbols(37, 4).0.modular_symbol_rep()
X^2*{0, Infinity}
```

is_ModularSymbolsElement(*x*)

Return True if x is an element of a modular symbols space.

EXAMPLES:

```
sage: sage.modular.modsym.element.is_ModularSymbolsElement(ModularSymbols(11, 2).0)
True
sage: sage.modular.modsym.element.is_ModularSymbolsElement(13)
False
```

set_modsym_print_mode(*mode='manin'*)

Set the mode for printing of elements of modular symbols spaces.

INPUT:

- `mode` - a string. The possibilities are as follows:
- `'manin'` - (the default) formal sums of Manin symbols [P(X,Y),(u,v)]
- `'modular'` - formal sums of Modular symbols P(X,Y)*alpha,beta, where alpha and beta are cusps
- `'vector'` - as vectors on the basis for the ambient space

OUTPUT: none

EXAMPLE:

```
sage: M = ModularSymbols(13, 8)
sage: x = M.0 + M.1 + M.14
sage: set_modsym_print_mode('manin'); x
[X^5*Y,(1,11)] + [X^5*Y,(1,12)] + [X^6,(1,11)]
sage: set_modsym_print_mode('modular'); x
1610510*X^6*{-1/11, 0} - 248832*X^6*{-1/12, 0} + 893101*X^5*Y*{-1/11, 0} - 103680*X^5*Y*{-1/12,
sage: set_modsym_print_mode('vector'); x
(1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0)
sage: set_modsym_print_mode()
```

43.6 Modular symbols {alpha, beta}

The ModularSymbol class represents a single modular symbol $X^i Y^{k-2-i}\{\alpha, \beta\}$.

AUTHOR:

- William Stein (2005, 2009)

TESTS:

```
sage: s = ModularSymbols(11).2.modular_symbol_rep()[0][1]; s
{-1/9, 0}
sage: loads(dumps(s)) == s
True
```

class ModularSymbol(*space, i, alpha, beta*)

The modular symbol $X^i \cdot Y^{k-2-i} \cdot \{\alpha, \beta\}$.

alpha()

For a symbol of the form $X^i Y^{k-2-i}\{\alpha, \beta\}$, return α.

EXAMPLES:

```
sage: s = ModularSymbols(11,4).1.modular_symbol_rep()[0][1]; s
X^2*{-1/6, 0}
sage: s.alpha()
-1/6
sage: type(s.alpha())
<class 'sage.modular.cusps.Cusp'>
```

apply(*g*)

Act on this symbol by the element $g \in \mathrm{GL}_2(\mathbf{Q})$.

INPUT:

- g – a list [a,b,c,d], corresponding to the 2x2 matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix} \in \mathrm{GL}_2(\mathbf{Q})$.

OUTPUT:

- FormalSum – a formal sum $\sum_i c_i x_i$, where c_i are scalars and x_i are ModularSymbol objects, such that the sum $\sum_i c_i x_i$ is the image of this symbol under the action of g. No reduction is performed modulo the relations that hold in self.space().

The action of g on symbols is by

$$P(X,Y)\{\alpha,\beta\} \mapsto P(dX - bY, -cx + aY)\{g(\alpha), g(\beta)\}.$$

Note that for us we have $P = X^i Y^{k-2-i}$, which simplifies computation of the polynomial part slightly.

EXAMPLES:

```
sage: s = ModularSymbols(11,2).1.modular_symbol_rep()[0][1]; s
{-1/8, 0}
sage: a=1;b=2;c=3;d=4; s.apply([a,b,c,d])
{15/29, 1/2}
sage: x = -1/8;  (a*x+b)/(c*x+d)
15/29
sage: x = 0;  (a*x+b)/(c*x+d)
1/2
sage: s = ModularSymbols(11,4).1.modular_symbol_rep()[0][1]; s
X^2*{-1/6, 0}
sage: s.apply([a,b,c,d])
16*X^2*{11/21, 1/2} - 16*X*Y*{11/21, 1/2} + 4*Y^2*{11/21, 1/2}
sage: P = s.polynomial_part()
sage: X,Y = P.parent().gens()
sage: P(d*X-b*Y, -c*X+a*Y)
16*X^2 - 16*X*Y + 4*Y^2
sage: x=-1/6; (a*x+b)/(c*x+d)
11/21
sage: x=0; (a*x+b)/(c*x+d)
1/2
sage: type(s.apply([a,b,c,d]))
<class 'sage.structure.formal_sum.FormalSum'>
```

beta()

For a symbol of the form $X^i Y^{k-2-i}\{\alpha,\beta\}$, return β.

EXAMPLES:

```
sage: s = ModularSymbols(11,4).1.modular_symbol_rep()[0][1]; s
X^2*{-1/6, 0}
sage: s.beta()
0
sage: type(s.beta())
<class 'sage.modular.cusps.Cusp'>
```

i()

For a symbol of the form $X^i Y^{k-2-i}\{\alpha,\beta\}$, return i.

EXAMPLES:

```
sage: s = ModularSymbols(11).2.modular_symbol_rep()[0][1]
sage: s.i()
0
sage: s = ModularSymbols(1,28).0.modular_symbol_rep()[0][1]; s
X^22*Y^4*{0, Infinity}
sage: s.i()
22
```

manin_symbol_rep()
Returns a representation of self as a formal sum of Manin symbols. (The result is not cached.)
EXAMPLES:

```
sage: M = ModularSymbols(11,4)
sage: s = M.1.modular_symbol_rep()[0][1]; s
X^2*{-1/6, 0}
sage: s.manin_symbol_rep()
-[Y^2,(1,1)] - 2*[X*Y,(-1,0)] - [X^2,(-6,1)] - [X^2,(-1,0)]
sage: M(s.manin_symbol_rep()) == M([2,-1/6,0])
True
```

polynomial_part()
Return the polynomial part of this symbol, i.e. for a symbol of the form $X^iY^{k-2-i}\{\alpha,\beta\}$, return X^iY^{k-2-i}.
EXAMPLES:

```
sage: s = ModularSymbols(11).2.modular_symbol_rep()[0][1]
sage: s.polynomial_part()
1
sage: s = ModularSymbols(1,28).0.modular_symbol_rep()[0][1]; s
X^22*Y^4*{0, Infinity}
sage: s.polynomial_part()
X^22*Y^4
```

space()
The list of Manin symbols to which this symbol belongs.
EXAMPLES:

```
sage: s = ModularSymbols(11).2.modular_symbol_rep()[0][1]
sage: s.space()
Manin Symbol List of weight 2 for Gamma0(11)
```

weight()
Return the weight of the modular symbols space to which this symbol belongs; i.e. for a symbol of the form $X^iY^{k-2-i}\{\alpha,\beta\}$, return k.
EXAMPLES:

```
sage: s = ModularSymbols(1,28).0.modular_symbol_rep()[0][1]
sage: s.weight()
28
```

43.7 Manin symbols

This module defines the class ManinSymbol. A Manin Symbol of weight k, level N has the form $[P(X,Y),(u:v)]$ where $P(X,Y) \in \mathbb{Z}[X,Y]$ is homogeneous of weight $k-2$ and $(u:v) \in \mathbb{P}^1(\mathbb{Z}/N\mathbb{Z})$. The ManinSymbol class holds a "monomial Manin Symbol" of the simpler form $[X^iY^{k-2-i},(u:v)]$, which is stored as a triple (i,u,v); the weight and level are obtained from the parent structure, which is a ManinSymbolList.

Integer matrices $[a,b;c,d]$ act on Manin Symbols on the right, sending $[P(X,Y),(u,v)]$ to $[P(aX+bY,cX+dY),(u,v)g]$. Diagonal matrices (with $b=c=0$, such as $I=[-1,0;0,1]$ and $J=[-1,0;0,-1]$) and anti-diagonal matrices (with $a=d=0$, such as $S=[0,-1;1,0]$) map monomiai Manin Symbols to monomial Manin Symbols, up to a scalar factor. For general matrices (such as $T=[0,1,-1,-1]$ and $T^2=[-1,-1;0,1]$) the image of a monomial Manin Symbol is expressed as a formal sum of monomial Manin Symbols, with integer coefficients.

There are various different classes holding lists of Manin symbols of different types. The hierarchy is as follows:

```
class ManinSymbolList(SageObject)

class ManinSymbolList_group(ManinSymbolList)
    class ManinSymbolList_gamma0(ManinSymbolList_group)
    class ManinSymbolList_gamma1(ManinSymbolList_group)
    class ManinSymbolList_gamma_h(ManinSymbolList_group)

class ManinSymbolList_character(ManinSymbolList)
```

class ManinSymbol (*parent, t*)

A Manin symbol $[X^i \cdot Y^{k-2-i}, (u, v)]$.

INPUT:

- `parent` - ManinSymbolList.
- `t` - a 3-tuple (i, u, v) of integers.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,2)
sage: s = ManinSymbol(m,(2,2,3)); s
(2,3)
sage: s == loads(dumps(s))
True

::

sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3)); s
[X^2*Y^4,(2,3)]
```

apply (*a, b, c, d*)

Return the image of self under the matrix $[a, b; c, d]$.

Not implemented for raw ManinSymbol objects, only for members of ManinSymbolLists.

EXAMPLE:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,2)
sage: m.apply(10,[1,0,0,1]) # not implemented for base class
```

copy ()

Return a copy of this ManinSymbol.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3))
sage: s2=s.copy()
sage: s2
[X^2*Y^4,(2,3)]
```

endpoints (*N=None*)

Returns cusps $alpha$, $beta$ such that this Manin symbol, viewed as a symbol for level N, is $X^i * Y^{k-2-i}\{alpha, beta\}$.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3)); s
[X^2*Y^4,(2,3)]
sage: s.endpoints()
(1/3, 1/2)
```

i

Return the i field of this ManinSymbol (i, u, v).

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3))
sage: s._ManinSymbol__get_i()
2
sage: s.i
2
```

level()

Return the level of this ManinSymbol.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3))
sage: s.level()
5
```

lift_to_sl2z(*N=None*)

Returns a lift of this Manin Symbol to $SL_2(\mathbb{Z})$.

If this Manin symbol is (c, d) and N is its level, this function returns a list $[a, b, c', d']$ that defines a 2x2 matrix with determinant 1 and integer entries, such that $c = c'$ (mod N) and $d = d'$ (mod N).

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3))
sage: s
[X^2*Y^4,(2,3)]
sage: s.lift_to_sl2z()
[1, 1, 2, 3]
```

modular_symbol_rep()

Returns a representation of self as a formal sum of modular symbols.

The result is not cached.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3))
sage: s.modular_symbol_rep()
 144*X^6*{1/3, 1/2} - 384*X^5*Y*{1/3, 1/2} + 424*X^4*Y^2*{1/3, 1/2} - 248*X^3*Y^3*{1/3, 1/2}
```

parent()

Return the parent of this ManinSymbol.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3))
sage: s.parent()
Manin Symbol List of weight 8 for Gamma0(5)
```

tuple()

Return the 3-tuple (i, u, v) of this ManinSymbol.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3))
sage: s.tuple()
(2, 2, 3)
```

u

Return the u field of this ManinSymbol (i, u, v).

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3))
sage: s.u # indirect doctest
2
```

v

Return the v field of this ManinSymbol (i, u, v).

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3))
sage: s.v # indirect doctest
3
```

weight()

Return the weight of this ManinSymbol.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: s = ManinSymbol(m,(2,2,3))
sage: s.weight()
8
```

class **ManinSymbolList**(*weight, list*)

Base class for lists of all Manin symbols for a given weight, group or character.

apply(*j, X*)

Apply the matrix $X = [a, b; c, d]$ to the j-th Manin symbol.

Implemented in derived classes.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList
sage: m = ManinSymbolList(6,P1List(11))
sage: m.apply(10, [1,2,0,1])
...
NotImplementedError: Only implemented in derived classes
```

apply_I (j)

Apply the matrix $I = [-1, 0; 0, 1]$ to the j-th Manin symbol.

Implemented in derived classes.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList
sage: m = ManinSymbolList(6,P1List(11))
sage: m.apply_I(10)
...
NotImplementedError: Only implemented in derived classes
```

apply_S (j)

Apply the matrix $S = [0, -1; 1, 0]$ to the j-th Manin symbol.

Implemented in derived classes.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList
sage: m = ManinSymbolList(6,P1List(11))
sage: m.apply_S(10)
...
NotImplementedError: Only implemented in derived classes
```

apply_T (j)

Apply the matrix $T = [0, 1; -1, -1]$ to the j-th Manin symbol.

Implemented in derived classes.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList
sage: m = ManinSymbolList(6,P1List(11))
sage: m.apply_T(10)
...
NotImplementedError: Only implemented in derived classes
```

apply_TT (j)

Apply the matrix $TT = T^2 = [-1, -1; 0, 1]$ to the j-th Manin symbol.

Implemented in derived classes.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList
sage: m = ManinSymbolList(6,P1List(11))
sage: m.apply_TT(10)
...
NotImplementedError: Only implemented in derived classes
```

index (x)

Return the index of x in the list of Manin symbols, where x is a 3-tuple of ints. If x is not in the list, then return -1.

INPUT:

- x - 3-tuple of integers, (i, u, v) defining a valid Manin symbol, which need not be normalized.

OUTPUT:

(int) the index of the normalized Manin symbol equivalent to (i, u, v).

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList
sage: m = ManinSymbolList(6,P1List(11))
sage: m.index(m[2])
```

```
2
sage: all([i == m.index(m[i]) for i in xrange(len(m))])
True
```

manin_symbol(*i*)

Returns the i'th ManinSymbol in this ManinSymbolList.

INPUT:

- `i` - integer, a valid index of a symbol in this list.

OUTPUT:

`ManinSymbol` - the i'th Manin symbol in the list.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList
sage: m = ManinSymbolList(6,P1List(11))
sage: m.manin_symbol(3) # not implemented for base class

sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(6, 4)
sage: s = m.manin_symbol(3); s
[Y^2,(1,2)]
sage: type(s)
<class 'sage.modular.modsym.manin_symbols.ManinSymbol'>
```

manin_symbol_list()

Returns all the ManinSymbols in this ManinSymbolList as a list

Cached for subsequent calls.

OUTPUT:

a list of `ManinSymbol` objects, which is a copy of the complete list of Manin symbols.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList
sage: m = ManinSymbolList(6,P1List(11))
sage: m.manin_symbol_list() # not implemented for the base class

sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(6, 4)
sage: m.manin_symbol_list()
[[Y^2,(0,1)],
[Y^2,(1,0)],
[Y^2,(1,1)],
...
[X^2,(3,1)],
[X^2,(3,2)]]
```

normalize(*x*)

Returns a normalized ManinSymbol from x.

To be implemented in derived classes.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList
sage: m = ManinSymbolList(6,P1List(11))
sage: m.normalize((0,6,7)) # not implemented in base class
```

weight()

Returns the weight of the ManinSymbols in this ManinSymbolList.

OUTPUT:

integer - the weight of the Manin symbols in the list.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(6, 4)
sage: m.weight()
4
```

class ManinSymbolList_character(*character, weight*)

List of Manin Symbols with character.

INPUT:

- character - (DirichletCharacter) the Dirichlet character.
- weight - (integer) the weight.

EXAMPLE:

```
sage: eps = DirichletGroup(4).gen(0)
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_character
sage: m = ManinSymbolList_character(eps,2); m
Manin Symbol List of weight 2 for Gamma1(4) with character [-1]
sage: m.manin_symbol_list()
[(0,1), (1,0), (1,1), (1,2), (1,3), (2,1)]
sage: m == loads(dumps(m))
True
```

apply(*j, m*)

Apply the integer matrix $m = [a, b; c, d]$ to the j-th Manin symbol.

INPUT:

- j (integer): the index of the symbol to act on.
- m (list of ints): $[a, b, c, d]$ where $m = [a, b; c, d]$ is the matrix to be applied.

OUTPUT:

A list of pairs (j, c_i), where each c_i is an integer, j is an integer (the j-th Manin symbol), and the sum $c_i * x_i$ is the image of self under the right action of the matrix $[a, b; c, d]$. Here the right action of $g = [a, b; c, d]$ on a Manin symbol $[P(X, Y), (u, v)]$ is by definition $[P(aX + bY, cX + dY), (u, v) * g]$.

EXAMPLES:

```
sage: eps = DirichletGroup(4).gen(0)
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_character
sage: m = ManinSymbolList_character(eps,4)
sage: m[6]
(1, 0, 1)
sage: m.apply(4, [1,0,0,1])
[(4, 1)]
sage: m.apply(1, [-1,0,0,1])
[(1, -1)]
```

apply_I(*j*)

Apply the matrix $I = [-1, 0, 0, 1]$ to the j-th Manin symbol.

INPUT:

- j - (integer) a symbol index

OUTPUT:

(k, s) where k is the index of the symbol obtained by acting on the j'th symbol with I, and s is the parity of of the j'th symbol.

EXAMPLE:

```
sage: eps = DirichletGroup(4).gen(0)
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_character
sage: m = ManinSymbolList_character(eps,2); m
Manin Symbol List of weight 2 for Gamma1(4) with character [-1]
sage: m.apply_I(4)
(2, -1)
sage: [m.apply_I(i) for i in xrange(len(m))]
[(0, 1), (1, -1), (4, -1), (3, -1), (2, -1), (5, 1)]
```

apply_S (*j*)

Apply the matrix $S = [0, 1; -1, 0]$ to the j-th Manin symbol.

INPUT:

- j - (integer) a symbol index.

OUTPUT:

(k, s) where k is the index of the symbol obtained by acting on the j'th symbol with S, and s is the parity of of the j'th symbol.

EXAMPLE:

```
sage: eps = DirichletGroup(4).gen(0)
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_character
sage: m = ManinSymbolList_character(eps,2); m
Manin Symbol List of weight 2 for Gamma1(4) with character [-1]
sage: m.apply_S(4)
(2, -1)
sage: [m.apply_S(i) for i in xrange(len(m))]
[(1, 1), (0, -1), (4, 1), (5, -1), (2, -1), (3, 1)]
```

apply_T (*j*)

Apply the matrix $T = [0, 1, -1, -1]$ to the j-th Manin symbol.

INPUT:

- j - (integer) a symbol index.

OUTPUT:

A list of pairs (j, c_i), where each c_i is an integer, j is an integer (the j-th Manin symbol), and the sum $c_i * x_i$ is the image of self under the right action of the matrix T.

EXAMPLE:

```
sage: eps = DirichletGroup(4).gen(0)
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_character
sage: m = ManinSymbolList_character(eps,2); m
Manin Symbol List of weight 2 for Gamma1(4) with character [-1]
sage: m.apply_T(4)
[(1, -1)]
sage: [m.apply_T(i) for i in xrange(len(m))]
[[(4, 1)], [(0, -1)], [(3, 1)], [(5, 1)], [(1, -1)], [(2, 1)]]
```

apply_TT (*j*)

Apply the matrix $TT = [-1, -1, 0, 1]$ to the j-th Manin symbol.

INPUT:

- j - (integer) a symbol index

OUTPUT:

A list of pairs (j, c_i), where each c_i is an integer, j is an integer (the j-th Manin symbol), and the sum $c_i * x_i$ is the image of self under the right action of the matrix T^2.

EXAMPLE:

```
sage: eps = DirichletGroup(4).gen(0)
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_character
sage: m = ManinSymbolList_character(eps,2); m
Manin Symbol List of weight 2 for Gamma1(4) with character [-1]
sage: m.apply_TT(4)
[(0, 1)]
sage: [m.apply_TT(i) for i in xrange(len(m))]
[[(1, -1)], [(4, -1)], [(5, 1)], [(2, 1)], [(0, 1)], [(3, 1)]]
```

character()

Return the character of this ManinSymbolList_character object.

OUTPUT:

The Dirichlet character of this Manin symbol list.

EXAMPLE:

```
sage: eps = DirichletGroup(4).gen(0)
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_character
sage: m = ManinSymbolList_character(eps,2); m
Manin Symbol List of weight 2 for Gamma1(4) with character [-1]
sage: m.character()
[-1]
```

index(x)

Returns the index in the list of standard Manin symbols of a symbol that is equivalent, modulo a scalar s, to x. Returns the index and the scalar.

If x is not in the list, return (-1, 0).

INPUT:

- x - 3-tuple of integers (i, u, v), defining an element of this list of Manin symbols, which need not be normalized.

OUTPUT:

(i, s) where i (int) is the index of the Manin symbol equivalent to (i, u, v) (or -1) and s is the scalar (an element of the base field) or the int 0.

EXAMPLE:

```
sage: eps = DirichletGroup(4).gen(0)
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_character
sage: m = ManinSymbolList_character(eps,4); m
Manin Symbol List of weight 4 for Gamma1(4) with character [-1]
sage: [m.index(s.tuple()) for s in m.manin_symbol_list()]
[(0, 1),
(1, 1),
(2, 1),
(3, 1),
...
(16, 1),
(17, 1)]
```

level()

Return the level of this ManinSymbolList.

OUTPUT:

integer - the level of the symbols in this list.

EXAMPLES:

```
sage: eps = DirichletGroup(4).gen(0)
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_character
```

```
sage: ManinSymbolList_character(eps,4).level()
4
```

normalize(*x*)

Returns the normalization of the Manin Symbol `x` with respect to this list, togther with the normalizing scalar.

INPUT:

- `x` - 3-tuple of integers `(i,u,v)`, defining an element of this list of Manin symbols, which need not be normalized.

OUTPUT:

`((i,u,v),s)`, where `(i,u,v)` is the normalized Manin symbol equivalent to `x`, and `s` is the normalizing scalar.

EXAMPLE:

```
sage: eps = DirichletGroup(4).gen(0)
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_character
sage: m = ManinSymbolList_character(eps,4); m
Manin Symbol List of weight 4 for Gamma1(4) with character [-1]
sage: [m.normalize(s.tuple()) for s in m.manin_symbol_list()]
[((0, 0, 1), 1),
((0, 1, 0), 1),
((0, 1, 1), 1),
...
((2, 1, 3), 1),
((2, 2, 1), 1)]
```

class ManinSymbolList_gamma0(*level, weight*)

Class for Manin Symbols for $\Gamma_0(N)$.

INPUT:

- `level` - (integer): the level.
- `weight` - (integer): the weight.

EXAMPLE:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,2); m
Manin Symbol List of weight 2 for Gamma0(5)
sage: m.manin_symbol_list()
[(0,1), (1,0), (1,1), (1,2), (1,3), (1,4)]
sage: m = ManinSymbolList_gamma0(6,4); m
Manin Symbol List of weight 4 for Gamma0(6)
sage: len(m)
36
```

class ManinSymbolList_gamma1(*level, weight*)

Class for Manin Symbols for $\Gamma_1(N)$.

INPUT:

- `level` - (integer): the level.
- `weight` - (integer): the weight.

EXAMPLE:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma1
sage: m = ManinSymbolList_gamma1(5,2); m
Manin Symbol List of weight 2 for Gamma1(5)
sage: m.manin_symbol_list()
[(0,1),
(0,2),
(0,3),
...
(4,3),
(4,4)]
sage: m = ManinSymbolList_gamma1(6,4); m
Manin Symbol List of weight 4 for Gamma1(6)
sage: len(m)
72
sage: m == loads(dumps(m))
True
```

class ManinSymbolList_gamma_h(*group, weight*)

Class for Manin Symbols for $\Gamma_H(N)$.

INPUT:

- `group` - (integer): the congruence subgroup.
- `weight` - (integer): the weight.

EXAMPLE:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma_h
sage: G = GammaH(117, [4])
sage: m = ManinSymbolList_gamma_h(G,2); m
Manin Symbol List of weight 2 for Congruence Subgroup Gamma_H(117) with H generated by [4]
sage: m.manin_symbol_list()[100:110]
[(1,88),
(1,89),
(1,90),
(1,91),
(1,92),
(1,93),
(1,94),
(1,95),
(1,96),
(1,97)]
sage: len(m.manin_symbol_list())
2016
sage: m == loads(dumps(m))
True
```

group()

Return the group associated to self.

EXAMPLES:

```
sage: ModularSymbols(GammaH(12, [5]), 2).manin_symbols().group()
Congruence Subgroup Gamma_H(12) with H generated by [5]
```

class ManinSymbolList_group(*level, weight, syms*)

Base class for Manin symbol lists for a given group.

INPUT:

- `level` - integer level

- `weight` - integer weight
- `syms` - something with a normalize and list method, e.g., P1List.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_group
sage: ManinSymbolList_group(11, 2, P1List(11))
<class 'sage.modular.modsym.manin_symbols.ManinSymbolList_group'>
```

apply (*j*, *m*)

Apply the matrix $m = [a, b; c, d]$ to the j-th Manin symbol.

INPUT:

- `j` - (int) a symbol index
- `m = [a, b, c, d]` a list of 4 integers, which defines a 2x2 matrix.

OUTPUT:

a list of pairs (j_i, α_i), where each α_i is a nonzero integer, j_i is an integer (index of the j_i-th Manin symbol), and $\sum_i \alpha_i x_{j_i}$ is the image of the j-th Manin symbol under the right action of the matrix [a,b;c,d]. Here the right action of g=[a,b;c,d] on a Manin symbol $[P(X, Y), (u, v)]$ is $[P(aX + bY, cX + dY), (u, v)g]$.

EXAMPLE:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: m.apply(40,[2,3,1,1])
[(0, 729), (6, 2916), (12, 4860), (18, 4320), (24, 2160), (30, 576), (36, 64)]
```

apply_I (*j*)

Apply the matrix $I = [-1, 0, 0, 1]$ to the j-th Manin symbol.

INPUT:

- `j` - (int) a symbol index

OUTPUT:

`(k, s)` where k is the index of the symbol obtained by acting on the j'th symbol with I, and s is the parity of of the j'th symbol (a Python `int`, either 1 or -1)

EXAMPLE:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: m.apply_I(4)
(3, 1)
sage: [m.apply_I(i) for i in xrange(10)]
[(0, 1),
(1, 1),
(5, 1),
(4, 1),
(3, 1),
(2, 1),
(6, -1),
(7, -1),
(11, -1),
(10, -1)]
```

apply_S (*j*)

Apply the matrix $S = [0, -1, 1, 0]$ to the j-th Manin symbol.

INPUT:

- `j` - (int) a symbol index

OUTPUT:

`(k, s)` where k is the index of the symbol obtained by acting on the j'th symbol with S, and s is the parity of of the j'th symbol (a Python `int`, either 1 or -1).

EXAMPLE:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: m.apply_S(4)
(40, 1)
sage: [m.apply_S(i) for i in xrange(len(m))]
[(37, 1),
(36, 1),
(41, 1),
(39, 1),
(40, 1),
(38, 1),
(31, -1),
(30, -1),
(35, -1),
(33, -1),
(34, -1),
(32, -1),
...
(4, 1),
(2, 1)]
```

apply_T (*j*)

Apply the matrix $T = [0, 1, -1, -1]$ to the j-th Manin symbol.

INPUT:

- `j` - (int) a symbol index

OUTPUT: see documentation for apply()

EXAMPLE:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: m.apply_T(4)
[(3, 1), (9, -6), (15, 15), (21, -20), (27, 15), (33, -6), (39, 1)]
sage: [m.apply_T(i) for i in xrange(10)]
[[(5, 1), (11, -6), (17, 15), (23, -20), (29, 15), (35, -6), (41, 1)],
[(0, 1), (6, -6), (12, 15), (18, -20), (24, 15), (30, -6), (36, 1)],
[(4, 1), (10, -6), (16, 15), (22, -20), (28, 15), (34, -6), (40, 1)],
[(2, 1), (8, -6), (14, 15), (20, -20), (26, 15), (32, -6), (38, 1)],
[(3, 1), (9, -6), (15, 15), (21, -20), (27, 15), (33, -6), (39, 1)],
[(1, 1), (7, -6), (13, 15), (19, -20), (25, 15), (31, -6), (37, 1)],
[(5, 1), (11, -5), (17, 10), (23, -10), (29, 5), (35, -1)],
[(0, 1), (6, -5), (12, 10), (18, -10), (24, 5), (30, -1)],
[(4, 1), (10, -5), (16, 10), (22, -10), (28, 5), (34, -1)],
[(2, 1), (8, -5), (14, 10), (20, -10), (26, 5), (32, -1)]]
```

apply_TT (*j*)

Apply the matrix $TT = [-1, -1, 0, 1]$ to the j-th Manin symbol.

INPUT:

- `j` - (int) a symbol index

OUTPUT: see documentation for apply()

EXAMPLE:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: m.apply_TT(4)
[(38, 1)]
sage: [m.apply_TT(i) for i in xrange(10)]
[[(37, 1)],
[(41, 1)],
[(39, 1)],
[(40, 1)],
[(38, 1)],
[(36, 1)],
[(31, -1), (37, 1)],
[(35, -1), (41, 1)],
[(33, -1), (39, 1)],
[(34, -1), (40, 1)]]
```

level()

Return the level of this ManinSymbolList.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: ManinSymbolList_gamma0(5,2).level()
5

sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma1
sage: ManinSymbolList_gamma1(51,2).level()
51

sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma_h
sage: ManinSymbolList_gamma_h(GammaH(117, [4]),2).level()
117
```

normalize(*x*)

Returns the normalization of the ModSym `x` with respect to this list.

INPUT:

- `x` - (3-tuple of ints) a tuple defining a ManinSymbol.

OUTPUT:

`(i,u,v)` - (3-tuple of ints) another tuple defining the associated normalized ManinSymbol.

EXAMPLE:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: m = ManinSymbolList_gamma0(5,8)
sage: [m.normalize(s.tuple()) for s in m.manin_symbol_list()][:10]
[(0, 0, 1),
(0, 1, 0),
(0, 1, 1),
(0, 1, 2),
(0, 1, 3),
(0, 1, 4),
(1, 0, 1),
(1, 1, 0),
(1, 1, 1),
(1, 1, 2)]
```

is_ManinSymbol(*x*)

Returns True if `x` is a ManinSymbol.

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbol, ManinSymbolList_gamma0, is_Mani
sage: m = ManinSymbolList_gamma0(6, 4)
sage: is_ManinSymbol(m[3])
False
sage: s = ManinSymbol(m,m[3])
sage: s
[Y^2,(1,2)]
sage: is_ManinSymbol(s)
True
```

43.8 Space of boundary modular symbols.

Used mainly for computing the cuspidal subspace of modular symbols. The space of boundary symbols of sign 0 is isomorphic as a Hecke module to the dual of the space of Eisenstein series, but this does not give a useful method of computing Eisenstein series, since there is no easy way to extract the constant terms.

We represent boundary modular symbols as a sum of Manin symbols of the form $[P, u/v]$, where u/v is a cusp for our group G. The group of boundary modular symbols naturally embeds into a vector space $B_k(G)$ (see Stein, section 8.4, or Merel, section 1.4, where this space is called $\mathbf{C}[\Gamma\backslash\mathbf{Q}]_k$, for a definition), which is a finite dimensional $\mathbf{Q}$ vector space of dimension equal to the number of cusps for G. The embedding takes $[P, u/v]$ to $P(u, v) \cdot [(u, v)]$. We represent the basis vectors by pairs $[(u, v)]$ with u, v coprime. On $B_k(G)$, we have the relations

$$[\gamma \cdot (u, v)] = [(u, v)]$$

for all $\gamma \in G$ and

$$[(\lambda u, \lambda v)] = \mathrm{sign}(\lambda)^k [(u, v)]$$

for all $\lambda \in \mathbf{Q}^\times$.

It's possible for these relations to kill a class, i.e., for a pair $[(u, v)]$ to be 0. For example, when $N = 4$ and $k = 3$ then $(-1, -2)$ is equivalent mod $\Gamma_1(4)$ to $(1, 2)$ since $2 = -2 \bmod 4$ and $1 = -1 \bmod 2$. But since k is odd, $[(-1, -2)]$ is also equivalent to $-[(1, 2)]$. Thus this symbol is equivalent to its negative, hence 0 (notice that this wouldn't be the case in characteristic 2). This happens for any irregular cusp when the weight is odd; there are no irregular cusps on $\Gamma_1(N)$ except when $N = 4$, but there can be more on Γ_H groups. See also prop 2.30 of Stein's Ph.D. thesis.

In addition, in the case that our space is of sign $\sigma = 1$ or -1, we also have the relation $[(-u, v)] = \sigma \cdot [(u, v)]$. This relation can also combine with the above to kill a cusp class - for instance, take (u,v) = (1,3) for $\Gamma_1(5)$. Then since the cusp $\frac{1}{3}$ is $\Gamma_1(5)$-equivalent to the cusp $-\frac{1}{3}$, we have that $[(1, 3)] = [(-1, 3)]$. Now, on the minus subspace, we also have that $[(-1, 3)] = -[(1, 3)]$, which means this class must vanish. Notice that this cannot be used to show that $[(1, 0)]$ or $[(0, 1)]$ is 0.

Note: Special care must be taken when working with the images of the cusps 0 and ∞ in $B_k(G)$. For all cusps *except* 0 and ∞, multiplying the cusp by -1 corresponds to taking $[(u, v)]$ to $[(-u, v)]$ in $B_k(G)$. This means that $[(u, v)]$ is equivalent to $[(-u, v)]$ whenever $\frac{u}{v}$ is equivalent to $-\frac{u}{v}$, except in the case of 0 and ∞. We have the following conditions for $[(1, 0)]$ and $[(0, 1)]$:

- $[(0, 1)] = \sigma \cdot [(0, 1)]$, so $[(0, 1)]$ is 0 exactly when $\sigma = -1$.
- $[(1, 0)] = \sigma \cdot [(-1, 0)]$ and $[(1, 0)] = (-1)^k [(-1, 0)]$, so $[(1, 0)] = 0$ whenever $\sigma \neq (-1)^k$.

Note: For all the spaces of boundary symbols below, no work is done to determine the cusps for G at creation time. Instead, cusps are added as they are discovered in the course of computation. As a result, the rank of a space can change as a computation proceeds.

REFERENCES:

- Merel, "Universal Fourier expansions of modular forms." Springer LNM 1585 (1994), pg. 59-95.
- Stein, "Modular Forms, a computational approach." AMS (2007).

class BoundarySpace(*group=Modular Group SL(2, Z), weight=2, sign=0, base_ring=Rational Field, character=None*)

character()

Return the Dirichlet character assocaited to this space of boundary modular symbols.

EXAMPLES:

```
sage: ModularSymbols(DirichletGroup(7).0, 6).boundary_space().character()
[zeta6]
```

free_module()

Return the underlying free module for self.

EXAMPLES:

```
sage: B = ModularSymbols(Gamma1(7), 5, sign=-1).boundary_space()
sage: B.free_module()
Sparse vector space of dimension 0 over Rational Field
sage: x = B(Cusp(0)) ; y = B(Cusp(1/7)) ; B.free_module()
Sparse vector space of dimension 2 over Rational Field
```

gen(*i=0*)

Return the i-th generator of this space.

EXAMPLES:

```
sage: B = ModularSymbols(Gamma0(24), 4).boundary_space()
sage: B.gen(0)
...
ValueError: only 0 generators known for Space of Boundary Modular Symbols for Congruence Sub
sage: B(Cusp(1/3))
[1/3]
sage: B.gen(0)
[1/3]
```

group()

Return the congruence subgroup associated to this space of boundary modular symbols.

EXAMPLES:

```
sage: ModularSymbols(GammaH(14,[3]), 2).boundary_space().group()
Congruence Subgroup Gamma_H(14) with H generated by [3]
```

is_ambient()

Return True if self is a space of boundary symbols associated to an ambient space of modular symbols.

EXAMPLES:

```
sage: M = ModularSymbols(Gamma1(6), 4)
sage: M.is_ambient()
True
sage: M.boundary_space().is_ambient()
True
```

rank()

The rank of the space generated by boundary symbols that have been found so far in the course of computing the boundary map.

> **Warning:** This number may change as more elements are coerced into this space!! (This is an implementation detail that will likely change.)

EXAMPLES:

```
sage: M = ModularSymbols(Gamma0(72), 2) ; B = M.boundary_space()
sage: B.rank()
0
sage: _ = [ B(x) for x in M.basis() ]
sage: B.rank()
16
```

sign()

Return the sign of the complex conjugation involution on this space of boundary modular symbols.

EXAMPLES:

```
sage: ModularSymbols(13,2,sign=-1).boundary_space().sign()
-1
```

weight()

Return the weight of this space of boundary modular symbols.

EXAMPLES:

```
sage: ModularSymbols(Gamma1(9), 5).boundary_space().weight()
5
```

class **BoundarySpaceElement**(*parent, x*)

coordinate_vector()

Return self as a vector on the QQ-vector space with basis self.parent()._known_cusps().

EXAMPLES:

```
sage: B = ModularSymbols(18,4,sign=1).boundary_space()
sage: x = B(Cusp(1/2)) ; x
[1/2]
sage: x.coordinate_vector()
(1)
sage: ((18/5)*x).coordinate_vector()
(18/5)
sage: B(Cusp(0))
[0]
sage: x.coordinate_vector()
(1)
sage: x = B(Cusp(1/2)) ; x
[1/2]
sage: x.coordinate_vector()
(1, 0)
```

class **BoundarySpace_wtk_eps**(*eps, weight, sign=0*)

class **BoundarySpace_wtk_g0**(*level, weight, sign, F*)

class **BoundarySpace_wtk_g1**(*level, weight, sign, F*)

class **BoundarySpace_wtk_gamma_h**(*group, weight, sign, F*)

43.9 Heilbronn matrix computation

class **Heilbronn**()

apply()

Return a list of pairs ((c,d),m), which is obtained as follows: 1) Compute the images (a,b) of the vector (u,v) (mod N) acted on by each of the HeilbronnCremona matrices in self. 2) Reduce each (a,b) to canonical form (c,d) using p1normalize 3) Sort. 4) Create the list ((c,d),m), where m is the number of times that (c,d) appears in the list created in steps 1-3 above. Note that the pairs ((c,d),m) are sorted lexicographically by (c,d).

INPUT:

- `u, v, N` - integers

OUTPUT: list

EXAMPLES:

```
sage: H = sage.modular.modsym.heilbronn.HeilbronnCremona(2); H
The Cremona-Heilbronn matrices of determinant 2
sage: H.apply(1,2,7)
[((1, 5), 1), ((1, 6), 1), ((1, 1), 1), ((1, 4), 1)]
```

to_list()

Return the list of Heilbronn matrices corresponding to self. Each matrix is given as a list of four ints.

EXAMPLES:

```
sage: H = HeilbronnCremona(2); H
The Cremona-Heilbronn matrices of determinant 2
sage: H.to_list()
[[1, 0, 0, 2], [2, 0, 0, 1], [2, 1, 0, 1], [1, 0, 1, 2]]
```

class **HeilbronnCremona**()

p

class **HeilbronnMerel**()

n

hecke_images_gamma0_weight2()

INPUT:

- `u, v, N` - integers so that gcd(u,v,N) = 1
- `indices` - a list of positive integers
- `R` - matrix over QQ that writes each elements of P1 = P1List(N) in terms of a subset of P1.

OUTPUT: a dense matrix with rational entries whose columns are the images T_n(x) for n in indices and x the Manin symbol (u,v), expressed in terms of the basis.

EXAMPLES:

```
sage: M = ModularSymbols(23,2,1)
sage: A = sage.modular.modsym.heilbronn.hecke_images_gamma0_weight2(1,0,23,[1..6],M.manin_gens_t
sage: A
[ 1  0  0]
[ 3  0 -1]
[ 4 -2 -1]
[ 7 -2 -2]
[ 6  0 -2]
[12 -2 -4]
sage: z = M((1,0))
sage: [M.T(n)(z).element() for n in [1..6]]
[(1, 0, 0), (3, 0, -1), (4, -2, -1), (7, -2, -2), (6, 0, -2), (12, -2, -4)]
```

hecke_images_gamma0_weight_k()

INPUT:

- u, v, N - integers so that gcd(u,v,N) = 1
- i - integer with 0 = i = k-2
- k - weight
- indices - a list of positive integers
- R - matrix over QQ that writes each elements of P1 = P1List(N) in terms of a subset of P1.

OUTPUT: a dense matrix with rational entries whose columns are the images T_n(x) for n in indices and x the Manin symbol $[X^i * Y^(k-2-i), (u,v)]$, expressed in terms of the basis.

EXAMPLES:

```
sage: M = ModularSymbols(15,6,sign=-1)
sage: R = M.manin_gens_to_basis()
sage: sage.modular.modsym.heilbronn.hecke_images_gamma0_weight_k(4,1,3,15,6,[1,11,12], R)
[       0        0      1/8     -1/8        0        0        0        0]
[-4435/22 -1483/22     -112 -4459/22  2151/22 -5140/11  4955/22  2340/11]
[ 1253/22  1981/22       -2  3177/22 -1867/22  6560/11 -7549/22  -612/11]
sage: x = M((3,4,1)) ; x.element()
(0, 0, 1/8, -1/8, 0, 0, 0, 0)
sage: M.T(11)(x).element()
(-4435/22, -1483/22, -112, -4459/22, 2151/22, -5140/11, 4955/22, 2340/11)
sage: M.T(12)(x).element()
(1253/22, 1981/22, -2, 3177/22, -1867/22, 6560/11, -7549/22, -612/11)
```

hecke_images_nonquad_character_weight2()

Return images of the Hecke operators T_n for n in the list indices, where chi must be a quadratic Dirichlet character with values in QQ.

R is assumed to be the relation matrix of a weight modular symbols space over QQ with character chi.

INPUT:

- u, v, N - integers so that gcd(u,v,N) = 1
- indices - a list of positive integers
- chi - a Dirichlet character that takes values in a nontrivial extension of QQ.
- R - matrix over QQ that writes each elements of P1 = P1List(N) in terms of a subset of P1.

OUTPUT: a dense matrix with entries in the field QQ(chi) (the values of chi) whose columns are the images T_n(x) for n in indices and x the Manin symbol (u,v), expressed in terms of the basis.

EXAMPLES:

```
sage: chi = DirichletGroup(13).0^2
sage: M = ModularSymbols(chi)
sage: eps = M.character()
sage: R = M.manin_gens_to_basis()
sage: sage.modular.modsym.heilbronn.hecke_images_nonquad_character_weight2(1,0,13,[1,2,6],eps,R)
[           1            0            0            0]
[   zeta6 + 2            0            0           -1]
[           7 -2*zeta6 + 1   -zeta6 - 1     -2*zeta6]
sage: x = M((1,0)); x.element()
(1, 0, 0, 0)
sage: M.T(2)(x).element()
(zeta6 + 2, 0, 0, -1)
sage: M.T(6)(x).element()
(7, -2*zeta6 + 1, -zeta6 - 1, -2*zeta6)
```

hecke_images_quad_character_weight2()

INPUT:

- u, v, N - integers so that gcd(u,v,N) = 1
- indices - a list of positive integers
- chi - a Dirichlet character that takes values in QQ
- R - matrix over QQ(chi) that writes each elements of P1 = P1List(N) in terms of a subset of P1.

OUTPUT: a dense matrix with entries in the rational field QQ (the values of chi) whose columns are the images T_n(x) for n in indices and x the Manin symbol (u,v), expressed in terms of the basis.

EXAMPLES:

```
sage: chi = DirichletGroup(29,QQ).0
sage: M = ModularSymbols(chi)
sage: R = M.manin_gens_to_basis()
sage: sage.modular.modsym.heilbronn.hecke_images_quad_character_weight2(2,1,29,[1,3,4],chi,R)
[ 0  0  0  0  0 -1]
[ 0  1  0  1  1  1]
[ 0 -2  0  2 -2 -1]
sage: x = M((2,1)) ; x.element()
(0, 0, 0, 0, 0, -1)
sage: M.T(3)(x).element()
(0, 1, 0, 1, 1, 1)
sage: M.T(4)(x).element()
(0, -2, 0, 2, -2, -1)
```

43.10 List of Elements of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$

class P1List()

The class for $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$, the projective line modulo N.

EXAMPLES:

```
sage: P = P1List(12); P
The projective line over the integers modulo 12
sage: list(P)
[(0, 1), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), (1, 9), (1, 10)]
```

Saving and loading works.

```
sage: loads(dumps(P)) == P
True
```

N()

Returns the level or modulus of this P1List.

EXAMPLES:

```
sage: L = P1List(120)
sage: L.N()
120
```

apply_I()

Return the index of the result of applying the matrix $I = [-1, 0; 0, 1]$ to the i'th element of this P1List.

INPUT:

- i - integer (the index of the element to act on).

EXAMPLES:

```
sage: L = P1List(120)
sage: L[10]
(1, 9)
sage: L.apply_I(10)
112
sage: L[112]
(1, 111)
sage: L.normalize(-1,9)
(1, 111)

This operation is an involution:

sage: all([L.apply_I(L.apply_I(i))==i for i in xrange(len(L))])
True
```

apply_S()

Return the index of the result of applying the matrix $S = [0, -1; 1, 0]$ to the i'th element of this P1List.

INPUT:

- i - integer (the index of the element to act on).

EXAMPLES:

```
sage: L = P1List(120)
sage: L[10]
(1, 9)
sage: L.apply_S(10)
159
sage: L[159]
(3, 13)
sage: L.normalize(-9,1)
(3, 13)

This operation is an involution:

sage: all([L.apply_S(L.apply_S(i))==i for i in xrange(len(L))])
True
```

apply_T()

Return the index of the result of applying the matrix $T = [0, 1; -1, -1]$ to the i'th element of this P1List.

INPUT:

- i - integer (the index of the element to act on).

EXAMPLES:

```
sage: L = P1List(120)
sage: L[10]
(1, 9)
sage: L.apply_T(10)
157
sage: L[157]
(3, 10)
sage: L.normalize(9,-10)
(3, 10)

This operation has order three:

sage: all([L.apply_T(L.apply_T(L.apply_T(i)))==i for i in xrange(len(L))])
True
```

index()

Returns the index of the class of (u, v) in the fixed list of representatives of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$.

INPUT:

•u, v - integers, with $\gcd(u, v, N) = 1$.

OUTPUT:

•i - the index of u, v, in the P1list.

EXAMPLES:

```
sage: L = P1List(120)
sage: L[100]
(1, 99)
sage: L.index(1,99)
100
sage: all([L.index(L[i][0],L[i][1])==i for i in range(len(L))])
True
```

index_of_normalized_pair()

Returns the index of the class of (u, v) in the fixed list of representatives of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$.

INPUT:

•u, v - integers, with $\gcd(u, v, N) = 1$, normalized so they lie in the list.

OUTPUT:

•i - the index of $(u : v)$, in the P1list.

EXAMPLES:

```
sage: L = P1List(120)
sage: L[100]
(1, 99)
sage: L.index_of_normalized_pair(1,99)
100
sage: all([L.index_of_normalized_pair(L[i][0],L[i][1])==i for i in range(len(L))])
True
```

lift_to_sl2z()

Lift the i'th element of this P1list to an element of $SL(2, \mathbf{Z})$.

If the i'th element is (c, d), this function computes and returns a list $[a, b, c', d']$ that defines a 2x2 matrix with determinant 1 and integer entries, such that $c = c'$ (mod N) and $d = d'$ (mod N).

INPUT:

•i - integer (the index of the element to lift).

EXAMPLES:

```
sage: p = P1List(11)
sage: p.list()[3]
(1, 2)

sage: p.lift_to_sl2z(3)
[0, -1, 1, 2]
```

AUTHORS:

•Justin Walker

list()

Returns the underlyng list of this P1List object.

EXAMPLES:

```
sage: L = P1List(8)
sage: type(L)
<type 'sage.modular.modsym.p1list.P1List'>
sage: type(L.list())
<type 'list'>
```

normalize()

Returns a normalised element of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$.

INPUT:

- u, v - integers, with $\gcd(u, v, N) = 1$.

OUTPUT:

- a 2-tuple (uu, vv) where $(uu : vv)$ is a *normalized* representative of $(u : v)$.

NOTE: See also normalize_with_scalar() which also returns the normalizing scalar.

EXAMPLES:

```
sage: L = P1List(120)
sage: (u,v) = (555555555,7777)
sage: uu,vv = L.normalize(555555555,7777)
sage: (uu,vv)
(15, 13)
sage: (uu*v-vv*u) % L.N() == 0
True
```

normalize_with_scalar()

Returns a normalised element of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$, together with the normalizing scalar.

INPUT:

- u, v - integers, with $\gcd(u, v, N) = 1$.

OUTPUT:

- a 3-tuple (uu, vv, ss) where $(uu : vv)$ is a *normalized* representative of $(u : v)$, and ss is a scalar such that $(ss * uu, ss * vv) = (u, v) \pmod N$.

EXAMPLES:

```
sage: L = P1List(120)
sage: (u,v) = (555555555,7777)
sage: uu,vv,ss = L.normalize_with_scalar(555555555,7777)
sage: (uu,vv)
(15, 13)
sage: ((ss*uu-u)%L.N(), (ss*vv-v)%L.N())
(0, 0)
sage: (uu*v-vv*u) % L.N() == 0
True
```

class **export**()

lift_to_sl2z()

Return a list of Python ints $[a, b, c', d']$ that are the entries of a 2x2 matrix with determinant 1 and lower two entries congruent to c, d modulo N.

INPUT:

- c, d, N - integers such that $\gcd(c, d, N) = 1$.

EXAMPLES:

```
sage: lift_to_sl2z(2,3,6)
[1, 1, 2, 3]
sage: lift_to_sl2z(15,6,24)
[-2, -17, 15, 126]
sage: lift_to_sl2z(15,6,2400000)
[-2L, -320001L, 15L, 2400006L]
```

lift_to_sl2z_int()

Lift a pair (c, d) to an element of $SL(2, \mathbf{Z})$.

(c, d) is assumed to be an element of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$. This function computes and returns a list $[a, b, c', d']$ that defines a 2x2 matrix, with determinant 1 and integer entries, such that $c = c' \pmod N$ and $d = d' \pmod N$.

INPUT:

- c, d, N - integers such that $\gcd(c, d, N) = 1$.

EXAMPLES:

```
sage: from sage.modular.modsym.p1list import lift_to_sl2z_int
sage: lift_to_sl2z_int(2,6,11)
[1, 8, 2, 17]
sage: m=Matrix(Integers(),2,2,lift_to_sl2z_int(2,6,11))
sage: m
[ 1  8]
[ 2 17]
```

AUTHOR:

- Justin Walker

lift_to_sl2z_llong()

Lift a pair (c, d) (modulo N) to an element of $SL(2, \mathbf{Z})$.

(c, d) is assumed to be an element of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$. This function computes and returns a list $[a, b, c', d']$ that defines a 2x2 matrix, with determinant 1 and integer entries, such that $c = c' \pmod N$ and $d = d' \pmod N$.

INPUT:

- c, d, N - integers such that $\gcd(c, d, N) = 1$.

EXAMPLES:

```
sage: from sage.modular.modsym.p1list import lift_to_sl2z_llong
sage: lift_to_sl2z_llong(2,6,11)
[1L, 8L, 2L, 17L]
sage: m=Matrix(Integers(),2,2,lift_to_sl2z_llong(2,6,11))
sage: m
[ 1  8]
[ 2 17]
```

AUTHOR:

- Justin Walker

p1_normalize()

Computes the canonical representative of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$ equivalent to (u, v) along with a transforming scalar.

INPUT:

- N - an integer
- u - an integer

- v - an integer

OUTPUT: If gcd(u,v,N) = 1, then returns

- uu - an integer
- vv - an integer
- ss - an integer such that $(ss * uu, ss * vv)$ is equivalent to (u, v) mod N; if $\gcd(u, v, N) \neq 1$, returns 0, 0, 0.

EXAMPLES:

```
sage: from sage.modular.modsym.p1list import p1_normalize
sage: p1_normalize(90,7,77)
(1, 11, 7)
sage: p1_normalize(90,7,78)
(1, 24, 7)
sage: (7*24-78*1) % 90
0
sage: (7*24) % 90
78
```

```
sage: from sage.modular.modsym.p1list import p1_normalize
sage: p1_normalize(50001,12345,54322)
(3, 4667, 4115)
sage: (12345*4667-54321*3) % 50001
3
sage: 4115*3 % 50001
12345
sage: 4115*4667 % 50001 == 54322 % 50001
True
```

p1_normalize_int()

Computes the canonical representative of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$ equivalent to (u, v) along with a transforming scalar.

INPUT:

- N - an integer
- u - an integer
- v - an integer

OUTPUT: If gcd(u,v,N) = 1, then returns

- uu - an integer
- vv - an integer
- ss - an integer such that $(ss * uu, ss * vv)$ is congruent to (u, v) (mod N); if $\gcd(u, v, N) \neq 1$, returns 0, 0, 0.

EXAMPLES:

```
sage: from sage.modular.modsym.p1list import p1_normalize_int
sage: p1_normalize_int(90,7,77)
(1, 11, 7)
sage: p1_normalize_int(90,7,78)
(1, 24, 7)
sage: (7*24-78*1) % 90
0
sage: (7*24) % 90
78
```

p1_normalize_llong()

Computes the canonical representative of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$ equivalent to (u, v) along with a transforming scalar.

INPUT:

- N - an integer
- u - an integer
- v - an integer

OUTPUT: If gcd(u,v,N) = 1, then returns

- uu - an integer
- vv - an integer
- ss - an integer such that $(ss * uu, ss * vv)$ is equivalent to (u, v) mod N;

  if $\gcd(u, v, N) \neq 1$, returns 0, 0, 0.

EXAMPLES:

```
sage: from sage.modular.modsym.p1list import p1_normalize_llong
sage: p1_normalize_llong(90000,7,77)
(1, 11, 7)
sage: p1_normalize_llong(90000,7,78)
(1, 77154, 7)
sage: (7*77154-78*1) % 90000
0
sage: (7*77154) % 90000
78
```

p1list()

Returns the elements of the projective line modulo N, $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$, as a plain list of 2-tuples.

INPUT:

- N (integer) - a positive integer (less than 2^31).

OUTPUT:

A list of the elements of the projective line $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$, as plain 2-tuples.

EXAMPLES:

```
sage: from sage.modular.modsym.p1list import p1list
sage: list(p1list(7))
[(0, 1), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4), (1, 5), (1, 6)]
sage: N=23456
sage: len(p1list(N)) == N*prod([1+1/p for p,e in N.factor()])
True
```

p1list_int()

Returns a list of the normalized elements of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$.

INPUT:

- N - integer (the level or modulus).

EXAMPLES:

```
sage: from sage.modular.modsym.p1list import p1list_int
sage: p1list_int(6)
[(0, 1),
(1, 0),
(1, 1),
(1, 2),
(1, 3),
(1, 4),
(1, 5),
(2, 1),
(2, 3),
(2, 5),
(3, 1),
(3, 2)]


sage: p1list_int(120)
[(0, 1),
(1, 0),
(1, 1),
(1, 2),
(1, 3),
...
(30, 7),
(40, 1),
(40, 3),
(40, 11),
(60, 1)]
```

p1list_llong()

Returns a list of the normalized elements of $\mathbb{P}^1(\mathbf{Z}/N\mathbf{Z})$, as a plain list of 2-tuples.

INPUT:

- N - integer (the level or modulus).

EXAMPLES:

```
sage: from sage.modular.modsym.p1list import p1list_llong
sage: N = 50000
sage: L = p1list_llong(50000)
sage: len(L) == N*prod([1+1/p for p,e in N.factor()])
True
sage: L[0]
(0, 1)
sage: L[len(L)-1]
(25000, 1)
```

43.11 List of coset represenatives for $\Gamma_1(N)$ in $\mathrm{SL}_2(\mathbf{Z})$.

class G1list(N)

A class representing a list of coset representatives for $\Gamma_1(N)$ in $\mathrm{SL}_2(\mathbf{Z})$. What we actually calculate is a list of elements of $(\mathbf{Z}/N\mathbf{Z})^2$ of exact order N.

TESTS:

```
sage: L = sage.modular.modsym.g1list.G1list(18)
sage: loads(dumps(L)) == L
True
```

list()

Return a list of vectors representing the cosets. Do not change the returned list!

EXAMPLE:

```
sage: L = sage.modular.modsym.g1list.G1list(4); L.list()
[(0, 1), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 1), (2, 3), (3, 0), (3, 1), (3, 2), (3,
```

normalize(*u*, *v*)

Given a pair (u, v) of integers, return the unique pair (u', v') such that the pair (u', v') appears in `self.list()` and (u, v) is equivalent to (u', v'). This is rather trivial, but is here for consistency with the `P1List` class which is the equivalent for Γ_0 (where the problem is rather harder).

This will only make sense if $\gcd(u, v, N) = 1$; otherwise the output will not be an element of self.

EXAMPLE:

```
sage: L = sage.modular.modsym.g1list.G1list(4); L.normalize(6, 1)
(2, 1)
sage: L = sage.modular.modsym.g1list.G1list(4); L.normalize(6, 2) # nonsense!
(2, 2)
```

43.12 List of coset represenatives for $\Gamma_H(N)$ in $\mathrm{SL}_2(\mathbf{Z})$.

class GHlist(*group*)

A class representing a list of coset representatives for $\Gamma_H(N)$ in $\mathrm{SL}_2(\mathbf{Z})$.

TESTS:

```
sage: L = sage.modular.modsym.ghlist.GHlist(GammaH(18,[13]))
sage: loads(dumps(L)) == L
True
```

list()

Return a list of vectors representing the cosets. Do not change the returned list!

EXAMPLE:

```
sage: L = sage.modular.modsym.ghlist.GHlist(GammaH(4,[])); L.list()
[(0, 1), (0, 3), (1, 0), (1, 1), (1, 2), (1, 3), (2, 1), (2, 3), (3, 0), (3, 1), (3, 2), (3,
```

normalize(*u*, *v*)

Given a pair (u, v) of integers, return the unique pair (u', v') such that the pair (u', v') appears in `self.list()` and (u, v) is equivalent to (u', v').

This will only make sense if $\gcd(u, v, N) = 1$; otherwise the output will not be an element of self.

EXAMPLES:

```
sage: sage.modular.modsym.ghlist.GHlist(GammaH(24, [17, 19])).normalize(17, 6)
(1, 6)
sage: sage.modular.modsym.ghlist.GHlist(GammaH(24, [7, 13])).normalize(17, 6)
(5, 6)
sage: sage.modular.modsym.ghlist.GHlist(GammaH(24, [5, 23])).normalize(17, 6)
(7, 18)
```

43.13 Relation matrices for ambient modular symbols spaces

This file contains functions that are used by the various ambient modular symbols classes to compute presentations of spaces in terms of generators and relations, using the standard methods based on Manin symbols.

T_relation_matrix_wtk_g0 (*syms, mod, field, sparse*)

Compute a matrix whose echelon form gives the quotient by 3-term T relations. Despite the name, this is used for all modular symbols spaces (including those with character and those for Γ_1 and Γ_H groups), not just Γ_0.

INPUT:

- `syms` - ManinSymbols
- `mod` - list that gives quotient modulo some two-term relations, i.e., the S relations, and if sign is nonzero, the I relations.
- `field` - base_ring
- `sparse` - (True or False) whether to use sparse rather than dense linear algebra

OUTPUT: A sparse matrix whose rows correspond to the reduction of the T relations modulo the S and I relations.

EXAMPLE:

```
sage: from sage.modular.modsym.relation_matrix import *
sage: L = sage.modular.modsym.manin_symbols.ManinSymbolList_gamma_h(GammaH(36, [17,19]), 2)
sage: modS = sparse_2term_quotient(modS_relations(L), 216, QQ)
sage: T_relation_matrix_wtk_g0(L, modS, QQ, False)
72 x 216 dense matrix over Rational Field
sage: T_relation_matrix_wtk_g0(L, modS, GF(17), True)
72 x 216 sparse matrix over Finite Field of size 17
```

compute_presentation (*syms, sign, field, sparse=None*)

Compute the presentation for self, as a quotient of Manin symbols modulo relations.

INPUT:

- `syms` - manin_symbols.ManinSymbols
- `sign` - integer (-1, 0, 1)
- `field` - a field

OUTPUT:

- sparse matrix whose rows give each generator in terms of a basis for the quotient
- list of integers that give the basis for the quotient
- mod: list where mod[i]=(j,s) means that x_i = s*x_j modulo the 2-term S (and possibly I) relations.

ALGORITHM:

1. Let $S = [0, -1; 1, 0], T = [0, -1; 1, -1]$, and $I = [-1, 0; 0, 1]$.
2. Let $x_0, \ldots, x_{n-1}$ by a list of all non-equivalent Manin symbols.
3. Form quotient by 2-term S and (possibly) I relations.
4. Create a sparse matrix A with m columns, whose rows encode the relations

$$[x_i] + [x_i T] + [x_i T^2] = 0.$$

There are about n such rows. The number of nonzero entries per row is at most 3*(k-1). Note that we must include rows for *all* i, since even if $[x_i] = [x_j]$, it need not be the case that $[x_i T] = [x_j T]$, since S and T do not commute. However, in many cases we have an a priori formula for the dimension of the quotient by all these relations, so we can omit many relations and just check that there are enough at the end–if there aren't, we add in more.

5. Compute the reduced row echelon form of A using sparse Gaussian elimination.

6. Use what we've done above to read off a sparse matrix R that uniquely expresses each of the n Manin symbols in terms of a subset of Manin symbols, modulo the relations. This subset of Manin symbols is a basis for the quotient by the relations.

EXAMPLE:

```
sage: L = sage.modular.modsym.manin_symbols.ManinSymbolList_gamma0(8,2)
sage: sage.modular.modsym.relation_matrix.compute_presentation(L, 1, GF(9,'a'), True)
([2 0 0]
[1 0 0]
[0 0 0]
[0 2 0]
[0 0 0]
[0 0 2]
[0 0 0]
[0 2 0]
[0 0 0]
[0 1 0]
[0 1 0]
[0 0 1], [1, 9, 11], [(1, 2), (1, 1), (0, 0), (9, 2), (0, 0), (11, 2), (0, 0), (9, 2), (0, 0), (
```

gens_to_basis_matrix(*syms, relation_matrix, mod, field, sparse*)

Compute echelon form of 3-term relation matrix, and read off each generator in terms of basis.

INPUT:

- `syms` - a ManinSymbols object
- `relation_matrix` - as output by `__compute_T_relation_matrix(self, mod)`
- `mod` - quotient of modular symbols modulo the 2-term S (and possibly I) relations
- `field` - base field
- `sparse` - (bool): whether or not matrix should be sparse

OUTPUT:

- `matrix` - a matrix whose ith row expresses the Manin symbol generators in terms of a basis of Manin symbols (modulo the S, (possibly I,) and T rels) Note that the entries of the matrix need not be integers.
- `list` - integers i, such that the Manin symbols x_i are a basis.

EXAMPLE:

```
sage: from sage.modular.modsym.relation_matrix import *
sage: L = sage.modular.modsym.manin_symbols.ManinSymbolList_gamma1(4, 3)
sage: modS = sparse_2term_quotient(modS_relations(L), 24, GF(3))
sage: gens_to_basis_matrix(L, T_relation_matrix_wtk_g0(L, modS, GF(3), 24), modS, GF(3), True)
(24 x 2 sparse matrix over Finite Field of size 3, [13, 23])
```

modI_relations(*syms, sign*)

Compute quotient of Manin symbols by the I relations.

INPUT:

- `syms` - ManinSymbols
- `sign` - int (either -1, 0, or 1)

OUTPUT:

- `rels` - set of pairs of pairs (j, s), where if mod[i] = (j,s), then x_i = s*x_j (mod S relations)

EXAMPLE:

```
sage: L = sage.modular.modsym.manin_symbols.ManinSymbolList_gamma1(4, 3)
sage: sage.modular.modsym.relation_matrix.modI_relations(L, 1)
set([((14, 1), (20, 1)), ((0, 1), (0, -1)), ((7, 1), (7, -1)), ((9, 1), (3, -1)), ((3, 1), (9, -
```

Warning: We quotient by the involution eta((u,v)) = (-u,v), which has the opposite sign as the involution in Merel's Springer LNM 1585 paper! Thus our +1 eigenspace is his -1 eigenspace, etc. We do this for consistency with MAGMA.

modS_relations (*syms*)

Compute quotient of Manin symbols by the S relations.

Here S is the 2x2 matrix [0, -1; 1, 0].

INPUT:

- `syms` - manin_symbols.ManinSymbols

OUTPUT:

- `rels` - set of pairs of pairs (j, s), where if mod[i] = (j,s), then x_i = s*x_j (mod S relations)

EXAMPLES:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma0
sage: from sage.modular.modsym.relation_matrix import modS_relations

sage: syms = ManinSymbolList_gamma0(2, 4); syms
Manin Symbol List of weight 4 for Gamma0(2)
sage: modS_relations(syms)
set([((3, -1), (4, 1)), ((5, -1), (5, 1)), ((1, 1), (6, 1)), ((0, 1), (7, 1)), ((3, 1), (4, -1))

sage: syms = ManinSymbolList_gamma0(7, 2); syms
Manin Symbol List of weight 2 for Gamma0(7)
sage: modS_relations(syms)
set([((3, 1), (4, 1)), ((2, 1), (7, 1)), ((5, 1), (6, 1)), ((0, 1), (1, 1))])
```

Next we do an example with Gamma1:

```
sage: from sage.modular.modsym.manin_symbols import ManinSymbolList_gamma1
sage: syms = ManinSymbolList_gamma1(3,2); syms
Manin Symbol List of weight 2 for Gamma1(3)
sage: modS_relations(syms)
set([((3, 1), (6, 1)), ((0, 1), (5, 1)), ((0, 1), (2, 1)), ((3, 1), (4, 1)), ((6, 1), (7, 1)), (
```

relation_matrix_wtk_g0 (*syms, sign, field, sparse*)

Compute the matrix of relations. Despite the name, this is used for all spaces (not just for Gamma0). For a description of the algorithm, see the docstring for `compute_presentation`.

INPUT:

- `syms`: sage.modular.modsym.manin_symbols.ManinSymbolList object
- `sign`: integer (0, 1 or -1)
- `field`: the base field (non-field base rings not supported at present)
- `sparse`: (True or False) whether to use sparse arithmetic.

Note that ManinSymbolList objects already have a specific weight, so there is no need for an extra `weight` parameter.

OUTPUT: a pair (R, mod) where

- R is a matrix as output by `T_relation_matrix_wtk_g0`
- mod is a set of 2-term relations as output by `sparse_2term_quotient`

EXAMPLE:

```
sage: L =  sage.modular.modsym.manin_symbols.ManinSymbolList_gamma0(8,2)
sage: A = sage.modular.modsym.relation_matrix.relation_matrix_wtk_g0(L, 0, GF(2), True); A
([0 0 0 0 0 0 0 0 1 0 0 0]
[0 0 0 0 0 0 0 0 1 1 1 0]
[0 0 0 0 0 0 1 0 0 1 1 0]
[0 0 0 0 0 0 1 0 0 0 0 0],
 [(1, 1),
  (1, 1),
  (8, 1),
  (10, 1),
  (6, 1),
  (11, 1),
  (6, 1),
  (9, 1),
  (8, 1),
  (9, 1),
  (10, 1),
  (11, 1)])
sage: A[0].is_sparse()
True
```

sparse_2term_quotient (*rels, n, F*)

Performs Sparse Gauss elimination on a matrix all of whose columns have at most 2 nonzero entries. We use an obvious algorithm, whichs runs fast enough. (Typically making the list of relations takes more time than computing this quotient.) This algorithm is more subtle than just "identify symbols in pairs", since complicated relations can cause generators to surprisingly equal 0.

INPUT:

- `rels` - set of pairs ((i,s), (j,t)). The pair represents the relation s*x_i + t*x_j = 0, where the i, j must be Python int's.
- `n` - int, the x_i are x_0, ..., x_n-1.
- `F` - base field

OUTPUT:

- `mod` - list such that mod[i] = (j,s), which means that x_i is equivalent to s*x_j, where the x_j are a basis for the quotient.

EXAMPLE: We quotient out by the relations

$$3 * x0 - x1 = 0, \qquad x1 + x3 = 0, \qquad x2 + x3 = 0, \qquad x4 - x5 = 0$$

to get

```
sage: v = [((int(0),3), (int(1),-1)), ((int(1),1), (int(3),1)), ((int(2),1),(int(3),1)), ((int(4
sage: rels = set(v)
sage: n = 6
sage: from sage.modular.modsym.relation_matrix import sparse_2term_quotient
sage: sparse_2term_quotient(rels, n, QQ)
[(3, -1/3), (3, -1), (3, -1), (3, 1), (5, 1), (5, 1)]
```

CHAPTER

FORTYFOUR

MODULAR FORMS

44.1 Creating Spaces of Modular Forms

EXAMPLES:

```
sage: m = ModularForms(Gamma1(4),11)
sage: m
Modular Forms space of dimension 6 for Congruence Subgroup Gamma1(4) of weight 11 over Rational Field
sage: m.basis()
[
q - 134*q^5 + O(q^6),
q^2 + 80*q^5 + O(q^6),
q^3 + 16*q^5 + O(q^6),
q^4 - 4*q^5 + O(q^6),
1 + 4092/50521*q^2 + 472384/50521*q^3 + 4194300/50521*q^4 + O(q^6),
q + 1024*q^2 + 59048*q^3 + 1048576*q^4 + 9765626*q^5 + O(q^6)
]
```

CuspForms(*group=1, weight=2, base_ring=None, use_cache=True, prec=6*)

Create a space of cuspidal modular forms.

See the documentation for the ModularForms command for a description of the input parameters.

EXAMPLES:

```
sage: CuspForms(11,2)
Cuspidal subspace of dimension 1 of Modular Forms space of dimension 2 for Congruence Subgroup G
```

EisensteinForms(*group=1, weight=2, base_ring=None, use_cache=True, prec=6*)

Create a space of eisenstein modular forms.

See the documentation for the ModularForms command for a description of the input parameters.

EXAMPLES:

```
sage: EisensteinForms(11,2)
Eisenstein subspace of dimension 1 of Modular Forms space of dimension 2 for Congruence Subgroup
```

ModularForms(*group=1, weight=2, base_ring=None, use_cache=True, prec=6*)

Create an ambient space of modular forms.

INPUT:

- `group` - A congruence subgroup or a Dirichlet character eps.
- `weight` - int, the weight, which must be an integer = 1.

- `base_ring` - the base ring (ignored if group is a Dirichlet character)

Create using the command ModularForms(group, weight, base_ring) where group could be either a congruence subgroup or a Dirichlet character.

EXAMPLES: First we create some spaces with trivial character:

```
sage: ModularForms(Gamma0(11),2).dimension()
2
sage: ModularForms(Gamma0(1),12).dimension()
2
```

If we give an integer N for the congruence subgroup, it defaults to $\Gamma_0(N)$:

```
sage: ModularForms(1,12).dimension()
2
sage: ModularForms(11,4)
Modular Forms space of dimension 4 for Congruence Subgroup Gamma0(11) of weight 4 over Rational
```

We create some spaces for $\Gamma_1(N)$.

```
sage: ModularForms(Gamma1(13),2)
Modular Forms space of dimension 13 for Congruence Subgroup Gamma1(13) of weight 2 over Rational
sage: ModularForms(Gamma1(13),2).dimension()
13
sage: [ModularForms(Gamma1(7),k).dimension() for k in [2,3,4,5]]
[5, 7, 9, 11]
sage: ModularForms(Gamma1(5),11).dimension()
12
```

We create a space with character:

```
sage: e = (DirichletGroup(13).0)^2
sage: e.order()
6
sage: M = ModularForms(e, 2); M
Modular Forms space of dimension 3, character [zeta6] and weight 2 over Cyclotomic Field of orde
sage: f = M.T(2).charpoly('x'); f
x^3 + (-2*zeta6 - 2)*x^2 - 2*zeta6*x + 14*zeta6 - 7
sage: f.factor()
(x - 2*zeta6 - 1) * (x - zeta6 - 2) * (x + zeta6 + 1)
```

More examples of spaces with character:

```
sage: e = DirichletGroup(5, RationalField()).gen(); e
[-1]
sage: m = ModularForms(e, 2); m
Modular Forms space of dimension 2, character [-1] and weight 2 over Rational Field
sage: m == loads(dumps(m))
True
sage: m.T(2).charpoly('x')
x^2 - 1
sage: m = ModularForms(e, 6); m.dimension()
4
sage: m.T(2).charpoly('x')
x^4 - 917*x^2 - 42284
```

This came up in a subtle bug (trac #5923):

```
sage: ModularForms(gp(1), gap(12))
Modular Forms space of dimension 2 for Modular Group SL(2,Z) of weight 12 over Rational Field
```

ModularForms_clear_cache()

Clear the cache of modular forms.

EXAMPLES:

```
sage: M = ModularForms(37,2)
sage: sage.modular.modform.constructor._cache == {}
False

sage: sage.modular.modform.constructor.ModularForms_clear_cache()
sage: sage.modular.modform.constructor._cache
{}
```

Newform(*identifier, group=None, weight=2, base_ring=Rational Field, names=None*)

INPUT:

- `identifier` - a canonical label, or the index of the specific newform desired
- `group` - the congruence subgroup of the newform
- `weight` - the weight of the newform (default 2)
- `base_ring` - the base ring
- `names` - if the newform has coefficients in a number field, a generator name must be specified

EXAMPLES:

```
sage: Newform('67a', names='a')
q + 2*q^2 - 2*q^3 + 2*q^4 + 2*q^5 + O(q^6)
sage: Newform('67b', names='a')
q + a1*q^2 + (-a1 - 3)*q^3 + (-3*a1 - 3)*q^4 - 3*q^5 + O(q^6)
```

Newforms(*group, weight=2, base_ring=Rational Field, names=None*)

INPUT:

- `group` - the congruence subgroup of the newform
- `weight` - the weight of the newform (default 2)
- `base_ring` - the base ring
- `names` - if the newform has coefficients in a number field, a generator name must be specified

EXAMPLES:

```
sage: Newforms(11, 2)
[q - 2*q^2 - q^3 + 2*q^4 + q^5 + O(q^6)]
sage: Newforms(65, names='a')
[q - q^2 - 2*q^3 - q^4 - q^5 + O(q^6),
 q + a1*q^2 + (a1 + 1)*q^3 + (-2*a1 - 1)*q^4 + q^5 + O(q^6),
 q + a2*q^2 + (-a2 + 1)*q^3 + q^4 - q^5 + O(q^6)]
```

canonical_parameters(*group, level, weight, base_ring*)

Given a group, level, weight, and base_ring as input by the user, return a canonicalized version of them, where level is a Sage integer, group really is a group, weight is a Sage integer, and base_ring a Sage ring. Note that we can't just get the level from the group, because we have the convention that the character for Gamma1(N) is None (which makes good sense).

INPUT:

- `group` - int, long, Sage integer, group, dirichlet character, or
- `level` - int, long, Sage integer, or group
- `weight` - coercible to Sage integer
- `base_ring` - commutative Sage ring

OUTPUT:

- `level` - Sage integer
- `group` - congruence subgroup
- `weight` - Sage integer
- `ring` - commutative Sage ring

EXAMPLES:

```
sage: from sage.modular.modform.constructor import canonical_parameters
sage: v = canonical_parameters(5, 5, int(7), ZZ); v
(5, Congruence Subgroup Gamma0(5), 7, Integer Ring)
sage: type(v[0]), type(v[1]), type(v[2]), type(v[3])
(<type 'sage.rings.integer.Integer'>,
 <class 'sage.modular.arithgroup.congroup_gamma0.Gamma0_class'>,
 <type 'sage.rings.integer.Integer'>,
 <type 'sage.rings.integer_ring.IntegerRing_class'>)
sage: canonical_parameters( 5, 7, 7, ZZ )
...
ValueError: group and level do not match.
```

parse_label(*s*)

Given a string s corresponding to a newform label, return the corresponding group and index.

EXAMPLES:

```
sage: sage.modular.modform.constructor.parse_label('11a')
(Congruence Subgroup Gamma0(11), 0)
sage: sage.modular.modform.constructor.parse_label('11aG1')
(Congruence Subgroup Gamma1(11), 0)
sage: sage.modular.modform.constructor.parse_label('11wG1')
(Congruence Subgroup Gamma1(11), 22)
```

44.2 Generic spaces of modular forms

EXAMPLES (computation of base ring): Return the base ring of this space of modular forms.

EXAMPLES: For spaces of modular forms for $\Gamma_0(N)$ or $\Gamma_1(N)$, the default base ring is $\mathbf{Q}$:

```
sage: ModularForms(11,2).base_ring()
Rational Field
sage: ModularForms(1,12).base_ring()
Rational Field
sage: CuspForms(Gamma1(13),3).base_ring()
Rational Field
```

The base ring can be explicitly specified in the constructor function.

```
sage: ModularForms(11,2,base_ring=GF(13)).base_ring()
Finite Field of size 13
```

For modular forms with character the default base ring is the field generated by the image of the character.

```
sage: ModularForms(DirichletGroup(13).0,3).base_ring()
Cyclotomic Field of order 12 and degree 4
```

For example, if the character is quadratic then the field is $\mathbf{Q}$ (if the characteristic is 0).

```
sage: ModularForms(DirichletGroup(13).0^6,3).base_ring()
Rational Field
```

An example in characteristic 7:

```
sage: ModularForms(13,3,base_ring=GF(7)).base_ring()
Finite Field of size 7
```

class ModularFormsSpace(*group, weight, character, base_ring*)

A generic space of modular forms.

base_extend(*base_ring*)

Return the base extension of self to base_ring.

EXAMPLES:

```
sage: M = ModularForms(11,2) ; M
Modular Forms space of dimension 2 for Congruence Subgroup Gamma0(11) of weight 2 over Ratio
sage: M.base_extend(CyclotomicField(5))
Modular Forms space of dimension 2 for Congruence Subgroup Gamma0(11) of weight 2 over Cyclo
```

basis()

Return a basis for self.

EXAMPLES:

```
sage: MM = ModularForms(11,2)
sage: MM.basis()
[
q - 2*q^2 - q^3 + 2*q^4 + q^5 + O(q^6),
1 + 12/5*q + 36/5*q^2 + 48/5*q^3 + 84/5*q^4 + 72/5*q^5 + O(q^6)
]
```

change_ring(*R*)

Change the base ring of this space of modular forms.

TODO: Write this function.

EXAMPLES:

```
sage: sage.modular.modform.space.ModularFormsSpace(Gamma0(11),2,DirichletGroup(1).0,QQ).char
...
NotImplementedError: This function has not yet been implemented.
```

character()

Return the Dirichlet character of this space.

EXAMPLES:

```
sage: M = ModularForms(DirichletGroup(11).0, 3)
sage: M.character()
[zeta10]
sage: s = M.cuspidal_submodule()
sage: s.character()
[zeta10]
```

```
sage: CuspForms(DirichletGroup(11).0,3).character()
[zeta10]
```

cuspidal_submodule()

Return the cuspidal submodule of self.

EXAMPLES:

```
sage: N = ModularForms(6,4) ; N
Modular Forms space of dimension 5 for Congruence Subgroup Gamma0(6) of weight 4 over Ration
sage: N.eisenstein_subspace().dimension()
4

sage: N.cuspidal_submodule()
Cuspidal subspace of dimension 1 of Modular Forms space of dimension 5 for Congruence Subgro

sage: N.cuspidal_submodule().dimension()
1
```

cuspidal_subspace()

Synonym for cuspidal_submodule.

EXAMPLES:

```
sage: N = ModularForms(6,4) ; N
Modular Forms space of dimension 5 for Congruence Subgroup Gamma0(6) of weight 4 over Ration
sage: N.eisenstein_subspace().dimension()
4

sage: N.cuspidal_subspace()
Cuspidal subspace of dimension 1 of Modular Forms space of dimension 5 for Congruence Subgro

sage: N.cuspidal_submodule().dimension()
1
```

decomposition()

This function returns a list of submodules $V(f_i, t)$ corresponding to newforms f_i of some level dividing the level of self, such that the direct sum of the submodules equals self, if possible. The space $V(f_i, t)$ is the image under $g(q)$ maps to $g(q^t)$ of the intersection with $R[[q]]$ of the space spanned by the conjugates of f_i, where R is the base ring of self.

TODO: Implement this function.

EXAMPLES:

```
sage: M = ModularForms(11,2); M.decomposition()
...
NotImplementedError
```

echelon_basis()

Return a basis for self in reduced echelon form. This means that if we view the q-expansions of the basis as defining rows of a matrix (with infinitely many columns), then this matrix is in reduced echelon form.

EXAMPLES:

```
sage: M = ModularForms(Gamma0(11),4)
sage: M.echelon_basis()
[
1 + O(q^6),
q - 9*q^4 - 10*q^5 + O(q^6),
q^2 + 6*q^4 + 12*q^5 + O(q^6),
q^3 + q^4 + q^5 + O(q^6)
]
```

```
sage: M.cuspidal_subspace().echelon_basis()
[
q + 3*q^3 - 6*q^4 - 7*q^5 + O(q^6),
q^2 - 4*q^3 + 2*q^4 + 8*q^5 + O(q^6)
]

sage: M = ModularForms(SL2Z, 12)
sage: M.echelon_basis()
[
1 + 196560*q^2 + 16773120*q^3 + 398034000*q^4 + 4629381120*q^5 + O(q^6),
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 + O(q^6)
]

sage: M = CuspForms(Gamma0(17),4, prec=10)
sage: M.echelon_basis()
[
q + 2*q^5 - 8*q^7 - 8*q^8 + 7*q^9 + O(q^10),
q^2 - 3/2*q^5 - 7/2*q^6 + 9/2*q^7 + q^8 - 4*q^9 + O(q^10),
q^3 - 2*q^6 + q^7 - 4*q^8 - 2*q^9 + O(q^10),
q^4 - 1/2*q^5 - 5/2*q^6 + 3/2*q^7 + 2*q^9 + O(q^10)
]
```

echelon_form()

Return a space of modular forms isomorphic to self but with basis of q-expansions in reduced echelon form.

This is useful, e.g., the default basis for spaces of modular forms is rarely in echelon form, but echelon form is useful for quickly recognizing whether a q-expansion is in the space.

EXAMPLES: We first illustrate two ambient spaces and their echelon forms.

```
sage: M = ModularForms(11)
sage: M.basis()
[
q - 2*q^2 - q^3 + 2*q^4 + q^5 + O(q^6),
1 + 12/5*q + 36/5*q^2 + 48/5*q^3 + 84/5*q^4 + 72/5*q^5 + O(q^6)
]
sage: M.echelon_form().basis()
[
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + O(q^6),
q - 2*q^2 - q^3 + 2*q^4 + q^5 + O(q^6)
]

sage: M = ModularForms(Gamma1(6),4)
sage: M.basis()
[
q - 2*q^2 - 3*q^3 + 4*q^4 + 6*q^5 + O(q^6),
1 + O(q^6),
q - 8*q^4 + 126*q^5 + O(q^6),
q^2 + 9*q^4 + O(q^6),
q^3 + O(q^6)
]
sage: M.echelon_form().basis()
[
1 + O(q^6),
q + 94*q^5 + O(q^6),
q^2 + 36*q^5 + O(q^6),
q^3 + O(q^6),
q^4 - 4*q^5 + O(q^6)
]
```

We create a space with a funny basis then compute the corresponding echelon form.

```
sage: M = ModularForms(11,4)
sage: M.basis()
[
q + 3*q^3 - 6*q^4 - 7*q^5 + O(q^6),
q^2 - 4*q^3 + 2*q^4 + 8*q^5 + O(q^6),
1 + O(q^6),
q + 9*q^2 + 28*q^3 + 73*q^4 + 126*q^5 + O(q^6)
]
sage: F = M.span_of_basis([M.0 + 1/3*M.1, M.2 + M.3]); F.basis()
[
q + 1/3*q^2 + 5/3*q^3 - 16/3*q^4 - 13/3*q^5 + O(q^6),
1 + q + 9*q^2 + 28*q^3 + 73*q^4 + 126*q^5 + O(q^6)
]
sage: E = F.echelon_form(); E.basis()
[
1 + 26/3*q^2 + 79/3*q^3 + 235/3*q^4 + 391/3*q^5 + O(q^6),
q + 1/3*q^2 + 5/3*q^3 - 16/3*q^4 - 13/3*q^5 + O(q^6)
]
```

eisenstein_series()

Compute the Eisenstein series associated to this space.

Note: This function should be overridden by all derived classes.

EXAMPLES:

```
sage: M = sage.modular.modform.space.ModularFormsSpace(Gamma0(11),2,DirichletGroup(1).0,base
...
NotImplementedError: computation of Eisenstein series in this space not yet implemented
```

eisenstein_submodule()

Return the Eisenstein submodule for this space of modular forms.

EXAMPLES:

```
sage: M = ModularForms(11,2)
sage: M.eisenstein_submodule()
Eisenstein subspace of dimension 1 of Modular Forms space of dimension 2 for Congruence Subg
```

eisenstein_subspace()

Synonym for eisenstein_submodule.

EXAMPLES:

```
sage: M = ModularForms(11,2)
sage: M.eisenstein_subspace()
Eisenstein subspace of dimension 1 of Modular Forms space of dimension 2 for Congruence Subg
```

embedded_submodule()

Return the underlying module of self.

EXAMPLES:

```
sage: N = ModularForms(6,4)
sage: N.dimension()
5

sage: N.embedded_submodule()
Vector space of dimension 5 over Rational Field
```

find_in_space(*f, forms=None, prec=None, indep=True*)

INPUT:

- f - a modular form or power series
- forms - (default: None) a specific list of modular forms or q-expansions.
- prec - if forms are given, compute with them to the given precision
- indep - (default: True) whether the given list of forms are assumed to form a basis.

OUTPUT: A list of numbers that give f as a linear combination of the basis for this space or of the given forms if independent=True.

Note: If the list of forms is given, they do *not* have to be in self.

EXAMPLES:

```
sage: M = ModularForms(11,2)
sage: N = ModularForms(10,2)
sage: M.find_in_space( M.basis()[0] )
[1, 0]

sage: M.find_in_space( N.basis()[0], forms=N.basis() )
[1, 0, 0]

sage: M.find_in_space( N.basis()[0] )
...
ArithmeticError: vector is not in free module
```

gen(*n*)

Return the nth generator of self.

EXAMPLES:

```
sage: N = ModularForms(6,4)
sage: N.basis()
[
q - 2*q^2 - 3*q^3 + 4*q^4 + 6*q^5 + O(q^6),
1 + O(q^6),
q - 8*q^4 + 126*q^5 + O(q^6),
q^2 + 9*q^4 + O(q^6),
q^3 + O(q^6)
]

sage: N.gen(0)
q - 2*q^2 - 3*q^3 + 4*q^4 + 6*q^5 + O(q^6)

sage: N.gen(4)
q^3 + O(q^6)

sage: N.gen(5)
...
ValueError: Generator 5 not defined
```

gens()

Return a complete set of generators for self.

EXAMPLES:

```
sage: N = ModularForms(6,4)
sage: N.gens()
[
q - 2*q^2 - 3*q^3 + 4*q^4 + 6*q^5 + O(q^6),
1 + O(q^6),
q - 8*q^4 + 126*q^5 + O(q^6),
q^2 + 9*q^4 + O(q^6),
q^3 + O(q^6)
]
```

group()

Return the congruence subgroup associated to this space of modular forms.

EXAMPLES:

```
sage: ModularForms(Gamma0(12),4).group()
Congruence Subgroup Gamma0(12)

sage: CuspForms(Gamma1(113),2).group()
Congruence Subgroup Gamma1(113)
```

Note that $\Gamma_1(1)$ and $\Gamma_0(1)$ are replaced by $\mathrm{SL}_2(\mathbf{Z})$.

```
sage: CuspForms(Gamma1(1),12).group()
Modular Group SL(2,Z)
sage: CuspForms(SL2Z,12).group()
Modular Group SL(2,Z)
```

has_character()

Return True if this space of modular forms has a specific character.

This is True exactly when the character() function does not return None.

EXAMPLES: A space for $\Gamma_0(N)$ has trivial character, hence has a character.

```
sage: CuspForms(Gamma0(11),2).has_character()
True
```

A space for $\Gamma_1(N)$ (for $N \geq 2$) never has a specific character.

```
sage: CuspForms(Gamma1(11),2).has_character()
False
sage: CuspForms(DirichletGroup(11).0,3).has_character()
True
```

has_coerce_map_from_impl(*from_par*)

Code to make ModularFormsSpace work well with coercion framework.

EXAMPLES:

```
sage: M = ModularForms(22,2)
sage: M.has_coerce_map_from_impl(M.cuspidal_subspace())
True
sage: M.has_coerce_map_from(ModularForms(22,4))
False
```

integral_basis()

Return an integral basis for this space of modular forms.

EXAMPLES: In this example the integral and echelon bases are different.

```
sage: m = ModularForms(97,2,prec=10)
sage: s = m.cuspidal_subspace()
sage: s.integral_basis()
[
q + 2*q^7 + 4*q^8 - 2*q^9 + O(q^10),
q^2 + q^4 + q^7 + 3*q^8 - 3*q^9 + O(q^10),
q^3 + q^4 - 3*q^8 + q^9 + O(q^10),
2*q^4 - 2*q^8 + O(q^10),
q^5 - 2*q^8 + 2*q^9 + O(q^10),
q^6 + 2*q^7 + 5*q^8 - 5*q^9 + O(q^10),
3*q^7 + 6*q^8 - 4*q^9 + O(q^10)
]
sage: s.echelon_basis()
[
```

```
q + 2/3*q^9 + O(q^10),
q^2 + 2*q^8 - 5/3*q^9 + O(q^10),
q^3 - 2*q^8 + q^9 + O(q^10),
q^4 - q^8 + O(q^10),
q^5 - 2*q^8 + 2*q^9 + O(q^10),
q^6 + q^8 - 7/3*q^9 + O(q^10),
q^7 + 2*q^8 - 4/3*q^9 + O(q^10)
]
```

Here's another example where there is a big gap in the valuations:

```
sage: m = CuspForms(64,2)
sage: m.integral_basis()
[
q + O(q^6),
q^2 + O(q^6),
q^5 + O(q^6)
]
```

TESTS:

```
sage: m = CuspForms(11*2^4,2, prec=13); m
Cuspidal subspace of dimension 19 of Modular Forms space of dimension 30 for Congruence Subg
sage: m.integral_basis()          # takes a long time (3 or 4 seconds)
[
q + O(q^13),
q^2 + O(q^13),
q^3 + O(q^13),
q^4 + O(q^13),
q^5 + O(q^13),
q^6 + O(q^13),
q^7 + O(q^13),
q^8 + O(q^13),
q^9 + O(q^13),
q^10 + O(q^13),
q^11 + O(q^13),
q^12 + O(q^13),
O(q^13),
O(q^13),
O(q^13),
O(q^13),
O(q^13),
O(q^13),
O(q^13)
]
```

is_ambient()

Return True if this an ambient space of modular forms.

EXAMPLES:

```
sage: M = ModularForms(Gamma1(4),4)
sage: M.is_ambient()
True

sage: E = M.eisenstein_subspace()
sage: E.is_ambient()
False
```

level()

Return the level of self.

EXAMPLES:

```
sage: M = ModularForms(47,3)
sage: M.level()
47
```

modular_symbols(*sign=0*)

Return the space of modular symbols corresponding to self with the given sign.

EXAMPLES:

```
sage: M = sage.modular.modform.space.ModularFormsSpace(Gamma0(11),2,DirichletGroup(1).0,base
...
NotImplementedError: computation of associated modular symbols space not yet implemented
```

new_submodule(*p=None*)

Return the new submodule of self. If p is specified, return the p-new submodule of self.

Note: This function should be overridden by all derived classes.

EXAMPLES:

```
sage: M = sage.modular.modform.space.ModularFormsSpace(Gamma0(11),2,DirichletGroup(1).0,base
...
NotImplementedError: computation of new submodule not yet implemented
```

new_subspace(*p=None*)

Synonym for new_submodule.

EXAMPLES:

```
sage: M = sage.modular.modform.space.ModularFormsSpace(Gamma0(11),2,DirichletGroup(1).0,base
...
NotImplementedError: computation of new submodule not yet implemented
```

newforms(*names=None*)

Return all newforms in the cuspidal subspace of self.

EXAMPLES:

```
sage: CuspForms(18,4).newforms()
[q + 2*q^2 + 4*q^4 - 6*q^5 + O(q^6)]
sage: CuspForms(32,4).newforms()
[q - 8*q^3 - 10*q^5 + O(q^6), q + 22*q^5 + O(q^6), q + 8*q^3 - 10*q^5 + O(q^6)]
sage: CuspForms(23).newforms('b')
[q + b0*q^2 + (-2*b0 - 1)*q^3 + (-b0 - 1)*q^4 + 2*b0*q^5 + O(q^6)]
sage: CuspForms(23).newforms()
...
ValueError: Please specify a name to be used when generating names for generators of Hecke e
```

prec(*new_prec=None*)

Return or set the default precision used for displaying q-expansions of elements of this space.

INPUT:

- `new_prec` - positive integer (default: None)

OUTPUT: if new_prec is None, returns the current precision.

EXAMPLES:

```
sage: M = ModularForms(1,12)
sage: S = M.cuspidal_subspace()
sage: S.prec()
6
sage: S.basis()
[
```

```
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 + O(q^6)
]
sage: S.prec(8)
8
sage: S.basis()
[
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7 + O(q^8)
]
```

q_echelon_basis(*prec=None*)

Return the echelon form of the basis of q-expansions of self up to precision prec.

The q-expansions are power series (not actual modular forms). The number of q-expansions returned equals the dimension.

EXAMPLES:

```
sage: M = ModularForms(11,2)
sage: M.q_expansion_basis()
[
q - 2*q^2 - q^3 + 2*q^4 + q^5 + O(q^6),
1 + 12/5*q + 36/5*q^2 + 48/5*q^3 + 84/5*q^4 + 72/5*q^5 + O(q^6)
]

sage: M.q_echelon_basis()
[
1 + 12*q^2 + 12*q^3 + 12*q^4 + 12*q^5 + O(q^6),
q - 2*q^2 - q^3 + 2*q^4 + q^5 + O(q^6)
]
```

q_expansion_basis(*prec=None*)

Return a sequence of q-expansions for the basis of this space computed to the given input precision.

INPUT:

- `prec` - integer (=0) or None

If prec is None, the prec is computed to be *at least* large enough so that each q-expansion determines the form as an element of this space.

Note: In fact, the q-expansion basis is always computed to *at least* `self.prec()`.

EXAMPLES:

```
sage: S = ModularForms(11,2).cuspidal_submodule()
sage: S.q_expansion_basis()
[
q - 2*q^2 - q^3 + 2*q^4 + q^5 + O(q^6)
]
sage: S.q_expansion_basis(5)
[
q - 2*q^2 - q^3 + 2*q^4 + O(q^5)
]
sage: S = ModularForms(1,24).cuspidal_submodule()
sage: S.q_expansion_basis(8)
[
q + 195660*q^3 + 12080128*q^4 + 44656110*q^5 - 982499328*q^6 - 147247240*q^7 + O(q^8),
q^2 - 48*q^3 + 1080*q^4 - 15040*q^5 + 143820*q^6 - 985824*q^7 + O(q^8)
]
```

q_integral_basis(*prec=None*)

Return a $\mathbf{Z}$-reduced echelon basis of q-expansions for self.

The q-expansions are power series with coefficients in $\mathbf{Z}$; they are *not* actual modular forms.

The base ring of self must be $\mathbf{Q}$. The number of q-expansions returned equals the dimension.

EXAMPLES:

```
sage: S = CuspForms(11,2)
sage: S.q_integral_basis(5)
[
q - 2*q^2 - q^3 + 2*q^4 + O(q^5)
]
```

set_precision(*new_prec*)

Set the default precision used for displaying q-expansions.

INPUT:

- new_prec - positive integer

EXAMPLES:

```
sage: M = ModularForms(Gamma0(37),2)
sage: M.set_precision(10)
sage: S = M.cuspidal_subspace()
sage: S.basis()
[
q + q^3 - 2*q^4 - q^7 - 2*q^9 + O(q^10),
q^2 + 2*q^3 - 2*q^4 + q^5 - 3*q^6 - 4*q^9 + O(q^10)
]

sage: S.set_precision(0)
sage: S.basis()
[
O(q^0),
O(q^0)
]
```

The precision of subspaces is the same as the precision of the ambient space.

```
sage: S.set_precision(2)
sage: M.basis()
[
q + O(q^2),
O(q^2),
1 + 2/3*q + O(q^2)
]
```

The precision must be nonnegative:

```
sage: S.set_precision(-1)
...
ValueError: n (=-1) must be >= 0
```

We do another example with nontrivial character.

```
sage: M = ModularForms(DirichletGroup(13).0^2)
sage: M.set_precision(10)
sage: M.cuspidal_subspace().0
q + (-zeta6 - 1)*q^2 + (2*zeta6 - 2)*q^3 + zeta6*q^4 + (-2*zeta6 + 1)*q^5 + (-2*zeta6 + 4)*q
```

span(*B*)

Take a set B of forms, and return the subspace of self with B as a basis.

EXAMPLES:

```
sage: N = ModularForms(6,4) ; N
Modular Forms space of dimension 5 for Congruence Subgroup Gamma0(6) of weight 4 over Ration
```

```
sage: N.span_of_basis([N.basis()[0]])
Modular Forms subspace of dimension 1 of Modular Forms space of dimension 5 for Congruence S

sage: N.span_of_basis([N.basis()[0], N.basis()[1]])
Modular Forms subspace of dimension 2 of Modular Forms space of dimension 5 for Congruence S

sage: N.span_of_basis( N.basis() )
Modular Forms subspace of dimension 5 of Modular Forms space of dimension 5 for Congruence S
```

span_of_basis(*B*)

Take a set B of forms, and return the subspace of self with B as a basis.

EXAMPLES:

```
sage: N = ModularForms(6,4) ; N
Modular Forms space of dimension 5 for Congruence Subgroup Gamma0(6) of weight 4 over Ration

sage: N.span_of_basis([N.basis()[0]])
Modular Forms subspace of dimension 1 of Modular Forms space of dimension 5 for Congruence S

sage: N.span_of_basis([N.basis()[0], N.basis()[1]])
Modular Forms subspace of dimension 2 of Modular Forms space of dimension 5 for Congruence S

sage: N.span_of_basis( N.basis() )
Modular Forms subspace of dimension 5 of Modular Forms space of dimension 5 for Congruence S
```

sturm_bound(*M=None*)

For a space M of modular forms, this function returns an integer B such that two modular forms in either self or M are equal if and only if their q-expansions are equal to precision B (note that this is 1+ the usual Sturm bound, since $O(q^{\text{prec}})$ has precision prec). If M is none, then M is set equal to self.

EXAMPLES:

```
sage: S37=CuspForms(37,2)
sage: S37.sturm_bound()
8
sage: M = ModularForms(11,2)
sage: M.sturm_bound()
3
sage: ModularForms(Gamma1(15),2).sturm_bound()
33
```

Note: Reference for the Sturm bound that we use in the definition of of this function:

J. Sturm, On the congruence of modular forms, Number theory (New York, 1984-1985), Springer, Berlin, 1987, pp. 275-280.

Useful Remark:

Kevin Buzzard pointed out to me (William Stein) in Fall 2002 that the above bound is fine for Gamma1 with character, as one sees by taking a power of f. More precisely, if $f \cong 0 \pmod{p}$ for first s coefficients, then $f^r = 0 \pmod{p}$ for first sr coefficents. Since the weight of f^r is $r$weight(f), it follows that if $s \geq$ the sturm bound for Γ_0 at weight(f), then f^r has valuation large enough to be forced to be 0 at $r\cdot$ weight(f) by Sturm bound (which is valid if we choose r right). Thus $f \cong 0 \pmod{p}$. Conclusion: For Γ_1 with fixed character, the Sturm bound is *exactly* the same as for Γ_0. A key point is that we are finding $\mathbf{Z}[\varepsilon]$ generators for the Hecke algebra here, not $\mathbf{Z}$-generators. So if one wants generators for the Hecke algebra over $\mathbf{Z}$, this bound is wrong.

This bound works over any base, even a finite field. There might be much better bounds over $\mathbf{Q}$, or for comparing two eigenforms.

weight()

Return the weight of this space of modular forms.

EXAMPLES:

```
sage: M = ModularForms(Gamma1(13),11)
sage: M.weight()
11

sage: M = ModularForms(Gamma0(997),100)
sage: M.weight()
100

sage: M = ModularForms(Gamma0(97),4)
sage: M.weight()
4
sage: M.eisenstein_submodule().weight()
4
```

contains_each(*V, B*)

Determine whether or not V contains every element of B. Used here for linear algebra, but works very generally.

EXAMPLES:

```
sage: contains_each = sage.modular.modform.space.contains_each
sage: contains_each( range(20), prime_range(20) )
True
sage: contains_each( range(20), range(30) )
False
```

is_ModularFormsSpace(*x*)

Return True if x is a `ModularFormsSpace`.

EXAMPLES:

```
sage: from sage.modular.modform.space import is_ModularFormsSpace
sage: is_ModularFormsSpace(ModularForms(11,2))
True
sage: is_ModularFormsSpace(CuspForms(11,2))
True
sage: is_ModularFormsSpace(3)
False
```

44.3 Ambient Spaces of Modular Forms.

EXAMPLES:

We compute a basis for the ambient space $M_2(\Gamma_1(25), \chi)$, where χ is quadratic.

```
sage: chi = DirichletGroup(25,QQ).0; chi
[-1]
sage: n = ModularForms(chi,2); n
Modular Forms space of dimension 6, character [-1] and weight 2 over Rational Field
sage: type(n)
<class 'sage.modular.modform.ambient_eps.ModularFormsAmbient_eps'>
```

Compute a basis:

```
sage: n.basis()
[
```

```
1 + O(q^6),
q + O(q^6),
q^2 + O(q^6),
q^3 + O(q^6),
q^4 + O(q^6),
q^5 + O(q^6)
]
```

Compute the same basis but to higher precision:

```
sage: n.set_precision(20)
sage: n.basis()
[
1 + 10*q^10 + 20*q^15 + O(q^20),
q + 5*q^6 + q^9 + 12*q^11 - 3*q^14 + 17*q^16 + 8*q^19 + O(q^20),
q^2 + 4*q^7 - q^8 + 8*q^12 + 2*q^13 + 10*q^17 - 5*q^18 + O(q^20),
q^3 + q^7 + 3*q^8 - q^12 + 5*q^13 + 3*q^17 + 6*q^18 + O(q^20),
q^4 - q^6 + 2*q^9 + 3*q^14 - 2*q^16 + 4*q^19 + O(q^20),
q^5 + q^10 + 2*q^15 + O(q^20)
]
```

TESTS:

```
sage: m = ModularForms(Gamma1(20),2,GF(7))
sage: loads(dumps(m)) == m
True
```

```
sage: m = ModularForms(GammaH(11,[2]), 2); m
Modular Forms space of dimension 2 for Congruence Subgroup Gamma_H(11) with H generated by [2] of wei
sage: type(m)
<class 'sage.modular.modform.ambient.ModularFormsAmbient'>
```

class ModularFormsAmbient(*group, weight, base_ring, character=None*)

An ambient space of modular forms.

ambient_space()

Return the ambient space that contains this ambient space. This is, of course, just this space again.

EXAMPLES:

```
sage: m = ModularForms(Gamma0(3),30)
sage: m.ambient_space() is m
True
```

change_ring(*base_ring*)

Change the base ring of this space of modular forms.

INPUT:

- R - ring

EXAMPLES:

```
sage: M = ModularForms(Gamma0(37),2)
sage: M.basis()
[
q + q^3 - 2*q^4 + O(q^6),
q^2 + 2*q^3 - 2*q^4 + q^5 + O(q^6),
1 + 2/3*q + 2*q^2 + 8/3*q^3 + 14/3*q^4 + 4*q^5 + O(q^6)
]
```

The basis after changing the base ring is the reduction modulo 3 of an integral basis.

```
sage: M3 = M.change_ring(GF(3))
sage: M3.basis()
[
1 + q^3 + q^4 + 2*q^5 + O(q^6),
q + q^3 + q^4 + O(q^6),
q^2 + 2*q^3 + q^4 + q^5 + O(q^6)
]
```

cuspidal_submodule()

Return the cuspidal submodule of this ambient module.

EXAMPLES:

```
sage: ModularForms(Gamma1(13)).cuspidal_submodule()
Cuspidal subspace of dimension 2 of Modular Forms space of dimension 13 for
Congruence Subgroup Gamma1(13) of weight 2 over Rational Field
```

dimension()

Return the dimension of this ambient space of modular forms, computed using a dimension formula (so it should be reasonably fast).

EXAMPLES:

```
sage: m = ModularForms(Gamma1(20),20)
sage: m.dimension()
238
```

eisenstein_params()

Return parameters that define all Eisenstein series in self.

OUTPUT: an immutable Sequence

EXAMPLES:

```
sage: m = ModularForms(Gamma0(22), 2)
sage: v = m.eisenstein_params(); v
[([1], [1], 2), ([1], [1], 11), ([1], [1], 22)]
sage: type(v)
<class 'sage.structure.sequence.Sequence'>
```

eisenstein_series()

Return all Eisenstein series associated to this space.

```
sage: ModularForms(27,2).eisenstein_series()
[
q^3 + O(q^6),
q - 3*q^2 + 7*q^4 - 6*q^5 + O(q^6),
1/12 + q + 3*q^2 + q^3 + 7*q^4 + 6*q^5 + O(q^6),
1/3 + q + 3*q^2 + 4*q^3 + 7*q^4 + 6*q^5 + O(q^6),
13/12 + q + 3*q^2 + 4*q^3 + 7*q^4 + 6*q^5 + O(q^6)
]
```

```
sage: ModularForms(Gamma1(5),3).eisenstein_series()
[
-1/5*zeta4 - 2/5 + q + (4*zeta4 + 1)*q^2 + (-9*zeta4 + 1)*q^3 + (4*zeta4 - 15)*q^4 + q^5 + O
q + (zeta4 + 4)*q^2 + (-zeta4 + 9)*q^3 + (4*zeta4 + 15)*q^4 + 25*q^5 + O(q^6),
1/5*zeta4 - 2/5 + q + (-4*zeta4 + 1)*q^2 + (9*zeta4 + 1)*q^3 + (-4*zeta4 - 15)*q^4 + q^5 + O
q + (-zeta4 + 4)*q^2 + (zeta4 + 9)*q^3 + (-4*zeta4 + 15)*q^4 + 25*q^5 + O(q^6)
]
```

```
sage: eps = DirichletGroup(13).0^2
sage: ModularForms(eps,2).eisenstein_series()
[
-7/13*zeta6 - 11/13 + q + (2*zeta6 + 1)*q^2 + (-3*zeta6 + 1)*q^3 + (6*zeta6 - 3)*q^4 - 4*q^5
q + (zeta6 + 2)*q^2 + (-zeta6 + 3)*q^3 + (3*zeta6 + 3)*q^4 + 4*q^5 + O(q^6)
]
```

eisenstein_submodule()

Return the Eisenstein submodule of this ambient module.

EXAMPLES:

```
sage: m = ModularForms(Gamma1(13),2); m
Modular Forms space of dimension 13 for Congruence Subgroup Gamma1(13) of weight 2 over Rati
sage: m.eisenstein_submodule()
Eisenstein subspace of dimension 11 of Modular Forms space of dimension 13 for Congruence Su
```

hecke_module_of_level(*N*)

Return the Hecke module of level N corresponding to self, which is the domain or codomain of a degeneracy map from self. Here N must be either a divisor or a multiple of the level of self.

EXAMPLES:

```
sage: ModularForms(25, 6).hecke_module_of_level(5)
Modular Forms space of dimension 3 for Congruence Subgroup Gamma0(5) of weight 6 over Ration
sage: ModularForms(Gamma1(4), 3).hecke_module_of_level(8)
Modular Forms space of dimension 7 for Congruence Subgroup Gamma1(8) of weight 3 over Ration
sage: ModularForms(Gamma1(4), 3).hecke_module_of_level(9)
...
ValueError: N (=9) must be a divisor or a multiple of the level of self (=4)
```

is_ambient()

Return True if this an ambient space of modular forms.

This is an ambient space, so this function always returns True.

EXAMPLES:

```
sage: ModularForms(11).is_ambient()
True
sage: CuspForms(11).is_ambient()
False
```

modular_symbols(*sign=0*)

Return the corresponding space of modular symbols with the given sign.

EXAMPLES:

```
sage: S = ModularForms(11,2)
sage: S.modular_symbols()
Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with sign 0 over Rational F
sage: S.modular_symbols(sign=1)
Modular Symbols space of dimension 2 for Gamma_0(11) of weight 2 with sign 1 over Rational F
sage: S.modular_symbols(sign=-1)
Modular Symbols space of dimension 1 for Gamma_0(11) of weight 2 with sign -1 over Rational

sage: ModularForms(1,12).modular_symbols()
Modular Symbols space of dimension 3 for Gamma_0(1) of weight 12 with sign 0 over Rational F
```

module()

Return the underlying free module corresponding to this space of modular forms. This is a free module (viewed as a tuple space) of the same dimension as this space over the same base ring.

EXAMPLES:

```
sage: m = ModularForms(Gamma1(13),10)
sage: m.free_module()
Vector space of dimension 69 over Rational Field
sage: ModularForms(Gamma1(13),4, GF(49,'b')).free_module()
Vector space of dimension 27 over Finite Field in b of size 7^2
```

new_submodule(*p=None*)

Return the new or p-new submodule of this ambient module.

INPUT:

- p - (default: None), if specified return only the p-new submodule.

EXAMPLES:

```
sage: m = ModularForms(Gamma0(33),2); m
Modular Forms space of dimension 6 for Congruence Subgroup Gamma0(33) of weight 2 over Ratio
sage: m.new_submodule()
Modular Forms subspace of dimension 1 of Modular Forms space of dimension 6 for Congruence S
```

Another example:

```
sage: M = ModularForms(17,4)
sage: N = M.new_subspace(); N
Modular Forms subspace of dimension 4 of Modular Forms space of dimension 6 for Congruence S
sage: N.basis()
[
q + 2*q^5 + O(q^6),
q^2 - 3/2*q^5 + O(q^6),
q^3 + O(q^6),
q^4 - 1/2*q^5 + O(q^6)
]

sage: ModularForms(12,4).new_submodule()
Modular Forms subspace of dimension 1 of Modular Forms space of dimension 9 for Congruence S
```

Unfortunately (TODO) - p-new submodules aren't yet implemented:

```
sage: m.new_submodule(3)            # not implemented
...
NotImplementedError
sage: m.new_submodule(11)           # not implemented
...
NotImplementedError
```

prec(*new_prec=None*)

Set or get default initial precision for printing modular forms.

INPUT:

- new_prec - positive integer (default: None)

OUTPUT: if new_prec is None, returns the current precision.

EXAMPLES:

```
sage: M = ModularForms(1,12, prec=3)
sage: M.prec()
3

sage: M.basis()
[
q - 24*q^2 + O(q^3),
1 + 65520/691*q + 134250480/691*q^2 + O(q^3)
]
```

```
sage: M.prec(5)
5
sage: M.basis()
[
q - 24*q^2 + 252*q^3 - 1472*q^4 + O(q^5),
1 + 65520/691*q + 134250480/691*q^2 + 11606736960/691*q^3 + 274945048560/691*q^4 + O(q^5)
]
```

rank()

This is a synonym for `self.dimension()`.

EXAMPLES:

```
sage: m = ModularForms(Gamma0(20),4)
sage: m.rank()
12
sage: m.dimension()
12
```

set_precision(*n*)

Set the default precision for displaying elements of this space.

EXAMPLES:

```
sage: m = ModularForms(Gamma1(5),2)
sage: m.set_precision(10)
sage: m.basis()
[
1 + 60*q^3 - 120*q^4 + 240*q^5 - 300*q^6 + 300*q^7 - 180*q^9 + O(q^10),
q + 6*q^3 - 9*q^4 + 27*q^5 - 28*q^6 + 30*q^7 - 11*q^9 + O(q^10),
q^2 - 4*q^3 + 12*q^4 - 22*q^5 + 30*q^6 - 24*q^7 + 5*q^8 + 18*q^9 + O(q^10)
]
sage: m.set_precision(5)
sage: m.basis()
[
1 + 60*q^3 - 120*q^4 + O(q^5),
q + 6*q^3 - 9*q^4 + O(q^5),
q^2 - 4*q^3 + 12*q^4 + O(q^5)
]
```

44.4 Modular Forms with Character

EXAMPLES:

```
sage: eps = DirichletGroup(13).0
sage: M = ModularForms(eps^2, 2); M
Modular Forms space of dimension 3, character [zeta6] and weight 2 over Cyclotomic Field of order 6 a

sage: S = M.cuspidal_submodule(); S
Cuspidal subspace of dimension 1 of Modular Forms space of dimension 3, character [zeta6] and weight
sage: S.modular_symbols()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 4 and level 13, weight
```

We create a spaces associated to Dirichlet characters of modulus 225:

```
sage: e = DirichletGroup(225).0
sage: e.order()
```

```
6
sage: e.base_ring()
Cyclotomic Field of order 60 and degree 16
sage: M = ModularForms(e,3)
```

Notice that the base ring is "minimized":

```
sage: M
Modular Forms space of dimension 66, character [zeta6, 1] and weight 3
over Cyclotomic Field of order 6 and degree 2
```

If we don't want the base ring to change, we can explicitly specify it:

```
sage: ModularForms(e, 3, e.base_ring())
Modular Forms space of dimension 66, character [zeta6, 1] and weight 3
over Cyclotomic Field of order 60 and degree 16
```

Next we create a space associated to a Dirichlet character of order 20:

```
sage: e = DirichletGroup(225).1
sage: e.order()
20
sage: e.base_ring()
Cyclotomic Field of order 60 and degree 16
sage: M = ModularForms(e,17); M
Modular Forms space of dimension 484, character [1, zeta20] and
weight 17 over Cyclotomic Field of order 20 and degree 8
```

We compute the Eisenstein subspace, which is fast even though the dimension of the space is large (since an explicit basis of q-expansions has not been computed yet).

```
sage: M.eisenstein_submodule()
Eisenstein subspace of dimension 8 of Modular Forms space of
dimension 484, character [1, zeta20] and weight 17 over Cyclotomic Field of order 20 and degree 8

sage: M.cuspidal_submodule()
Cuspidal subspace of dimension 476 of Modular Forms space of dimension 484, character [1, zeta20] and
```

TESTS:

```
sage: m = ModularForms(DirichletGroup(20).1,5)
sage: m == loads(dumps(m))
True
sage: type(m)
<class 'sage.modular.modform.ambient_eps.ModularFormsAmbient_eps'>
```

class ModularFormsAmbient_eps(*character, weight=2, base_ring=None*)

A space of modular forms with character.

base_extend(*R*)

Return space with same defining parameters as this ambient space of modular symbols, but with scalars extended to a different base ring. This differs from change_ring in that it is always possible to coerce elements of the original space to elements of the new one.

EXAMPLES:

```
sage: m = ModularForms(DirichletGroup(13).0^2,2); m
Modular Forms space of dimension 3, character [zeta6] and weight 2 over Cyclotomic Field of
sage: m.base_extend(CyclotomicField(12))
Modular Forms space of dimension 3, character [zeta12^2] and weight 2 over Cyclotomic Field
```

change_ring(*base_ring*)

Return space with same defining parameters as this ambient space of modular symbols, but defined over a different base ring.

EXAMPLES:

```
sage: m = ModularForms(DirichletGroup(13).0^2,2); m
Modular Forms space of dimension 3, character [zeta6] and weight 2 over Cyclotomic Field of
sage: m.change_ring(CyclotomicField(12))
Modular Forms space of dimension 3, character [zeta12^2] and weight 2 over Cyclotomic Field
```

It must be possible to change the ring of the underlying Dirichlet character:

```
sage: m.change_ring(QQ)
...
ValueError: cannot coerce element of order 6 into self
```

cuspidal_submodule()

Return the cuspidal submodule of this ambient space of modular forms.

EXAMPLES:

```
sage: eps = DirichletGroup(4).0
sage: M = ModularForms(eps, 5); M
Modular Forms space of dimension 3, character [-1] and weight 5 over Rational Field
sage: M.cuspidal_submodule()
Cuspidal subspace of dimension 1 of Modular Forms space of dimension 3, character [-1] and w
```

eisenstein_submodule()

Return the submodule of this ambient module with character that is spanned by Eisenstein series. This is the Hecke stable complement of the cuspidal submodule.

EXAMPLES:

```
sage: m = ModularForms(DirichletGroup(13).0^2,2); m
Modular Forms space of dimension 3, character [zeta6] and weight 2 over Cyclotomic Field of
sage: m.eisenstein_submodule()
Eisenstein subspace of dimension 2 of Modular Forms space of dimension 3, character [zeta6]
```

hecke_module_of_level(*N*)

Return the Hecke module of level N corresponding to self, which is the domain or codomain of a degeneracy map from self. Here N must be either a divisor or a multiple of the level of self, and a multiple of the conductor of the character of self.

EXAMPLES:

```
sage: M = ModularForms(DirichletGroup(15).0, 3); M.character().conductor()
3
sage: M.hecke_module_of_level(3)
Modular Forms space of dimension 2, character [-1] and weight 3 over Rational Field
sage: M.hecke_module_of_level(5)
...
ValueError: conductor(=3) must divide M(=5)
sage: M.hecke_module_of_level(30)
Modular Forms space of dimension 16, character [-1, 1] and weight 3 over Rational Field
```

modular_symbols(*sign=0*)

Return corresponding space of modular symbols with given sign.

EXAMPLES:

```
sage: eps = DirichletGroup(13).0
sage: M = ModularForms(eps^2, 2)
sage: M.modular_symbols()
Modular Symbols space of dimension 4 and level 13, weight 2, character [zeta6], sign 0, over
sage: M.modular_symbols(1)
Modular Symbols space of dimension 3 and level 13, weight 2, character [zeta6], sign 1, over
sage: M.modular_symbols(-1)
Modular Symbols space of dimension 1 and level 13, weight 2, character [zeta6], sign -1, ove
sage: M.modular_symbols(2)
...
ValueError: sign must be -1, 0, or 1
```

44.5 Modular Forms for $Gamma_0(N)$ over Q.

TESTS: sage: m = ModularForms(Gamma0(389),6) sage: loads(dumps(m)) == m True

class ModularFormsAmbient_g0_Q (*level, weight*)

A space of modular forms for Gamma_0(N) over QQ.

cuspidal_submodule ()

Return the cuspidal submodule of this space of modular forms for $Gamma_0(N)$.

EXAMPLES: sage: m = ModularForms(Gamma0(33),4) sage: s = m.cuspidal_submodule(); s Cuspidal subspace of dimension 10 of Modular Forms space of dimension 14 for Congruence Subgroup Gamma0(33) of weight 4 over Rational Field sage: type(s) <class 'sage.modular.modform.cuspidal_submodule.CuspidalSubmodule_g0_Q'>

eisenstein_submodule ()

Return the Eisenstein submodule of this space of modular forms for $Gamma_0(N)$.

EXAMPLES: sage: m = ModularForms(Gamma0(389),6) sage: m.eisenstein_submodule() Eisenstein subspace of dimension 2 of Modular Forms space of dimension 163 for Congruence Subgroup Gamma0(389) of weight 6 over Rational Field

44.6 Modular Forms for $\Gamma_1(N)$ over Q.

EXAMPLES:

```
sage: M = ModularForms(Gamma1(13),2); M
Modular Forms space of dimension 13 for Congruence Subgroup Gamma1(13) of weight 2 over Rational Fiel
sage: S = M.cuspidal_submodule(); S
Cuspidal subspace of dimension 2 of Modular Forms space of dimension 13 for Congruence Subgroup Gamma
sage: S.basis()
[
q - 4*q^3 - q^4 + 3*q^5 + O(q^6),
q^2 - 2*q^3 - q^4 + 2*q^5 + O(q^6)
]
```

TESTS:

```
sage: m = ModularForms(Gamma1(20),2)
sage: loads(dumps(m)) == m
True
```

class ModularFormsAmbient_g1_Q(*level, weight*)
A space of modular forms for the group $\Gamma_1(N)$ over the rational numbers.

cuspidal_submodule()
Return the cuspidal submodule of this modular forms space.
EXAMPLES:

```
sage: m = ModularForms(Gamma1(17),2); m
Modular Forms space of dimension 20 for Congruence Subgroup Gamma1(17) of weight 2 over Rati
sage: m.cuspidal_submodule()
Cuspidal subspace of dimension 5 of Modular Forms space of dimension 20 for Congruence Subgr
```

eisenstein_submodule()
Return the Eisenstein submodule of this modular forms space.
EXAMPLES:

```
sage: ModularForms(Gamma1(13),2).eisenstein_submodule()
Eisenstein subspace of dimension 11 of Modular Forms space of dimension 13 for Congruence Su
sage: ModularForms(Gamma1(13),10).eisenstein_submodule()
Eisenstein subspace of dimension 12 of Modular Forms space of dimension 69 for Congruence Su
```

44.7 Modular Forms over a Non-minimal Base Ring

class ModularFormsAmbient_R(*M, base_ring*)

cuspidal_submodule()
Return the cuspidal subspace of this space.
EXAMPLE:

```
sage: C = CuspForms(7, 4, base_ring=CyclotomicField(5)) # indirect doctest
sage: type(C)
<class 'sage.modular.modform.cuspidal_submodule.CuspidalSubmodule_R'>
```

44.8 Submodules of spaces of modular forms

EXAMPLES: sage: M = ModularForms(Gamma1(13),2); M Modular Forms space of dimension 13 for Congruence Subgroup Gamma1(13) of weight 2 over Rational Field sage: M.eisenstein_subspace() Eisenstein subspace of dimension 11 of Modular Forms space of dimension 13 for Congruence Subgroup Gamma1(13) of weight 2 over Rational Field sage: M == loads(dumps(M)) True sage: M.cuspidal_subspace() Cuspidal subspace of dimension 2 of Modular Forms space of dimension 13 for Congruence Subgroup Gamma1(13) of weight 2 over Rational Field

class ModularFormsSubmodule(*ambient_module, submodule, dual=None, check=False*)
A submodule of an ambient space of modular forms.

change_ring(*base_ring*)
Return the base change of this subspace of modular forms to base_ring.

EXAMPLES: sage: M = ModularForms(6,4) ; M.cuspidal_subspace().change_ring(GF(3)) Traceback (most recent call last): ... NotImplementedError: Base change only currently implemented for ambient spaces.

class ModularFormsSubmoduleWithBasis(*ambient_module, submodule, dual=None, check=False*)

44.9 The Cuspidal Subspace

EXAMPLES:

```
sage: S = CuspForms(SL2Z,12); S
Cuspidal subspace of dimension 1 of Modular Forms space of dimension 2 for
Modular Group SL(2,Z) of weight 12 over Rational Field
sage: S.basis()
[
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 + O(q^6)
]

sage: S = CuspForms(Gamma0(33),2); S
Cuspidal subspace of dimension 3 of Modular Forms space of dimension 6 for
Congruence Subgroup Gamma0(33) of weight 2 over Rational Field
sage: S.basis()
[
q - q^5 + O(q^6),
q^2 - q^4 - q^5 + O(q^6),
q^3 + O(q^6)
]

sage: S = CuspForms(Gamma1(3),6); S
Cuspidal subspace of dimension 1 of Modular Forms space of dimension 3 for
Congruence Subgroup Gamma1(3) of weight 6 over Rational Field
sage: S.basis()
[
q - 6*q^2 + 9*q^3 + 4*q^4 + 6*q^5 + O(q^6)
]
```

class **CuspidalSubmodule** (*ambient_space*)

Base class for cuspidal submodules of ambient spaces of modular forms.

modular_symbols (*sign=0*)

Return the corresponding space of modular symbols with the given sign.

EXAMPLES:

```
sage: S = ModularForms(11,2).cuspidal_submodule()
sage: S.modular_symbols()
Modular Symbols subspace of dimension 2 of Modular Symbols space
of dimension 3 for Gamma_0(11) of weight 2 with sign 0 over Rational Field

sage: S.modular_symbols(sign=-1)
Modular Symbols subspace of dimension 1 of Modular Symbols space
of dimension 1 for Gamma_0(11) of weight 2 with sign -1 over Rational Field

sage: M = S.modular_symbols(sign=1); M
Modular Symbols subspace of dimension 1 of Modular Symbols space of
dimension 2 for Gamma_0(11) of weight 2 with sign 1 over Rational Field
sage: M.sign()
1

sage: S = ModularForms(1,12).cuspidal_submodule()
sage: S.modular_symbols()
Modular Symbols subspace of dimension 2 of Modular Symbols space of
dimension 3 for Gamma_0(1) of weight 12 with sign 0 over Rational Field

sage: eps = DirichletGroup(13).0
```

```
sage: S = CuspForms(eps^2, 2)

sage: S.modular_symbols(sign=0)
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 4 and level 13

sage: S.modular_symbols(sign=1)
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 3 and level 13

sage: S.modular_symbols(sign=-1)
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 1 and level 13
```

class CuspidalSubmodule_R(*ambient_space*)
Cuspidal submodule over a non-minimal base ring.

class CuspidalSubmodule_eps(*ambient_space*)
Space of cusp forms with given Dirichlet character.

EXAMPLES:

```
sage: S = CuspForms(DirichletGroup(5).0,5); S
Cuspidal subspace of dimension 1 of Modular Forms space of dimension 3, character [zeta4] and we

sage: S.basis()
[
q + (-zeta4 - 1)*q^2 + (6*zeta4 - 6)*q^3 - 14*zeta4*q^4 + (15*zeta4 + 20)*q^5 + O(q^6)
]
sage: f = S.0
sage: f.qexp()
q + (-zeta4 - 1)*q^2 + (6*zeta4 - 6)*q^3 - 14*zeta4*q^4 + (15*zeta4 + 20)*q^5 + O(q^6)
sage: f.qexp(7)
q + (-zeta4 - 1)*q^2 + (6*zeta4 - 6)*q^3 - 14*zeta4*q^4 + (15*zeta4 + 20)*q^5 + 12*q^6 + O(q^7)
sage: f.qexp(3)
q + (-zeta4 - 1)*q^2 + O(q^3)
sage: f.qexp(2)
q + O(q^2)
sage: f.qexp(1)
O(q^1)
```

class CuspidalSubmodule_g0_Q(*ambient_space*)
Space of cusp forms for $\Gamma_0(N)$ over $\mathbf{Q}$.

class CuspidalSubmodule_g1_Q(*ambient_space*)
Space of cusp forms for $\Gamma_1(N)$ over $\mathbf{Q}$.

class CuspidalSubmodule_level1_Q(*ambient_space*)
Space of cusp forms of level 1 over $\mathbf{Q}$.

class CuspidalSubmodule_modsym_qexp(*ambient_space*)
Cuspidal submodule with q-expansions calculated via modular symbols.

new_submodule(*p=None*)
Return the new subspace of this space of cusp forms. This is computed using modular symbols.

EXAMPLE:

```
sage: CuspForms(55).new_submodule()
Modular Forms subspace of dimension 3 of Modular Forms space of dimension 8 for Congruence S
```

44.10 The Eisenstein Subspace

class EisensteinSubmodule(*ambient_space*)

The Eisenstein submodule of an ambient space of modular forms.

eisenstein_submodule()

Return the Eisenstein submodule of self. (Yes, this is just self.)

EXAMPLES:

```
sage: E = ModularForms(23,4).eisenstein_subspace()
sage: E == E.eisenstein_submodule()
True
```

modular_symbols(*sign=0*)

Return the corresponding space of modular symbols with given sign.

> **Warning:** If sign != 0, then the space of modular symbols will, in general, only correspond to a *subspace* of this space of modular forms. This can be the case for both sign +1 or -1.

EXAMPLES:

```
sage: E = ModularForms(11,2).eisenstein_submodule()
sage: M = E.modular_symbols(); M
Modular Symbols subspace of dimension 1 of Modular Symbols space
of dimension 3 for Gamma_0(11) of weight 2 with sign 0 over Rational Field
sage: M.sign()
0

sage: M = E.modular_symbols(sign=-1); M
Modular Symbols subspace of dimension 0 of Modular Symbols space of
dimension 1 for Gamma_0(11) of weight 2 with sign -1 over Rational Field

sage: E = ModularForms(1,12).eisenstein_submodule()
sage: E.modular_symbols()
Modular Symbols subspace of dimension 1 of Modular Symbols space of
dimension 3 for Gamma_0(1) of weight 12 with sign 0 over Rational Field

sage: eps = DirichletGroup(13).0
sage: E = EisensteinForms(eps^2, 2)
sage: E.modular_symbols()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 4 and level 13
```

class EisensteinSubmodule_eps(*ambient_space*)

Space of Eisenstein forms with given Dirichlet character.

EXAMPLES:

```
sage: e = DirichletGroup(27,CyclotomicField(3)).0**2
sage: M = ModularForms(e,2,prec=10).eisenstein_subspace()
sage: M.dimension()
6

sage: M.eisenstein_series()
[
-1/3*zeta6 - 1/3 + q + (2*zeta6 - 1)*q^2 + q^3 + (-2*zeta6 - 1)*q^4 + (-5*zeta6 + 1)*q^5 + O(q^6
-1/3*zeta6 - 1/3 + q^3 + O(q^6),
q + (-2*zeta6 + 1)*q^2 + (-2*zeta6 - 1)*q^4 + (5*zeta6 - 1)*q^5 + O(q^6),
q + (zeta6 + 1)*q^2 + 3*q^3 + (zeta6 + 2)*q^4 + (-zeta6 + 5)*q^5 + O(q^6),
q^3 + O(q^6),
```

```
q + (-zeta6 - 1)*q^2 + (zeta6 + 2)*q^4 + (zeta6 - 5)*q^5 + O(q^6)
]
sage: M.eisenstein_subspace().T(2).matrix().fcp()
(x + zeta3 + 2) * (x + 2*zeta3 + 1) * (x - 2*zeta3 - 1)^2 * (x - zeta3 - 2)^2
sage: ModularSymbols(e,2).eisenstein_subspace().T(2).matrix().fcp()
(x + zeta3 + 2) * (x + 2*zeta3 + 1) * (x - 2*zeta3 - 1)^2 * (x - zeta3 - 2)^2

sage: M.basis()
[
1 - 3*zeta3*q^6 + (-2*zeta3 + 2)*q^9 + O(q^10),
q + (5*zeta3 + 5)*q^7 + O(q^10),
q^2 - 2*zeta3*q^8 + O(q^10),
q^3 + (zeta3 + 2)*q^6 + 3*q^9 + O(q^10),
q^4 - 2*zeta3*q^7 + O(q^10),
q^5 + (zeta3 + 1)*q^8 + O(q^10)
]
```

class **EisensteinSubmodule_g0_Q**(*ambient_space*)
Space of Eisenstein forms for $\Gamma_0(N)$.

class **EisensteinSubmodule_g1_Q**(*ambient_space*)
Space of Eisenstein forms for $\Gamma_1(N)$.

class **EisensteinSubmodule_params**(*ambient_space*)

change_ring(*base_ring*)
Return self as a module over base_ring.
EXAMPLES:

```
sage: E = EisensteinForms(12,2) ; E
Eisenstein subspace of dimension 5 of Modular Forms space of dimension 5 for Congruence Subg
sage: E.basis()
[
1 + O(q^6),
q + 6*q^5 + O(q^6),
q^2 + O(q^6),
q^3 + O(q^6),
q^4 + O(q^6)
]
sage: E.change_ring(GF(5))
Eisenstein subspace of dimension 5 of Modular Forms space of dimension 5 for Congruence Subg
sage: E.change_ring(GF(5)).basis()
[
1 + O(q^6),
q + q^5 + O(q^6),
q^2 + O(q^6),
q^3 + O(q^6),
q^4 + O(q^6)
]
```

eisenstein_series()
Return the Eisenstein series that span this space (over the algebraic closure).
EXAMPLES:

```
sage: EisensteinForms(11,2).eisenstein_series()
[
5/12 + q + 3*q^2 + 4*q^3 + 7*q^4 + 6*q^5 + O(q^6)
]
```

```
sage: EisensteinForms(1,4).eisenstein_series()
[
1/240 + q + 9*q^2 + 28*q^3 + 73*q^4 + 126*q^5 + O(q^6)
]
sage: EisensteinForms(1,24).eisenstein_series()
[
236364091/131040 + q + 8388609*q^2 + 94143178828*q^3 + 70368752566273*q^4 + 1192092895507812
]
sage: EisensteinForms(5,4).eisenstein_series()
[
1/240 + q + 9*q^2 + 28*q^3 + 73*q^4 + 126*q^5 + O(q^6),
1/240 + q^5 + O(q^6)
]
sage: EisensteinForms(13,2).eisenstein_series()
[
1/2 + q + 3*q^2 + 4*q^3 + 7*q^4 + 6*q^5 + O(q^6)
]

sage: E = EisensteinForms(Gamma1(7),2)
sage: E.set_precision(4)
sage: E.eisenstein_series()
[
1/4 + q + 3*q^2 + 4*q^3 + O(q^4),
1/7*zeta6 - 3/7 + q + (-2*zeta6 + 1)*q^2 + (3*zeta6 - 2)*q^3 + O(q^4),
q + (-zeta6 + 2)*q^2 + (zeta6 + 2)*q^3 + O(q^4),
-1/7*zeta6 - 2/7 + q + (2*zeta6 - 1)*q^2 + (-3*zeta6 + 1)*q^3 + O(q^4),
q + (zeta6 + 1)*q^2 + (-zeta6 + 3)*q^3 + O(q^4)
]

sage: eps = DirichletGroup(13).0^2
sage: ModularForms(eps,2).eisenstein_series()
[
-7/13*zeta6 - 11/13 + q + (2*zeta6 + 1)*q^2 + (-3*zeta6 + 1)*q^3 + (6*zeta6 - 3)*q^4 - 4*q^5
q + (zeta6 + 2)*q^2 + (-zeta6 + 3)*q^3 + (3*zeta6 + 3)*q^4 + 4*q^5 + O(q^6)
]

sage: M = ModularForms(19,3).eisenstein_subspace()
sage: M.eisenstein_series()
[
]
```

new_eisenstein_series()

Return a list of the Eisenstein series in this space that are new.

EXAMPLE:

```
sage: E = EisensteinForms(25, 4)
sage: E.new_eisenstein_series()
[q + 7*zeta4*q^2 - 26*zeta4*q^3 - 57*q^4 + O(q^6),
 q - 9*q^2 - 28*q^3 + 73*q^4 + O(q^6),
 q - 7*zeta4*q^2 + 26*zeta4*q^3 - 57*q^4 + O(q^6)]
```

new_submodule(*p=None*)

Return the new submodule of self.

EXAMPLE:

```
sage: e = EisensteinForms(Gamma0(225), 2).new_submodule(); e
Modular Forms subspace of dimension 3 of Modular Forms space of dimension 42 for Congruence
sage: e.basis()
```

```
[
q + O(q^6),
q^2 + O(q^6),
q^4 + O(q^6)
]
```

parameters()

Return a list of parameters for each Eisenstein series spanning self. That is, for each such series, return a triple of the form (ψ, χ, level), where ψ and χ are the characters defining the Eisenstein series, and level is the smallest level at which this series occurs.

EXAMPLES:

```
sage: ModularForms(24,2).eisenstein_submodule().parameters()
[([1, 1, 1], [1, 1, 1], 2),
([1, 1, 1], [1, 1, 1], 3),
([1, 1, 1], [1, 1, 1], 4),
([1, 1, 1], [1, 1, 1], 6),
([1, 1, 1], [1, 1, 1], 8),
([1, 1, 1], [1, 1, 1], 12),
([1, 1, 1], [1, 1, 1], 24)]
sage: EisensteinForms(12,6).parameters()
[([1, 1], [1, 1], 1),
([1, 1], [1, 1], 2),
([1, 1], [1, 1], 3),
([1, 1], [1, 1], 4),
([1, 1], [1, 1], 6),
([1, 1], [1, 1], 12)]
sage: ModularForms(DirichletGroup(24).0,3).eisenstein_submodule().parameters()
[([1, 1, 1], [-1, 1, 1], 1),
([1, 1, 1], [-1, 1, 1], 2),
([1, 1, 1], [-1, 1, 1], 3),
([1, 1, 1], [-1, 1, 1], 6),
([-1, 1, 1], [1, 1, 1], 1),
([-1, 1, 1], [1, 1, 1], 2),
([-1, 1, 1], [1, 1, 1], 3),
([-1, 1, 1], [1, 1, 1], 6)]
```

cyclotomic_restriction(*L*, *K*)

Given two cyclotomic fields L and K, compute the compositum M of K and L, and return a function and the index [M:K]. The function is a map that acts as follows (here $M = Q(\zeta_m)$):

INPUT:

element alpha in L

OUTPUT:

a polynomial $f(x)$ in $K[x]$ such that $f(\zeta_m) = \alpha$, where we view alpha as living in M. (Note that ζ_m generates M, not L.)

EXAMPLES:

```
sage: L = CyclotomicField(12) ; N = CyclotomicField(33) ; M = CyclotomicField(132)
sage: z, n = sage.modular.modform.eisenstein_submodule.cyclotomic_restriction(L,N)
sage: n
2

sage: z(L.0)
-zeta33^19*x
sage: z(L.0)(M.0)
zeta132^11
```

```
sage: z(L.0^3-L.0+1)
(zeta33^19 + zeta33^8)*x + 1
sage: z(L.0^3-L.0+1)(M.0)
zeta132^33 - zeta132^11 + 1
sage: z(L.0^3-L.0+1)(M.0) - M(L.0^3-L.0+1)
0
```

cyclotomic_restriction_tower(*L, K*)

Suppose L/K is an extension of cyclotomic fields and L=Q(zeta_m). This function computes a map with the following property:

INPUT:

an element alpha in L

OUTPUT:

a polynomial $f(x)$ in $K[x]$ such that $f(zeta_m) = alpha$.

EXAMPLES:

```
sage: L = CyclotomicField(12) ; K = CyclotomicField(6)
sage: z = sage.modular.modform.eisenstein_submodule.cyclotomic_restriction_tower(L,K)
sage: z(L.0)
x
sage: z(L.0^2+L.0)
x + zeta6
```

44.11 Eisenstein Series

compute_eisenstein_params(*character, k*)

Compute and return a list of all parameters (χ, ψ, t) that define the Eisenstein series with given character and weight k.

Only the parity of k is relevant.

If character is an integer N, then the parameters for $\Gamma_1(N)$ are computed instead. Then the condition is that $\chi(-1) * \psi(-1) = (-1)^k$.

EXAMPLES:

```
sage: sage.modular.modform.eis_series.compute_eisenstein_params(DirichletGroup(30)(1), 3)
[]

sage: sage.modular.modform.eis_series.compute_eisenstein_params(DirichletGroup(30)(1), 4)
[([1, 1], [1, 1], 1),
([1, 1], [1, 1], 2),
([1, 1], [1, 1], 3),
([1, 1], [1, 1], 5),
([1, 1], [1, 1], 6),
([1, 1], [1, 1], 10),
([1, 1], [1, 1], 15),
([1, 1], [1, 1], 30)]
```

eisenstein_series_lseries(*weight, prec=53, max_imaginary_part=0, max_asymp_coeffs=40*)

Return the L-series of the weight $2k$ Eisenstein series on $\mathrm{SL}_2(\mathbf{Z})$.

This actually returns an interface to Tim Dokchitser's program for computing with the L-series of the Eisenstein series

INPUT:

- weight - even integer
- prec - integer (bits precision)
- max_imaginary_part - real number
- max_asymp_coeffs - integer

OUTPUT:

The L-series of the Eisenstein series.

EXAMPLES:

We compute with the L-series of E_{16} and then E_{20}:

```
sage: L = eisenstein_series_lseries(16)
sage: L(1)
-0.291657724743873
sage: L = eisenstein_series_lseries(20)
sage: L(2)
-5.02355351645987
```

eisenstein_series_qexp(*k, prec=10, K=Rational Field, var='q'*)

Return the q-expansion of the normalized weight k Eisenstein series to precision prec in the ring K. (The normalization chosen here is the one that forces the coefficient of q to be 1.)

Here's a rough description of how the algorithm works: we know $E_k = const + \sum_n sigma(n, k-1)q^n$. Now, we basically just compute all the $\sigma(n, k-1)$ simultaneously, as σ is multiplicative.

INPUT:

- k - an even positive integer
- prec - (default: 10) a nonnegative integer
- K - (default: QQ) a ring in which -(2*k)/B_k is invertible
- var - (default: 'q') variable name to use for q-expansion

EXAMPLES:

```
sage: eisenstein_series_qexp(2,5)
-1/24 + q + 3*q^2 + 4*q^3 + 7*q^4 + O(q^5)
sage: eisenstein_series_qexp(2,0)
O(q^0)
sage: eisenstein_series_qexp(2,5,GF(7))
2 + q + 3*q^2 + 4*q^3 + O(q^5)
sage: eisenstein_series_qexp(2,5,GF(7),var='T')
2 + T + 3*T^2 + 4*T^3 + O(T^5)

sage: eisenstein_series_qexp(10, 30, GF(17))
15 + q + 3*q^2 + 15*q^3 + 7*q^4 + 13*q^5 + 11*q^6 + 11*q^7 + 15*q^8 + 7*q^9 + 5*q^10 + 7*q^11 +
```

AUTHORS:

- William Stein: original implementation
- Craig Citro (2007-06-01): rewrote for massive speedup

44.12 Elements of modular forms spaces.

class **EisensteinSeries** (*parent, vector, t, chi, psi*)

An Eisenstein series.

EXAMPLES:

```
sage: E = EisensteinForms(1,12)
sage: E.eisenstein_series()
[
691/65520 + q + 2049*q^2 + 177148*q^3 + 4196353*q^4 + 48828126*q^5 + O(q^6)
]
sage: E = EisensteinForms(11,2)
sage: E.eisenstein_series()
[
5/12 + q + 3*q^2 + 4*q^3 + 7*q^4 + 6*q^5 + O(q^6)
]
sage: E = EisensteinForms(Gamma1(7),2)
sage: E.set_precision(4)
sage: E.eisenstein_series()
[
1/4 + q + 3*q^2 + 4*q^3 + O(q^4),
1/7*zeta6 - 3/7 + q + (-2*zeta6 + 1)*q^2 + (3*zeta6 - 2)*q^3 + O(q^4),
q + (-zeta6 + 2)*q^2 + (zeta6 + 2)*q^3 + O(q^4),
-1/7*zeta6 - 2/7 + q + (2*zeta6 - 1)*q^2 + (-3*zeta6 + 1)*q^3 + O(q^4),
q + (zeta6 + 1)*q^2 + (-zeta6 + 3)*q^3 + O(q^4)
]
```

L()

Return the conductor of self.chi().

EXAMPLES:

```
sage: EisensteinForms(DirichletGroup(17).0,99).eisenstein_series()[1].L()
17
```

M()

Return the conductor of self.psi().

EXAMPLES:

```
sage: EisensteinForms(DirichletGroup(17).0,99).eisenstein_series()[1].M()
1
```

character()

Return the character associated to self.

EXAMPLES:

```
sage: EisensteinForms(DirichletGroup(17).0,99).eisenstein_series()[1].character()
[zeta16]

sage: chi = DirichletGroup(7)[4]
sage: E = EisensteinForms(chi).eisenstein_series() ; E
[
-1/7*zeta6 - 2/7 + q + (2*zeta6 - 1)*q^2 + (-3*zeta6 + 1)*q^3 + (-2*zeta6 - 1)*q^4 + (5*zeta
q + (zeta6 + 1)*q^2 + (-zeta6 + 3)*q^3 + (zeta6 + 2)*q^4 + (zeta6 + 4)*q^5 + O(q^6)
]
sage: E[0].character() == chi
True
sage: E[1].character() == chi
True
```

TESTS:

```
sage: [ [ f.character() == chi for f in EisensteinForms(chi).eisenstein_series() ] for chi i
[[True], [], [True, True], [], [True, True], [], [True, True], [], [True, True], [], [True,

sage: [ [ f.character() == chi for f in EisensteinForms(chi).eisenstein_series() ] for chi i
[[True, True, True, True, True], [], [True, True], [], [True, True, True, True], [], [True,
```

chi()

Return the parameter chi associated to self.

EXAMPLES:

```
sage: EisensteinForms(DirichletGroup(17).0,99).eisenstein_series()[1].chi()
[zeta16]
```

new_level()

Return level at which self is new.

EXAMPLES:

```
sage: EisensteinForms(DirichletGroup(17).0,99).eisenstein_series()[1].level()
17
sage: EisensteinForms(DirichletGroup(17).0,99).eisenstein_series()[1].new_level()
17
sage: [ [x.level(), x.new_level()] for x in EisensteinForms(DirichletGroup(60).0^2,2).eisens
[[60, 2], [60, 3], [60, 2], [60, 5], [60, 2], [60, 2], [60, 2], [60, 3], [60, 2], [60, 2], [
```

parameters()

Return chi, psi, and t, which are the defining parameters of self.

EXAMPLES:

```
sage: EisensteinForms(DirichletGroup(17).0,99).eisenstein_series()[1].parameters()
([zeta16], [1], 1)
```

psi()

Return the parameter psi associated to self.

EXAMPLES:

```
sage: EisensteinForms(DirichletGroup(17).0,99).eisenstein_series()[1].psi()
[1]
```

t()

Return the parameter t associated to self.

EXAMPLES:

```
sage: EisensteinForms(DirichletGroup(17).0,99).eisenstein_series()[1].t()
1
```

class **ModularFormElement**(*parent, x, check=True*)

atkin_lehner_eigenvalue(*d=None*)

Return the eigenvalue of the Atkin-Lehner operator W_d acting on this modular form (which is either 1 or -1), or None if this form is not an eigenvector for this operator.

EXAMPLE:

```
sage: CuspForms(1, 30).0.atkin_lehner_eigenvalue()
1
sage: CuspForms(2, 8).0.atkin_lehner_eigenvalue()
...
NotImplementedError: Don't know how to compute Atkin-Lehner matrix acting on this space (try
```

modform_lseries (*prec=53, max_imaginary_part=0, max_asymp_coeffs=40*)

Return the L-series of the weight k modular form f on $\mathrm{SL}_2(\mathbf{Z})$.

This actually returns an interface to Tim Dokchitser's program for computing with the L-series of the modular form.

INPUT:

- `prec` - integer (bits precision)
- `max_imaginary_part` - real number
- `max_asymp_coeffs` - integer

OUTPUT:

The L-series of the modular form.

EXAMPLES:

We commpute with the L-series of the Eisenstein series E_4:

```
sage: f = ModularForms(1,4).0
sage: L = f.modform_lseries()
sage: L(1)
-0.0304484570583933
```

class **ModularFormElement_elliptic_curve** (*parent, E*)

A modular form attached to an elliptic curve.

atkin_lehner_eigenvalue (*d=None*)

Calculate the eigenvalue of the Atkin-Lehner operator W_d acting on this form. If d is None, default to the level of the form. As this form is attached to an elliptic curve, we can read this off from the root number of the curve if d is the level.

EXAMPLE:

```
sage: EllipticCurve('57a1').newform().atkin_lehner_eigenvalue()
1
sage: EllipticCurve('57b1').newform().atkin_lehner_eigenvalue()
-1
sage: EllipticCurve('57b1').newform().atkin_lehner_eigenvalue(19)
1
```

elliptic_curve ()

Return elliptic curve associated to self.

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: f = E.modular_form()
sage: f.elliptic_curve()
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: f.elliptic_curve() is E
True
```

class **ModularForm_abstract** ()

Constructor for generic class of a modular form. This should never be called directly; instead one should instantiate one of the derived classes of this class.

atkin_lehner_eigenvalue (*d=None*)

Return the eigenvalue of the Atkin-Lehner operator W_d acting on self (which is either 1 or -1), or None if this form is not an eigenvector for this operator. If d is not given or is None, use d = the level.

EXAMPLES:

```
sage: sage.modular.modform.element.ModularForm_abstract.atkin_lehner_eigenvalue(CuspForms(2,
...
NotImplementedError
```

base_ring()

Return the base_ring of self.

EXAMPLES:

```
sage: (ModularForms(117, 2).13).base_ring()
Rational Field
sage: (ModularForms(119, 2, base_ring=GF(7)).12).base_ring()
Finite Field of size 7
```

character()

Return the character of self.

EXAMPLES:

```
sage: ModularForms(DirichletGroup(17).0^2,2).2.character()
[zeta8]
```

coefficients(*X*)

The coefficients a_n of self, for integers n>=0 in the list X. If X is an Integer, return coefficients for indices from 1 to X.

This function caches the results of the compute function.

TESTS:

```
sage: e = DirichletGroup(11).gen()
sage: f = EisensteinForms(e, 3).eisenstein_series()[0]
sage: f.coefficients([0,1])
[15/11*zeta10^3 - 9/11*zeta10^2 - 26/11*zeta10 - 10/11,
1]
sage: f.coefficients([0,1,2,3])
[15/11*zeta10^3 - 9/11*zeta10^2 - 26/11*zeta10 - 10/11,
1,
4*zeta10 + 1,
-9*zeta10^3 + 1]
sage: f.coefficients([2,3])
[4*zeta10 + 1,
-9*zeta10^3 + 1]
```

Running this twice once revealed a bug, so we test it:

```
sage: f.coefficients([0,1,2,3])
[15/11*zeta10^3 - 9/11*zeta10^2 - 26/11*zeta10 - 10/11,
1,
4*zeta10 + 1,
-9*zeta10^3 + 1]
```

cuspform_lseries(*prec=53, max_imaginary_part=0, max_asymp_coeffs=40*)

Return the L-series of the weight k cusp form f on $\Gamma_0(N)$.

This actually returns an interface to Tim Dokchitser's program for computing with the L-series of the cusp form.

INPUT:

- `prec` - integer (bits precision)
- `max_imaginary_part` - real number
- `max_asymp_coeffs` - integer

OUTPUT:

The L-series of the cusp form.

EXAMPLES:

```
sage: f = CuspForms(2,8).newforms()[0]
sage: L = f.cuspform_lseries()
sage: L(1)
0.0884317737041015
sage: L(0.5)
0.0296568512531983
```

Consistency check with delta_lseries (which computes coefficients in pari):

```
sage: delta = CuspForms(1,12).0
sage: L = delta.cuspform_lseries()
sage: L(1)
0.0374412812685155
sage: L = delta_lseries()
sage: L(1)
0.0374412812685155
```

We check that #5262 is fixed:

```
sage: E=EllipticCurve('37b2')
sage: h=Newforms(37)[1]
sage: Lh = h.cuspform_lseries()
sage: LE=E.lseries()
sage: Lh(1), LE(1)
(0.725681061936153, 0.725681061936153)
sage: CuspForms(1, 30).0.cuspform_lseries().eps
-1
```

group()

Return the group for which self is a modular form.

EXAMPLES:

```
sage: ModularForms(Gamma1(11), 2).gen(0).group()
Congruence Subgroup Gamma1(11)
```

level()

Return the level of self.

EXAMPLES:

```
sage: ModularForms(25,4).0.level()
25
```

padded_list(*n*)

Return a list of length n whose entries are the first n coefficients of the q-expansion of self.

EXAMPLES:

```
sage: CuspForms(1,12).0.padded_list(20)
[0, 1, -24, 252, -1472, 4830, -6048, -16744, 84480, -113643, -115920, 534612, -370944, -5777
```

prec()

Return the precision to which self.q_expansion() is currently known. Note that this may be 0.

EXAMPLES:

```
sage: M = ModularForms(2,14)
sage: f = M.0
sage: f.prec()
0

sage: M.prec(20)
20
```

```
sage: f.prec()
0
sage: x = f.q_expansion() ; f.prec()
20
```

q_expansion(*prec=None*)

The q-expansion of the modular form to precision $O(q^{\text{prec}})$. This function takes one argument, which is the integer prec.

EXAMPLES:

We compute the cusp form Δ:

```
sage: delta = CuspForms(1,12).0
sage: delta.q_expansion()
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 + O(q^6)
```

We compute the q-expansion of one of the cusp forms of level 23:

```
sage: f = CuspForms(23,2).0
sage: f.q_expansion()
q - q^3 - q^4 + O(q^6)
sage: f.q_expansion(10)
q - q^3 - q^4 - 2*q^6 + 2*q^7 - q^8 + 2*q^9 + O(q^10)
sage: f.q_expansion(2)
q + O(q^2)
sage: f.q_expansion(1)
O(q^1)
sage: f.q_expansion(0)
O(q^0)
```

qexp(*prec=None*)

Same as self.q_expansion(prec).

EXAMPLES:

```
sage: CuspForms(1,12).0.qexp()
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 + O(q^6)
```

valuation()

Return the valuation of self (i.e. as an element of the power series ring in q).

EXAMPLES:

```
sage: ModularForms(11,2).0.valuation()
1
sage: ModularForms(11,2).1.valuation()
0
sage: ModularForms(25,6).1.valuation()
2
sage: ModularForms(25,6).6.valuation()
7
```

weight()

Return the weight of self.

EXAMPLES:

```
sage: (ModularForms(Gamma1(9),2).6).weight()
2
```

class **Newform**(*parent, component, names, check=True*)

abelian_variety()

Return the abelian variety associated to self.

EXAMPLES:

```
sage: Newforms(14,2)[0]
q - q^2 - 2*q^3 + q^4 + O(q^6)
sage: Newforms(14,2)[0].abelian_variety()
Newform abelian subvariety 14a of dimension 1 of J0(14)
```

atkin_lehner_eigenvalue(*d=None*)

Return the eigenvalue of the Atkin-Lehner operator W_d acting on this newform (which is either 1 or -1). A ValueError will be raised if the character of this form is not either trivial or quadratic. If d is not given or is None, then d defaults to the level of self.

EXAMPLE:

```
sage: [x.atkin_lehner_eigenvalue() for x in ModularForms(53).newforms('a')]
[1, -1]
sage: CuspForms(DirichletGroup(5).0, 5).newforms()[0].atkin_lehner_eigenvalue()
...
ValueError: Atkin-Lehner only leaves space invariant when character is trivial or quadratic.
```

element()

Find an element of the ambient space of modular forms which represents this newform.

Note: This can be quite expensive. Also, the polynomial defining the field of Hecke eigenvalues should be considered random, since it is generated by a random sum of Hecke operators. (The field itself is not random, of course.)

EXAMPLES:

```
sage: ls = Newforms(38,4,names='a')
sage: ls[0]
q - 2*q^2 - 2*q^3 + 4*q^4 - 9*q^5 + O(q^6)
sage: ls # random
[q - 2*q^2 - 2*q^3 + 4*q^4 - 9*q^5 + O(q^6),
q - 2*q^2 + (-a1 - 2)*q^3 + 4*q^4 + (2*a1 + 10)*q^5 + O(q^6),
q + 2*q^2 + (1/2*a2 - 1)*q^3 + 4*q^4 + (-3/2*a2 + 12)*q^5 + O(q^6)]
sage: type(ls[0])
<class 'sage.modular.modform.element.Newform'>
sage: ls[2][3].minpoly()
x^2 - 9*x + 2
sage: ls2 = [ x.element() for x in ls ]
sage: ls2 # random
[q - 2*q^2 - 2*q^3 + 4*q^4 - 9*q^5 + O(q^6),
q - 2*q^2 + (-a1 - 2)*q^3 + 4*q^4 + (2*a1 + 10)*q^5 + O(q^6),
q + 2*q^2 + (1/2*a2 - 1)*q^3 + 4*q^4 + (-3/2*a2 + 12)*q^5 + O(q^6)]
sage: type(ls2[0])
<class 'sage.modular.modform.element.ModularFormElement'>
sage: ls2[2][3].minpoly()
x^2 - 9*x + 2
```

hecke_eigenvalue_field()

Return the field generated over the rationals by the coefficients of this newform.

EXAMPLES:

```
sage: ls = Newforms(35, 2, names='a') ; ls
[q + q^3 - 2*q^4 - q^5 + O(q^6),
q + a1*q^2 + (-a1 - 1)*q^3 + (-a1 + 2)*q^4 + q^5 + O(q^6)]
sage: ls[0].hecke_eigenvalue_field()
Rational Field
sage: ls[1].hecke_eigenvalue_field()
Number Field in a1 with defining polynomial x^2 + x - 4
```

modular_symbols(*sign=0*)

Return the subspace with the specified sign of the space of modular symbols corresponding to this newform.

EXAMPLES:

```
sage: f = Newforms(18,4)[0]
sage: f.modular_symbols()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 18 for Gamma_0
sage: f.modular_symbols(1)
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 11 for Gamma_0
```

number()

Return the index of this space in the list of simple, new, cuspidal subspaces of the full space of modular symbols for this weight and level.

EXAMPLES:

```
sage: Newforms(43, 2, names='a')[1].number()
1
```

delta_lseries(*prec=53, max_imaginary_part=0, max_asymp_coeffs=40*)

Return the L-series of the modular form Delta.

This actually returns an interface to Tim Dokchitser's program for computing with the L-series of the modular form Δ.

INPUT:

- `prec` - integer (bits precision)
- `max_imaginary_part` - real number
- `max_asymp_coeffs` - integer

OUTPUT:

The L-series of Δ.

EXAMPLES:

```
sage: L = delta_lseries()
sage: L(1)
0.0374412812685155
```

is_ModularFormElement(*x*)

Return True if x is a modular form.

EXAMPLES:

```
sage: from sage.modular.modform.element import is_ModularFormElement
sage: is_ModularFormElement(5)
False
sage: is_ModularFormElement(ModularForms(11).0)
True
```

44.13 Hecke Operators on q-expansions.

hecke_operator_on_basis(*B, n, k, eps=None, already_echelonized=False*)

Given a basis B of q-expansions for a space of modular forms with character ε to precision at least $\#B \cdot n + 1$, this function computes the matrix of T_n relative to B.

Note: If the elements of B are not known to sufficient precision, this function will report that the vectors are linearly dependent (since they are to the specified precision).

INPUT:

- `B` - list of q-expansions
- `n` - an integer >= 1
- `k` - an integer
- `eps` - Dirichlet character
- `already_echelonized` – bool (default: False); if True, use that the basis is already in Echelon form, which saves a lot of time.

EXAMPLES:

```
sage: sage.modular.modform.constructor.ModularForms_clear_cache()
sage: ModularForms(1,12).q_expansion_basis()
[
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 + O(q^6),
1 + 65520/691*q + 134250480/691*q^2 + 11606736960/691*q^3 + 274945048560/691*q^4 + 3199218815520
]
sage: hecke_operator_on_basis(ModularForms(1,12).q_expansion_basis(), 3, 12)
...
ValueError: The given basis vectors must be linearly independent.

sage: hecke_operator_on_basis(ModularForms(1,12).q_expansion_basis(30), 3, 12)
[   252      0]
[     0 177148]
```

hecke_operator_on_qexp (*f, n, k, eps=None, prec=None, check=True, _return_list=False*)

Given the q-expansion f of a modular form with character ε, this function computes the image of f under the Hecke operator $T_{n,k}$ of weight k.

EXAMPLES:

```
sage: M = ModularForms(1,12)
sage: hecke_operator_on_qexp(M.basis()[0], 3, 12)
252*q - 6048*q^2 + 63504*q^3 - 370944*q^4 + O(q^5)
sage: hecke_operator_on_qexp(M.basis()[0], 1, 12, prec=7)
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 + O(q^7)
sage: hecke_operator_on_qexp(M.basis()[0], 1, 12)
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7 + 84480*q^8 - 113643*q^9 - 115

sage: M.prec(20)
20
sage: hecke_operator_on_qexp(M.basis()[0], 3, 12)
252*q - 6048*q^2 + 63504*q^3 - 370944*q^4 + 1217160*q^5 - 1524096*q^6 + O(q^7)
sage: hecke_operator_on_qexp(M.basis()[0], 1, 12)
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7 + 84480*q^8 - 113643*q^9 - 115

sage: (hecke_operator_on_qexp(M.basis()[0], 1, 12)*252).add_bigoh(7)
252*q - 6048*q^2 + 63504*q^3 - 370944*q^4 + 1217160*q^5 - 1524096*q^6 + O(q^7)

sage: hecke_operator_on_qexp(M.basis()[0], 6, 12)
-6048*q + 145152*q^2 - 1524096*q^3 + O(q^4)
```

An example on a formal power series:

```
sage: R.<q> = QQ[[]]
sage: f = q + q^2 + q^3 + q^7 + O(q^8)
sage: hecke_operator_on_qexp(f, 3, 12)
q + O(q^3)
sage: hecke_operator_on_qexp(delta_qexp(24), 3, 12).prec()
8
sage: hecke_operator_on_qexp(delta_qexp(25), 3, 12).prec()
9
```

44.14 Numerical computation of newforms

class **NumericalEigenforms** (*group, weight=2, eps=9.9999999999999995e-21, delta=0.01, tp=, [2, 3, 5]*)

numerical_eigenforms(group, weight=2, eps=1e-20, delta=1e-2, tp=[2,3,5])

INPUT:

- `group` - a congruence subgroup of a Dirichlet character of order 1 or 2
- `weight` - an integer >= 2
- `eps` - a small float; abs( ) < eps is what "equal to zero" is interpreted as for floating point numbers.
- `delta` - a small-ish float; eigenvalues are considered distinct if their difference has absolute value at least delta
- `tp` - use the Hecke operators T_p for p in tp when searching for a random Hecke operator with distinct Hecke eigenvalues.

OUTPUT:

A numerical eigenforms object, with the following useful methods:

- `ap()` - return all eigenvalues of T_p
- `eigenvalues()` - list of eigenvalues corresponding to the given list of primes, e.g.,:

```
[[eigenvalues of T_2],
 [eigenvalues of T_3],
 [eigenvalues of T_5], ...]
```

- `systems_of_eigenvalues()` - a list of the systems of eigenvalues of eigenforms such that the chosen random linear combination of Hecke operators has multiplicity 1 eigenvalues.

EXAMPLES:

```
sage: n = numerical_eigenforms(23)
sage: n == loads(dumps(n))
True
sage: n.ap(2)
[3.0, 0.61803398875, -1.61803398875]
sage: n.systems_of_eigenvalues(7)
[
[-1.61803398875, 2.2360679775, -3.2360679775],
[0.61803398875, -2.2360679775, 1.2360679775],
[3.0, 4.0, 6.0]
]
sage: n.systems_of_abs(7)
[
[0.6180339887..., 2.236067977..., 1.236067977...],
```

```
[1.6180339887..., 2.236067977..., 3.236067977...],
[3.0, 4.0, 6.0]
]
sage: n.eigenvalues([2,3,5])
[[3.0, 0.61803398875, -1.61803398875],
 [4.0, -2.2360679775, 2.2360679775],
 [6.0, 1.2360679775, -3.2360679775]]
```

ap(*p*)

Return a list of the eigenvalues of the Hecke operator T_p on all the computed eigenforms. The eigenvalues match up between one prime and the next.

INPUT:

- `p` - integer, a prime number

OUTPUT:

- `list` - a list of double precision complex numbers

EXAMPLES:

```
sage: n = numerical_eigenforms(11,4)
sage: n.ap(2) # random order
[9.0, 9.0, 2.73205080757, -0.732050807569]
sage: n.ap(3) # random order
[28.0, 28.0, -7.92820323028, 5.92820323028]
sage: m = n.modular_symbols()
sage: x = polygen(QQ, 'x')
sage: m.T(2).charpoly(x).factor()
(x - 9)^2 * (x^2 - 2*x - 2)
sage: m.T(3).charpoly(x).factor()
(x - 28)^2 * (x^2 + 2*x - 47)
```

eigenvalues(*primes*)

Return the eigenvalues of the Hecke operators corresponding to the primes in the input list of primes. The eigenvalues match up between one prime and the next.

INPUT:

- `primes` - a list of primes

OUTPUT:

list of lists of eigenvalues.

EXAMPLES:

```
sage: n = numerical_eigenforms(1,12)
sage: n.eigenvalues([3,5,13])
[[177148.0, 252.0], [48828126.0, 4830.0], [1.79216039404e+12, -577737.999...]]
```

level()

Return the level of this set of modular eigenforms.

EXAMPLES:

```
sage: n = numerical_eigenforms(61) ; n.level()
61
```

modular_symbols()

Return the space of modular symbols used for computing this set of modular eigenforms.

EXAMPLES:

```
sage: n = numerical_eigenforms(61) ; n.modular_symbols()
Modular Symbols space of dimension 5 for Gamma_0(61) of weight 2 with sign 1 over Rational F
```

systems_of_abs(*bound*)

Return the absolute values of all systems of eigenvalues for self for primes up to bound.

EXAMPLES:

```
sage: numerical_eigenforms(61).systems_of_abs(10)
[
[0.311107817466, 2.90321192591, 2.52542756084, 3.21431974338],
[1.0, 2.0, 3.0, 1.0],
[1.48119430409, 0.806063433525, 3.15632517466, 0.675130870567],
[2.17008648663, 1.70927535944, 1.63089761382, 0.460811127189],
[3.0, 4.0, 6.0, 8.0]
]
```

systems_of_eigenvalues(*bound*)

Return all systems of eigenvalues for self for primes up to bound.

EXAMPLES:

```
sage: numerical_eigenforms(61).systems_of_eigenvalues(10)
[
[-1.48119430409..., 0.806063433525..., 3.15632517466..., 0.675130870567...],
[-1.0..., -2.0..., -3.0..., 1.0...],
[0.311107817466..., 2.90321192591..., -2.52542756084..., -3.21431974338...],
[2.17008648663..., -1.70927535944..., -1.63089761382..., -0.460811127189...],
[3.0, 4.0, 6.0, 8.0]
]
```

weight()

Return the weight of this set of modular eigenforms.

EXAMPLES:

```
sage: n = numerical_eigenforms(61) ; n.weight()
2
```

support(*v, eps*)

Given a vector v and a threshold eps, return all indices where $|v|$ is larger than eps.

EXAMPLES:

```
sage: sage.modular.modform.numerical.support( numerical_eigenforms(61)._easy_vector(), 1.0 )
[]

sage: sage.modular.modform.numerical.support( numerical_eigenforms(61)._easy_vector(), 0.5 )
[0, 1]
```

44.15 The Victor Miller Basis

delta_qexp(*prec=10, var='q', K=Integer Ring*)

Return the q-expansion of Delta as a power series with coefficients in K (=ZZ by default).

INPUT:

- `prec` – integer; the absolute precision of the output
- `var` – (default: 'q') variable name
- `K` – (default: ZZ) base ring of answer

OUTPUT: a power series over K

ALGORITHM: Compute a simple very explicit modular form whose 8th power is Delta. Then compute the 8th power using NTL polynomial arithmetic, which is VERY fast. This function computes a *million* terms of Delta in under a minute.

EXAMPLES:

```
sage: delta_qexp(7)
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 + O(q^7)
sage: delta_qexp(7,'z')
z - 24*z^2 + 252*z^3 - 1472*z^4 + 4830*z^5 - 6048*z^6 + O(z^7)
sage: delta_qexp(-3)
...
ValueError: prec must be positive
```

AUTHORS:

- William Stein: original code
- David Harvey (2007-05): sped up first squaring step

victor_miller_basis(*k, prec=10, cusp_only=False, var='q'*)

Compute and return the Victor-Miller basis for modular forms of weight k and level 1 to precision $O(q^{prec})$. If `cusp_only` is True, return only a basis for the cuspidal subspace.

INPUT:

- `k` – an integer
- `prec` – (default: 10) a positive integer
- `cusp_only` – bool (default: False)
- `var` – string (default: 'q'

OUTPUT: A sequence whose entries are power series in ZZ[[var]].

EXAMPLES:

```
sage: victor_miller_basis(1, 6)
[]
sage: victor_miller_basis(0, 6)
[
1 + O(q^6)
]
sage: victor_miller_basis(2, 6)
[]
sage: victor_miller_basis(4, 6)
[
1 + 240*q + 2160*q^2 + 6720*q^3 + 17520*q^4 + 30240*q^5 + O(q^6)
]

sage: victor_miller_basis(6, 6, var='w')
[
1 - 504*w - 16632*w^2 - 122976*w^3 - 532728*w^4 - 1575504*w^5 + O(w^6)
]

sage: victor_miller_basis(6, 6)
[
1 - 504*q - 16632*q^2 - 122976*q^3 - 532728*q^4 - 1575504*q^5 + O(q^6)
]
sage: victor_miller_basis(12, 6)
```

```
[
1 + 196560*q^2 + 16773120*q^3 + 398034000*q^4 + 4629381120*q^5 + O(q^6),
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 + O(q^6)
]

sage: victor_miller_basis(12, 6, cusp_only=True)
[
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 + O(q^6)
]
sage: victor_miller_basis(24, 6, cusp_only=True)
[
q + 195660*q^3 + 12080128*q^4 + 44656110*q^5 + O(q^6),
q^2 - 48*q^3 + 1080*q^4 - 15040*q^5 + O(q^6)
]
sage: victor_miller_basis(24, 6)
[
1 + 52416000*q^3 + 39007332000*q^4 + 6609020221440*q^5 + O(q^6),
q + 195660*q^3 + 12080128*q^4 + 44656110*q^5 + O(q^6),
q^2 - 48*q^3 + 1080*q^4 - 15040*q^5 + O(q^6)
]
sage: victor_miller_basis(32, 6)
[
1 + 2611200*q^3 + 19524758400*q^4 + 19715347537920*q^5 + O(q^6),
q + 50220*q^3 + 87866368*q^4 + 18647219790*q^5 + O(q^6),
q^2 + 432*q^3 + 39960*q^4 - 1418560*q^5 + O(q^6)
]

sage: victor_miller_basis(40,200)[1:] == victor_miller_basis(40,200,cusp_only=True)
True
sage: victor_miller_basis(200,40)[1:] == victor_miller_basis(200,40,cusp_only=True)
True
```

44.16 Ambient Spaces of Modular Forms.

EXAMPLES:

We compute a basis for the ambient space $M_2(\Gamma_1(25), \chi)$, where χ is quadratic.

```
sage: chi = DirichletGroup(25,QQ).0; chi
[-1]
sage: n = ModularForms(chi,2); n
Modular Forms space of dimension 6, character [-1] and weight 2 over Rational Field
sage: type(n)
<class 'sage.modular.modform.ambient_eps.ModularFormsAmbient_eps'>
```

Compute a basis:

```
sage: n.basis()
[
1 + O(q^6),
q + O(q^6),
q^2 + O(q^6),
q^3 + O(q^6),
q^4 + O(q^6),
```

```
q^5 + O(q^6)
]
```

Compute the same basis but to higher precision:

```
sage: n.set_precision(20)
sage: n.basis()
[
1 + 10*q^10 + 20*q^15 + O(q^20),
q + 5*q^6 + q^9 + 12*q^11 - 3*q^14 + 17*q^16 + 8*q^19 + O(q^20),
q^2 + 4*q^7 - q^8 + 8*q^12 + 2*q^13 + 10*q^17 - 5*q^18 + O(q^20),
q^3 + q^7 + 3*q^8 - q^12 + 5*q^13 + 3*q^17 + 6*q^18 + O(q^20),
q^4 - q^6 + 2*q^9 + 3*q^14 - 2*q^16 + 4*q^19 + O(q^20),
q^5 + q^10 + 2*q^15 + O(q^20)
]
```

TESTS:

```
sage: m = ModularForms(Gamma1(20),2,GF(7))
sage: loads(dumps(m)) == m
True
```

```
sage: m = ModularForms(GammaH(11,[2]), 2); m
Modular Forms space of dimension 2 for Congruence Subgroup Gamma_H(11) with H generated by [2] of wei
sage: type(m)
<class 'sage.modular.modform.ambient.ModularFormsAmbient'>
```

class ModularFormsAmbient (*group, weight, base_ring, character=None*)

An ambient space of modular forms.

ambient_space()

Return the ambient space that contains this ambient space. This is, of course, just this space again.

EXAMPLES:

```
sage: m = ModularForms(Gamma0(3),30)
sage: m.ambient_space() is m
True
```

change_ring(*base_ring*)

Change the base ring of this space of modular forms.

INPUT:

- R - ring

EXAMPLES:

```
sage: M = ModularForms(Gamma0(37),2)
sage: M.basis()
[
q + q^3 - 2*q^4 + O(q^6),
q^2 + 2*q^3 - 2*q^4 + q^5 + O(q^6),
1 + 2/3*q + 2*q^2 + 8/3*q^3 + 14/3*q^4 + 4*q^5 + O(q^6)
]
```

The basis after changing the base ring is the reduction modulo 3 of an integral basis.

```
sage: M3 = M.change_ring(GF(3))
sage: M3.basis()
[
```

```
1 + q^3 + q^4 + 2*q^5 + O(q^6),
q + q^3 + q^4 + O(q^6),
q^2 + 2*q^3 + q^4 + q^5 + O(q^6)
]
```

cuspidal_submodule()

Return the cuspidal submodule of this ambient module.

EXAMPLES:

```
sage: ModularForms(Gamma1(13)).cuspidal_submodule()
Cuspidal subspace of dimension 2 of Modular Forms space of dimension 13 for
Congruence Subgroup Gamma1(13) of weight 2 over Rational Field
```

dimension()

Return the dimension of this ambient space of modular forms, computed using a dimension formula (so it should be reasonably fast).

EXAMPLES:

```
sage: m = ModularForms(Gamma1(20),20)
sage: m.dimension()
238
```

eisenstein_params()

Return parameters that define all Eisenstein series in self.

OUTPUT: an immutable Sequence

EXAMPLES:

```
sage: m = ModularForms(Gamma0(22), 2)
sage: v = m.eisenstein_params(); v
[([1], [1], 2), ([1], [1], 11), ([1], [1], 22)]
sage: type(v)
<class 'sage.structure.sequence.Sequence'>
```

eisenstein_series()

Return all Eisenstein series associated to this space.

```
sage: ModularForms(27,2).eisenstein_series()
[
q^3 + O(q^6),
q - 3*q^2 + 7*q^4 - 6*q^5 + O(q^6),
1/12 + q + 3*q^2 + q^3 + 7*q^4 + 6*q^5 + O(q^6),
1/3 + q + 3*q^2 + 4*q^3 + 7*q^4 + 6*q^5 + O(q^6),
13/12 + q + 3*q^2 + 4*q^3 + 7*q^4 + 6*q^5 + O(q^6)
]

sage: ModularForms(Gamma1(5),3).eisenstein_series()
[
-1/5*zeta4 - 2/5 + q + (4*zeta4 + 1)*q^2 + (-9*zeta4 + 1)*q^3 + (4*zeta4 - 15)*q^4 + q^5 + O
q + (zeta4 + 4)*q^2 + (-zeta4 + 9)*q^3 + (4*zeta4 + 15)*q^4 + 25*q^5 + O(q^6),
1/5*zeta4 - 2/5 + q + (-4*zeta4 + 1)*q^2 + (9*zeta4 + 1)*q^3 + (-4*zeta4 - 15)*q^4 + q^5 + O
q + (-zeta4 + 4)*q^2 + (zeta4 + 9)*q^3 + (-4*zeta4 + 15)*q^4 + 25*q^5 + O(q^6)
]

sage: eps = DirichletGroup(13).0^2
sage: ModularForms(eps,2).eisenstein_series()
[
-7/13*zeta6 - 11/13 + q + (2*zeta6 + 1)*q^2 + (-3*zeta6 + 1)*q^3 + (6*zeta6 - 3)*q^4 - 4*q^5
q + (zeta6 + 2)*q^2 + (-zeta6 + 3)*q^3 + (3*zeta6 + 3)*q^4 + 4*q^5 + O(q^6)
]
```

eisenstein_submodule()
Return the Eisenstein submodule of this ambient module.
EXAMPLES:

```
sage: m = ModularForms(Gamma1(13),2); m
Modular Forms space of dimension 13 for Congruence Subgroup Gamma1(13) of weight 2 over Rati
sage: m.eisenstein_submodule()
Eisenstein subspace of dimension 11 of Modular Forms space of dimension 13 for Congruence Su
```

hecke_module_of_level(*N*)
Return the Hecke module of level N corresponding to self, which is the domain or codomain of a degeneracy map from self. Here N must be either a divisor or a multiple of the level of self.
EXAMPLES:

```
sage: ModularForms(25, 6).hecke_module_of_level(5)
Modular Forms space of dimension 3 for Congruence Subgroup Gamma0(5) of weight 6 over Ration
sage: ModularForms(Gamma1(4), 3).hecke_module_of_level(8)
Modular Forms space of dimension 7 for Congruence Subgroup Gamma1(8) of weight 3 over Ration
sage: ModularForms(Gamma1(4), 3).hecke_module_of_level(9)
...
ValueError: N (=9) must be a divisor or a multiple of the level of self (=4)
```

is_ambient()
Return True if this an ambient space of modular forms.
This is an ambient space, so this function always returns True.
EXAMPLES:

```
sage: ModularForms(11).is_ambient()
True
sage: CuspForms(11).is_ambient()
False
```

modular_symbols(*sign=0*)
Return the corresponding space of modular symbols with the given sign.
EXAMPLES:

```
sage: S = ModularForms(11,2)
sage: S.modular_symbols()
Modular Symbols space of dimension 3 for Gamma_0(11) of weight 2 with sign 0 over Rational F
sage: S.modular_symbols(sign=1)
Modular Symbols space of dimension 2 for Gamma_0(11) of weight 2 with sign 1 over Rational F
sage: S.modular_symbols(sign=-1)
Modular Symbols space of dimension 1 for Gamma_0(11) of weight 2 with sign -1 over Rational

sage: ModularForms(1,12).modular_symbols()
Modular Symbols space of dimension 3 for Gamma_0(1) of weight 12 with sign 0 over Rational F
```

module()
Return the underlying free module corresponding to this space of modular forms. This is a free module (viewed as a tuple space) of the same dimension as this space over the same base ring.
EXAMPLES:

```
sage: m = ModularForms(Gamma1(13),10)
sage: m.free_module()
Vector space of dimension 69 over Rational Field
sage: ModularForms(Gamma1(13),4, GF(49,'b')).free_module()
Vector space of dimension 27 over Finite Field in b of size 7^2
```

new_submodule(*p=None*)
Return the new or p-new submodule of this ambient module.

INPUT:

- p - (default: None), if specified return only the p-new submodule.

EXAMPLES:

```
sage: m = ModularForms(Gamma0(33),2); m
Modular Forms space of dimension 6 for Congruence Subgroup Gamma0(33) of weight 2 over Ratio
sage: m.new_submodule()
Modular Forms subspace of dimension 1 of Modular Forms space of dimension 6 for Congruence S
```

Another example:

```
sage: M = ModularForms(17,4)
sage: N = M.new_subspace(); N
Modular Forms subspace of dimension 4 of Modular Forms space of dimension 6 for Congruence S
sage: N.basis()
[
q + 2*q^5 + O(q^6),
q^2 - 3/2*q^5 + O(q^6),
q^3 + O(q^6),
q^4 - 1/2*q^5 + O(q^6)
]

sage: ModularForms(12,4).new_submodule()
Modular Forms subspace of dimension 1 of Modular Forms space of dimension 9 for Congruence S
```

Unfortunately (TODO) - p-new submodules aren't yet implemented:

```
sage: m.new_submodule(3)            # not implemented
...
NotImplementedError
sage: m.new_submodule(11)           # not implemented
...
NotImplementedError
```

prec(*new_prec=None*)

Set or get default initial precision for printing modular forms.

INPUT:

- new_prec - positive integer (default: None)

OUTPUT: if new_prec is None, returns the current precision.

EXAMPLES:

```
sage: M = ModularForms(1,12, prec=3)
sage: M.prec()
3

sage: M.basis()
[
q - 24*q^2 + O(q^3),
1 + 65520/691*q + 134250480/691*q^2 + O(q^3)
]

sage: M.prec(5)
5
sage: M.basis()
[
q - 24*q^2 + 252*q^3 - 1472*q^4 + O(q^5),
1 + 65520/691*q + 134250480/691*q^2 + 11606736960/691*q^3 + 274945048560/691*q^4 + O(q^5)
]
```

rank()
This is a synonym for `self.dimension()`.
EXAMPLES:

```
sage: m = ModularForms(Gamma0(20),4)
sage: m.rank()
12
sage: m.dimension()
12
```

set_precision(*n*)
Set the default precision for displaying elements of this space.
EXAMPLES:

```
sage: m = ModularForms(Gamma1(5),2)
sage: m.set_precision(10)
sage: m.basis()
[
1 + 60*q^3 - 120*q^4 + 240*q^5 - 300*q^6 + 300*q^7 - 180*q^9 + O(q^10),
q + 6*q^3 - 9*q^4 + 27*q^5 - 28*q^6 + 30*q^7 - 11*q^9 + O(q^10),
q^2 - 4*q^3 + 12*q^4 - 22*q^5 + 30*q^6 - 24*q^7 + 5*q^8 + 18*q^9 + O(q^10)
]
sage: m.set_precision(5)
sage: m.basis()
[
1 + 60*q^3 - 120*q^4 + O(q^5),
q + 6*q^3 - 9*q^4 + O(q^5),
q^2 - 4*q^3 + 12*q^4 + O(q^5)
]
```

44.17 Compute spaces of half-integral weight modular forms.

Based on an algorithm in Basmaji's thesis.

AUTHORS:

- William Stein (2007-08)

half_integral_weight_modform_basis(*chi, k, prec*)
A basis for the space of weight $k/2$ forms with character χ. The modulus of χ must be divisible by 16 and k must be odd and > 1.
INPUT:

- `chi` - a Dirichlet character with modulus divisible by 16
- `k` - an odd integer = 1
- `prec` - a positive integer

OUTPUT: a list of power series

Warning:

1. This code is very slow because it requests computation of a basis of modular forms for integral weight spaces, and that computation is still very slow.
2. If you give an input prec that is too small, then the output list of power series may be larger than the dimension of the space of half-integral forms.

EXAMPLES:

We compute some half-integral weight forms of level 16*7

```
sage: half_integral_weight_modform_basis(DirichletGroup(16*7).0^2,3,30)
[q - 2*q^2 - q^9 + 2*q^14 + 6*q^18 - 2*q^21 - 4*q^22 - q^25 + O(q^30),
 q^2 - q^14 - 3*q^18 + 2*q^22 + O(q^30),
 q^4 - q^8 - q^16 + q^28 + O(q^30),
 q^7 - 2*q^15 + O(q^30)]
```

The following illustrates that choosing too low of a precision can give an incorrect answer.

```
sage: half_integral_weight_modform_basis(DirichletGroup(16*7).0^2,3,20)
[q - 2*q^2 - q^9 + 2*q^14 + 6*q^18 + O(q^20),
 q^2 - q^14 - 3*q^18 + O(q^20),
 q^4 - 2*q^8 + 2*q^12 - 4*q^16 + O(q^20),
 q^7 - 2*q^8 + 4*q^12 - 2*q^15 - 6*q^16 + O(q^20),
 q^8 - 2*q^12 + 3*q^16 + O(q^20)]
```

We compute some spaces of low level and the first few possible weights.

```
sage: half_integral_weight_modform_basis(DirichletGroup(16,QQ).1, 3, 10)
[]
sage: half_integral_weight_modform_basis(DirichletGroup(16,QQ).1, 5, 10)
[q - 2*q^3 - 2*q^5 + 4*q^7 - q^9 + O(q^10)]
sage: half_integral_weight_modform_basis(DirichletGroup(16,QQ).1, 7, 10)
[q - 2*q^2 + 4*q^3 + 4*q^4 - 10*q^5 - 16*q^7 + 19*q^9 + O(q^10),
 q^2 - 2*q^3 - 2*q^4 + 4*q^5 + 4*q^7 - 8*q^9 + O(q^10),
 q^3 - 2*q^5 - 2*q^7 + 4*q^9 + O(q^10)]
sage: half_integral_weight_modform_basis(DirichletGroup(16,QQ).1, 9, 10)
[q - 2*q^2 + 4*q^3 - 8*q^4 + 14*q^5 + 16*q^6 - 40*q^7 + 16*q^8 - 57*q^9 + O(q^10),
 q^2 - 2*q^3 + 4*q^4 - 8*q^5 - 8*q^6 + 20*q^7 - 8*q^8 + 32*q^9 + O(q^10),
 q^3 - 2*q^4 + 4*q^5 + 4*q^6 - 10*q^7 - 16*q^9 + O(q^10),
 q^4 - 2*q^5 - 2*q^6 + 4*q^7 + 4*q^9 + O(q^10),
 q^5 - 2*q^7 - 2*q^9 + O(q^10)]
```

This example once raised an error (see trac #5792).

```
sage: half_integral_weight_modform_basis(trivial_character(16),9,10)
[q - 2*q^2 + 4*q^3 - 8*q^4 + 4*q^6 - 16*q^7 + 48*q^8 - 15*q^9 + O(q^10),
 q^2 - 2*q^3 + 4*q^4 - 2*q^6 + 8*q^7 - 24*q^8 + O(q^10),
 q^3 - 2*q^4 - 4*q^7 + 12*q^8 + O(q^10),
 q^4 - 6*q^8 + O(q^10)]
```

ALGORITHM: Basmaji (page 55 of his Essen thesis, "Ein Algorithmus zur Berechnung von Hecke-Operatoren und Anwendungen auf modulare Kurven", http://wstein.org/scans/papers/basmaji/).

Let $S = S_{k+1}(\epsilon)$ be the space of cusp forms of even integer weight $k+1$ and character $\varepsilon = \chi\psi^{(k+1)/2}$, where ψ is the nontrivial mod-4 Dirichlet character. Let U be the subspace of $S \times S$ of elements (a, b) such that $\Theta_2 a = \Theta_3 b$. Then U is isomorphic to $S_{k/2}(\chi)$ via the map $(a, b) \mapsto a/\Theta_3$.

44.18 Graded Rings of Modular Forms

This module contains functions to find generators for the graded ring of modular forms of given level.

AUTHORS:

- William Stein (2007-08-24): first version

class ModularFormsRing(*group*)

The ring of modular forms (of weights 0 or at least 2) for a congruence subgroup of $\mathrm{SL}_2(\mathbf{Z})$.

EXAMPLES:

```
sage: ModularFormsRing(Gamma1(13))
Ring of modular forms for Congruence Subgroup Gamma1(13) of weights 0 and at least 2
sage: m = ModularFormsRing(4); m
Ring of modular forms for Congruence Subgroup Gamma0(4)
sage: m.modular_forms_of_weight(2)
Modular Forms space of dimension 2 for Congruence Subgroup Gamma0(4) of weight 2 over Rational F
sage: m.modular_forms_of_weight(10)
Modular Forms space of dimension 6 for Congruence Subgroup Gamma0(4) of weight 10 over Rational
sage: m == loads(dumps(m))
True
sage: m.generators()
[(2, 1 + 24*q^2 + 24*q^4 + 96*q^6 + 24*q^8 + O(q^10)),
(2, q + 4*q^3 + 6*q^5 + 8*q^7 + 13*q^9 + O(q^10))]
sage: m.q_expansion_basis(2,10)
[1 + 24*q^2 + 24*q^4 + 96*q^6 + 24*q^8 + O(q^10),
 q + 4*q^3 + 6*q^5 + 8*q^7 + 13*q^9 + O(q^10)]
sage: m.q_expansion_basis(3,10)
[]
sage: m.q_expansion_basis(10,10)
[1 + 10560*q^6 + 3960*q^8 + O(q^10),
 q - 8056*q^7 - 30855*q^9 + O(q^10),
 q^2 - 796*q^6 - 8192*q^8 + O(q^10),
 q^3 + 66*q^7 + 832*q^9 + O(q^10),
 q^4 + 40*q^6 + 528*q^8 + O(q^10),
 q^5 + 20*q^7 + 190*q^9 + O(q^10)]
```

generators(*prec=10, maxweight=20*)

Calculate modular forms generating a subring that contains all forms of weight up to maxweight (default 20).

EXAMPLES:

```
sage: ModularFormsRing(SL2Z).generators()
[(4, 1 + 240*q + 2160*q^2 + 6720*q^3 + 17520*q^4 + 30240*q^5 + 60480*q^6 + 82560*q^7 + 14040
sage: ModularFormsRing(SL2Z).generators(maxweight=5)
[(4, 1 + 240*q + 2160*q^2 + 6720*q^3 + 17520*q^4 + 30240*q^5 + 60480*q^6 + 82560*q^7 + 14040
```

modular_forms_of_weight(*weight*)

Return the space of modular forms on this group of the given weight.

EXAMPLES:

```
sage: R = ModularFormsRing(13)
sage: R.modular_forms_of_weight(10)
Modular Forms space of dimension 11 for Congruence Subgroup Gamma0(13) of weight 10 over Rat
sage: ModularFormsRing(Gamma1(13)).modular_forms_of_weight(3)
Modular Forms space of dimension 20 for Congruence Subgroup Gamma1(13) of weight 3 over Rati
```

q_expansion_basis(*weight, prec=None*)

Calculate a basis of q-expansions for the space of modular forms of the given weight for this group, calculated using the ring generators given by `find_generators`.

EXAMPLES:

```
sage: m = ModularFormsRing(Gamma0(4))
sage: m.q_expansion_basis(2,10)
[1 + 24*q^2 + 24*q^4 + 96*q^6 + 24*q^8 + O(q^10),
```

```
q + 4*q^3 + 6*q^5 + 8*q^7 + 13*q^9 + O(q^10)]
sage: m.q_expansion_basis(3,10)
[]
```

basis_for_modform_space(*gens, group, weight*)

Given a list of pairs `(k,f)` of a weight and a modular form of that weight, and a target weight l, return a basis of q-expansions for the weight l part of the graded algebra generated by those forms (which may or may not be the whole space of weight l forms for the given group).

EXAMPLES:

```
sage: X = ModularFormsRing(SL2Z).generators()
sage: sage.modular.modform.find_generators.basis_for_modform_space(X, SL2Z, 12)
[1 + 196560*q^2 + 16773120*q^3 + 398034000*q^4 + 4629381120*q^5 + 34417656000*q^6 + 187489935360
q - 24*q^2 + 252*q^3 - 1472*q^4 + 4830*q^5 - 6048*q^6 - 16744*q^7 + 84480*q^8 - 113643*q^9 + O(q
```

modform_generators(*group, maxweight=20, prec=None, start_gens=, [], start_weight=2*)

Find modular forms in $M_k(group)$ for $k \leq maxweight$, such that these forms generate – as an algebra – all forms on group of weight up to maxweight, where all forms are computed as q-expansions to precision prec.

INPUT:

- `group` – a level or a congruence subgroup
- `maxweight` – integer
- `prec` – integer (default: twice largest dimension)
- `start_gens` – list of pairs (k,f) where k is an integer and f is a power seris (default: []); if given, we assume the given pairs (k,f) are q-expansions of modular form of the given weight, and start creating modular forms generators using them.
- `start_weight` – an integer (default: 2)

OUTPUT:

a list of pairs (k, f), where f is the q-expansion of a modular form of weight k.

EXAMPLES:

```
sage: import sage.modular.modform.find_generators as fg
sage: forms = [(4, 240*eisenstein_series_qexp(4,5)), (6,504*eisenstein_series_qexp(6,5))]
sage: fg.multiply_forms_to_weight(forms, 12)
[(12, 1 - 1008*q + 220752*q^2 + 16519104*q^3 + 399517776*q^4 + O(q^5)), (12, 1 + 720*q + 179280*
sage: fg.multiply_forms_to_weight(forms, 24)
[(24, 1 - 2016*q + 1457568*q^2 - 411997824*q^3 + 16227967392*q^4 + O(q^5)), (24, 1 - 288*q - 325
sage: dimension_modular_forms(SL2Z,24)
3

sage: fg.modform_generators(1)
[(4, 1 + 240*q + 2160*q^2 + 6720*q^3 + O(q^4)), (6, 1 - 504*q - 16632*q^2 - 122976*q^3 + O(q^4))
sage: fg.modform_generators(2)
[(2, 1 + 24*q + 24*q^2 + 96*q^3 + 24*q^4 + 144*q^5 + 96*q^6 + 192*q^7 + 24*q^8 + 312*q^9 + 144*q
sage: fg.modform_generators(4, 12, 20)
[(2, 1 + 24*q^2 + 24*q^4 + 96*q^6 + 24*q^8 + 144*q^10 + 96*q^12 + 192*q^14 + 24*q^16 + 312*q^18
```

Here we see that for `\Gamma_0(11)` taking a basis of forms in weights 2 and 4 is enough to generate everything up to weight 12 (and probably everything else).:

```
sage: v = fg.modform_generators(11, 12)
sage: len(v)
3
```

```
sage: [k for k, _ in v]
[2, 2, 4]
sage: dimension_modular_forms(11,2)
2
sage: dimension_modular_forms(11,4)
4
```

For congruence subgroups not -1, we miss out some forms since we can't calculate weight 1 forms at present, but we can still find generators for the ring of forms of weight ≥ 2:

```
sage: fg.modform_generators(Gamma1(4), prec=10, maxweight=10)
[(2, 1 + 24*q^2 + 24*q^4 + 96*q^6 + 24*q^8 + O(q^10)),
(2, q + 4*q^3 + 6*q^5 + 8*q^7 + 13*q^9 + O(q^10)),
(3, 1 + 12*q^2 + 64*q^3 + 60*q^4 + 160*q^6 + 384*q^7 + 252*q^8 + O(q^10)),
(3, q + 4*q^2 + 8*q^3 + 16*q^4 + 26*q^5 + 32*q^6 + 48*q^7 + 64*q^8 + 73*q^9 + O(q^10))]
```

multiply_forms_to_weight(*forms, weight, stop_dim=None*)

Given a list of pairs `(k,f)`, where k is an integer and f is a power series, and a weight l, return all weight l forms obtained by multiplying together the given forms.

INPUT:

- forms – list of pairs (k, f) with k an integer and f a power series
- weight – an integer
- stop_dim – integer (optional): if set to an integer and we find that the series so far span a space of at least this dimension, then stop multiplying more forms together.

EXAMPLES:

```
sage: import sage.modular.modform.find_generators as f
sage: forms = [(4, 240*eisenstein_series_qexp(4,5)), (6,504*eisenstein_series_qexp(6,5))]
sage: f.multiply_forms_to_weight(forms, 12)
[(12, 1 - 1008*q + 220752*q^2 + 16519104*q^3 + 399517776*q^4 + O(q^5)), (12, 1 + 720*q + 179280*
sage: f.multiply_forms_to_weight(forms, 24)
[(24, 1 - 2016*q + 1457568*q^2 - 411997824*q^3 + 16227967392*q^4 + O(q^5)), (24, 1 - 288*q - 325
sage: dimension_modular_forms(SL2Z,24)
3
```

span_of_series(*v, prec=None, basis=False*)

Return the free module spanned by the given list of power series or objects with a padded_list method. If prec is not given, the precision used is the minimum of the precisions of the elements in the list (as determined by a prec method).

INPUT:

- v – a list of power series
- prec – optional; if given then the series do not have to be of finite precision, and will be considered to have precision prec.
- basis – (default: False) if True the input is assumed to determine linearly independent vectors, and the resulting free module has that as basis.

OUTPUT:

A free module of rank prec over the base ring of the forms (actually, of the first form in the list). If the list is empty, the free module is over QQ.

EXAMPLES:

An example involving modular forms:

```
sage: from sage.modular.modform.find_generators import span_of_series
sage: v = ModularForms(11,2, prec=5).basis(); v
[
q - 2*q^2 - q^3 + 2*q^4 + O(q^5),
1 + 12/5*q + 36/5*q^2 + 48/5*q^3 + 84/5*q^4 + O(q^5)
]
sage: span_of_series(v)
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 12 12 12]
[ 0  1 -2 -1  2]
```

Next we make sure the vectors give a basis:

```
sage: span_of_series(v,basis=True)
Vector space of degree 5 and dimension 2 over Rational Field
User basis matrix:
[   0    1   -2   -1    2]
[   1 12/5 36/5 48/5 84/5]
```

An example involving power series.:

```
sage: R.<x> = PowerSeriesRing(QQ, default_prec=5)
sage: v = [1/(1-x), 1/(1+x), 2/(1+x), 2/(1-x)]; v
[1 + x + x^2 + x^3 + x^4 + O(x^5),
 1 - x + x^2 - x^3 + x^4 + O(x^5),
 2 - 2*x + 2*x^2 - 2*x^3 + 2*x^4 + O(x^5),
 2 + 2*x + 2*x^2 + 2*x^3 + 2*x^4 + O(x^5)]
sage: span_of_series(v)
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[1 0 1 0 1]
[0 1 0 1 0]
sage: span_of_series(v,10)
Vector space of degree 10 and dimension 2 over Rational Field
Basis matrix:
[1 0 1 0 1 0 0 0 0 0]
[0 1 0 1 0 0 0 0 0 0]
```

An example involving polynomials.:

```
sage: x = polygen(QQ)
sage: span_of_series([x^3, 2*x^2 + 17*x^3, x^2])
...
ValueError: please specify a precision
sage: span_of_series([x^3, 2*x^2 + 17*x^3, x^2],5)
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[0 0 1 0 0]
[0 0 0 1 0]
sage: span_of_series([x^3, 2*x^2 + 17*x^3, x^2],3)
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 0 1]
```

CHAPTER

FORTYFIVE

MODULAR ABELIAN VARIETIES

45.1 Constructors for certain modular abelian varieties.

AUTHORS:

- William Stein (2007-03)

AbelianVariety(*X*)

Create the abelian variety corresponding to the given definining data.

INPUT:

- X - an integer, string, newform, modsym space, congruence subgroup or tuple of congruence subgroups

OUTPUT: a modular abelian variety

EXAMPLES:

```
sage: AbelianVariety(Gamma0(37))
Abelian variety J0(37) of dimension 2
sage: AbelianVariety('37a')
Newform abelian subvariety 37a of dimension 1 of J0(37)
sage: AbelianVariety(Newform('37a'))
Newform abelian subvariety 37a of dimension 1 of J0(37)
sage: AbelianVariety(ModularSymbols(37).cuspidal_submodule())
Abelian variety J0(37) of dimension 2
sage: AbelianVariety((Gamma0(37), Gamma0(11)))
Abelian variety J0(37) x J0(11) of dimension 3
sage: AbelianVariety(37)
Abelian variety J0(37) of dimension 2
sage: AbelianVariety([1,2,3])
...
TypeError: X must be an integer, string, newform, modsym space, congruence subgroup or tuple of
```

J0(*N*)

Return the Jacobian $J_0(N)$ of the modular curve $X_0(N)$.

EXAMPLES:

```
sage: J0(389)
Abelian variety J0(389) of dimension 32
```

The result is cached:

```
sage: J0(33) is J0(33)
True
```

J1(*N*)
Return the Jacobian $J_1(N)$ of the modular curve $X_1(N)$.

EXAMPLES:

```
sage: J1(389)
Abelian variety J1(389) of dimension 6112
```

JH(*N*, *H*)
Return the Jacobian $J_H(N)$ of the modular curve $X_H(N)$.

EXAMPLES:

```
sage: JH(389,[2])
Abelian variety JH(389,[2]) of dimension 32
```

45.2 Base class for modular abelian varieties

AUTHORS:

- William Stein (2007-03)

TESTS:

```
sage: A = J0(33)
sage: D = A.decomposition(); D
[
Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33),
Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33),
Simple abelian subvariety 33a(1,33) of dimension 1 of J0(33)
]
sage: loads(dumps(D)) == D
True
sage: loads(dumps(A)) == A
True
```

class ModularAbelianVariety(*groups, lattice=None, base_field=Rational Field, is_simple=None, newform_level=None, isogeny_number=None, number=None, check=True*)

lattice()
Return the lattice that defines this abelan variety.

OUTPUT:

•`lattice` - a lattice embedded in the rational homology of the ambient product Jacobian

EXAMPLES:

```
sage: A = (J0(11) * J0(37))[1]; A
Simple abelian subvariety 37a(1,37) of dimension 1 of J0(11) x J0(37)
sage: type(A)
<class 'sage.modular.abvar.abvar.ModularAbelianVariety'>
sage: A.lattice()
Free module of degree 6 and rank 2 over Integer Ring
```

```
Echelon basis matrix:
[ 0  0  1 -1  1  0]
[ 0  0  0  0  2 -1]
```

class ModularAbelianVariety_abstract(*groups, base_field, is_simple=None, newform_level=None, isogeny_number=None, number=None, check=True*)

ambient_morphism()

Return the morphism from self to the ambient variety. This is injective if self is natural a subvariety of the ambient product Jacobian.

OUTPUT: morphism

The output is cached.

EXAMPLES: We compute the ambient structure morphism for an abelian subvariety of $J_0(33)$:

```
sage: A,B,C = J0(33)
sage: phi = A.ambient_morphism()
sage: phi.domain()
Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33)
sage: phi.codomain()
Abelian variety J0(33) of dimension 3
sage: phi.matrix()
[ 1  1 -2  0  2 -1]
[ 0  3 -2 -1  2  0]
```

phi is of course injective

```
sage: phi.kernel()
(Finite subgroup with invariants [] over QQ of Simple abelian subvariety 11a(1,33) of dimens
 Abelian subvariety of dimension 0 of J0(33))
```

This is the same as the basis matrix for the lattice corresponding to self:

```
sage: A.lattice()
Free module of degree 6 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  1 -2  0  2 -1]
[ 0  3 -2 -1  2  0]
```

We compute a non-injecture map to an ambient space:

```
sage: Q,pi = J0(33)/A
sage: phi = Q.ambient_morphism()
sage: phi.matrix()
[  1   4   1   9  -1  -1]
[  0  15   0   0  30 -75]
[  0   0   5  10  -5  15]
[  0   0   0  15 -15  30]
sage: phi.kernel()[0]
Finite subgroup with invariants [5, 15, 15] over QQ of Abelian variety factor of dimension 2
```

ambient_variety()

Return the ambient modular abelian variety that contains this abelian variety. The ambient variety is always a product of Jacobians of modular curves.

OUTPUT: abelian variety

EXAMPLES:

```
sage: A = J0(33)[0]; A
Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33)
sage: A.ambient_variety()
Abelian variety J0(33) of dimension 3
```

base_extend(*K*)

EXAMPLES:

```
sage: A = J0(37); A
Abelian variety J0(37) of dimension 2
sage: A.base_extend(QQbar)
Abelian variety J0(37) over Algebraic Field of dimension 2
sage: A.base_extend(GF(7))
Abelian variety J0(37) over Finite Field of size 7 of dimension 2
```

base_field()

Synonym for `self.base_ring()`.

EXAMPLES:

```
sage: J0(11).base_field()
Rational Field
```

category()

Return the category of modular abelian varieties that contains this modular abelian variety.

EXAMPLES:

```
sage: J0(23).category()
Category of modular abelian varieties over Rational Field
```

change_ring(*R*)

Change the base ring of this modular abelian variety.

EXAMPLES:

```
sage: A = J0(23)
sage: A.change_ring(QQ)
Abelian variety J0(23) of dimension 2
```

complement(*A=None*)

Return a complement of this abelian variety.

INPUT:

- `A` - (default: None); if given, A must be an abelian variety that contains self, in which case the complement of self is taken inside A. Otherwise the complement is taken in the ambient product Jacobian.

OUTPUT: abelian variety

EXAMPLES:

```
sage: a,b,c = J0(33)
sage: (a+b).complement()
Simple abelian subvariety 33a(1,33) of dimension 1 of J0(33)
sage: (a+b).complement() == c
True
sage: a.complement(a+b)
Abelian subvariety of dimension 1 of J0(33)
```

cuspidal_subgroup()

Return the cuspidal subgroup of this modular abelian variety. This is the subgroup generated by rational cusps.

EXAMPLES:

```
sage: J = J0(54)
sage: C = J.cuspidal_subgroup()
sage: C.gens()
[[(1/3, 0, 0, 0, 0, 1/3, 0, 2/3)], [(0, 1/3, 0, 0, 0, 2/3, 0, 1/3)], [(0, 0, 1/9, 1/9, 1/9,
```

```
sage: C.invariants()
[3, 3, 3, 3, 3, 9]
sage: J1(13).cuspidal_subgroup()
Finite subgroup with invariants [19, 19] over QQ of Abelian variety J1(13) of dimension 2
sage: A = J0(33)[0]
sage: A.cuspidal_subgroup()
Finite subgroup with invariants [5] over QQ of Simple abelian subvariety 11a(1,33) of dimens
```

decomposition(*simple=True, bound=None*)

Return a sequence of abelian subvarieties of self that are all simple, have finite intersection and sum to self.

INPUT: simple- bool (default: True) if True, all factors are simple. If False, each factor returned is isogenous to a power of a simple and the simples in each factor are distinct.

- `bound` - int (default: None) if given, only use Hecke operators up to this bound when decomposing. This can give wrong answers, so use with caution!

EXAMPLES:

```
sage: m = ModularSymbols(11).cuspidal_submodule()
sage: d1 = m.degeneracy_map(33,1).matrix(); d3=m.degeneracy_map(33,3).matrix()
sage: w = ModularSymbols(33).submodule((d1 + d3).image(), check=False)
sage: A = w.abelian_variety(); A
Abelian subvariety of dimension 1 of J0(33)
sage: D = A.decomposition(); D
[
Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33)
]
sage: D[0] == A
True
sage: B = A + J0(33)[0]; B
Abelian subvariety of dimension 2 of J0(33)
sage: dd = B.decomposition(simple=False); dd
[
Abelian subvariety of dimension 2 of J0(33)
]
sage: dd[0] == B
True
sage: dd = B.decomposition(); dd
[
Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33),
Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33)
]
sage: sum(dd) == B
True
```

We decompose a product of two Jacobians:

```
sage: (J0(33) * J0(11)).decomposition()
[
Simple abelian subvariety 11a(1,11) of dimension 1 of J0(33) x J0(11),
Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33) x J0(11),
Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33) x J0(11),
Simple abelian subvariety 33a(1,33) of dimension 1 of J0(33) x J0(11)
]
```

degen_t(*none_if_not_known=False*)

If this abelian variety is obtained via decomposition then it gets labeled with the newform label along with some information about degeneracy maps. In particular, the label ends in a pair (t, N), where N is the ambient level and t is an integer that divides the quotient of N by the newform level. This function returns

the tuple (t, N), or raises a ValueError if self isn't simple.

Note: It need not be the case that self is literally equal to the image of the newform abelian variety under the t^{th} degeneracy map. See the documentation for the label method for more details.

INPUT:

- `none_if_not_known` - (default: False) - if True, return None instead of attempting to compute the degen map's t, if it isn't known. This None result is not cached.

OUTPUT: a pair (integer, integer)

EXAMPLES:

```
sage: D = J0(33).decomposition(); D
[
Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33),
Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33),
Simple abelian subvariety 33a(1,33) of dimension 1 of J0(33)
]
sage: D[0].degen_t()
(1, 33)
sage: D[1].degen_t()
(3, 33)
sage: D[2].degen_t()
(1, 33)
sage: J0(33).degen_t()
...
ValueError: self must be simple
```

degeneracy_map (*M_ls, t_ls*)

Return the degeneracy map with domain self and given level/parameter. If self.ambient_variety() is a product of Jacobians (as opposed to a single Jacobian), then one can provide a list of new levels and parameters, corresponding to the ambient Jacobians in order. (See the examples below.)

INPUT:

- `M, t` - integers level and t, or
- `Mlist, tlist` - if self is in a nontrivial product ambient Jacobian, input consists of a list of levels and corresponding list of t's.

OUTPUT: a degeneracy map

EXAMPLES: We make several degenerancy maps related to $J_0(11)$ and $J_0(33)$ and compute their matrices.

```
sage: d1 = J0(11).degeneracy_map(33, 1); d1
Degeneracy map from Abelian variety J0(11) of dimension 1 to Abelian variety J0(33) of dimen
sage: d1.matrix()
[ 0 -3  2  1 -2  0]
[ 1 -2  0  1  0 -1]
sage: d2 = J0(11).degeneracy_map(33, 3); d2
Degeneracy map from Abelian variety J0(11) of dimension 1 to Abelian variety J0(33) of dimen
sage: d2.matrix()
[-1  0  0  0  1 -2]
[-1 -1  1 -1  1  0]
sage: d3 = J0(33).degeneracy_map(11, 1); d3
Degeneracy map from Abelian variety J0(33) of dimension 3 to Abelian variety J0(11) of dimen
```

He we verify that first mapping from level 11 to level 33, then back is multiplication by 4:

```
sage: d1.matrix() * d3.matrix()
[4 0]
[0 4]
```

We compute a more complciated degeneracy map involving nontrivial product ambient Jacobians; note that this is just the block direct sum of the two matrices at the beginning of this example:

```
sage: d = (J0(11)*J0(11)).degeneracy_map([33,33], [1,3]); d
Degeneracy map from Abelian variety J0(11) x J0(11) of dimension 2 to Abelian variety J0(33)
sage: d.matrix()
[ 0 -3  2  1 -2  0  0  0  0  0  0  0]
[ 1 -2  0  1  0 -1  0  0  0  0  0  0]
[ 0  0  0  0  0  0 -1  0  0  0  1 -2]
[ 0  0  0  0  0  0 -1 -1  1 -1  1  0]
```

degree()

Return the degree of this abelian variety, which is the dimension of the ambient Jacobian product.

EXAMPLES:

```
sage: A = J0(23)
sage: A.dimension()
2
```

dimension()

Return the dimension of this abelian variety.

EXAMPLES:

```
sage: A = J0(23)
sage: A.dimension()
2
```

direct_product(*other*)

Compute the direct product of self and other.

INPUT:

- `self, other` - modular abelian varieties

OUTPUT: abelian variety

EXAMPLES:

```
sage: J0(11).direct_product(J1(13))
Abelian variety J0(11) x J1(13) of dimension 3
sage: A = J0(33)[0].direct_product(J0(33)[1]); A
Abelian subvariety of dimension 2 of J0(33) x J0(33)
sage: A.lattice()
Free module of degree 12 and rank 4 over Integer Ring
Echelon basis matrix:
[ 1  1 -2  0  2 -1  0  0  0  0  0  0]
[ 0  3 -2 -1  2  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  1  0  0  0 -1  2]
[ 0  0  0  0  0  0  0  1 -1  1  0 -2]
```

dual()

Return the dual of this abelian variety.

OUTPUT: abelian variety

Warning: This is currently only implemented when self is an abelian subvariety of the ambient Jacobian product, and the complement of self in the ambient product Jacobian share no common factors. A more general implementation will require implementing computation of the intersection pairing on integral homology and the resulting Weil pairing on torsion.

EXAMPLES: We compute the dual of the elliptic curve newform abelian variety of level 33, and find the kernel of the modular map, which has structure $(\mathbf{Z}/3)^2$.

```
sage: A,B,C = J0(33)
sage: C
```

```
Simple abelian subvariety 33a(1,33) of dimension 1 of J0(33)
sage: Cd, f = C.dual()
sage: f.matrix()
[3 0]
[0 3]
sage: f.kernel()[0]
Finite subgroup with invariants [3, 3] over QQ of Simple abelian subvariety 33a(1,33) of dim
```

By a theorem the modular degree must thus be 3:

```
sage: E = EllipticCurve('33a')
sage: E.modular_degree()
3
```

Next we compute the dual of a 2-dimensional new simple abelian subvariety of $J_0(43)$.

```
sage: A = AbelianVariety('43b'); A
Newform abelian subvariety 43b of dimension 2 of J0(43)
sage: Ad, f = A.dual()
```

The kernel shows that the modular degree is 2:

```
sage: f.kernel()[0]
Finite subgroup with invariants [2, 2] over QQ of Newform abelian subvariety 43b of dimensio
```

Unfortunately, the dual is not implemented in general:

```
sage: A = J0(22)[0]; A
Simple abelian subvariety 11a(1,22) of dimension 1 of J0(22)
sage: A.dual()
...
NotImplementedError: dual not implemented unless complement shares no simple factors with se
```

endomorphism_ring()

Return the endomorphism ring of self.

OUTPUT: b = self.sturm_bound()

EXAMPLES: We compute a few endomorphism rings:

```
sage: J0(11).endomorphism_ring()
Endomorphism ring of Abelian variety J0(11) of dimension 1
sage: J0(37).endomorphism_ring()
Endomorphism ring of Abelian variety J0(37) of dimension 2
sage: J0(33)[2].endomorphism_ring()
Endomorphism ring of Simple abelian subvariety 33a(1,33) of dimension 1 of J0(33)
```

No real computation is done:

```
sage: J1(123456).endomorphism_ring()
Endomorphism ring of Abelian variety J1(123456) of dimension 423185857
```

finite_subgroup(*X, field_of_definition=None, check=True*)

Return a finite subgroup of this modular abelian variety.

INPUT:

- `X` - list of elements of other finite subgroups of this modular abelian variety or elements that coerce into the rational homology (viewed as a rational vector space); also X could be a finite subgroup itself that is contained in this abelian variety.
- `field_of_definition` - (default: None) field over which this group is defined. If None try to figure out the best base field.

OUTPUT: a finite subgroup of a modular abelian variety

EXAMPLES:

```
sage: J = J0(11)
sage: J.finite_subgroup([[1/5,0], [0,1/3]])
Finite subgroup with invariants [15] over QQbar of Abelian variety J0(11) of dimension 1

sage: J = J0(33); C = J[0].cuspidal_subgroup(); C
Finite subgroup with invariants [5] over QQ of Simple abelian subvariety 11a(1,33) of dimens
sage: J.finite_subgroup([[0,0,0,0,0,1/6]])
Finite subgroup with invariants [6] over QQbar of Abelian variety J0(33) of dimension 3
sage: J.finite_subgroup(C)
Finite subgroup with invariants [5] over QQ of Abelian variety J0(33) of dimension 3
```

free_module()

Synonym for `self.lattice()`.

OUTPUT: a free module over $\mathbf{Z}$

EXAMPLES:

```
sage: J0(37).free_module()
Ambient free module of rank 4 over the principal ideal domain Integer Ring
sage: J0(37)[0].free_module()
Free module of degree 4 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1 -1  1  0]
[ 0  0  2 -1]
```

groups()

Return an ordered tuple of the congruence subgroups that the ambient product Jacobian is attached to.

Every modular abelian variety is a finite quotient of an abelian subvariety of a product of modular Jacobians J_Γ. This function returns a tuple containing the groups Γ.

EXAMPLES:

```
sage: A = (J0(37) * J1(13))[0]; A
Simple abelian subvariety 13aG1(1,13) of dimension 2 of J0(37) x J1(13)
sage: A.groups()
(Congruence Subgroup Gamma0(37), Congruence Subgroup Gamma1(13))
```

hecke_operator(*n*)

Return the n^{th} Hecke operator on the modular abelian variety, if this makes sense [[elaborate]]. Otherwise raise a ValueError.

EXAMPLES: We compute T_2 on $J_0(37)$.

```
sage: t2 = J0(37).hecke_operator(2); t2
Hecke operator T_2 on Abelian variety J0(37) of dimension 2
sage: t2.charpoly().factor()
x * (x + 2)
sage: t2.index()
2
```

Note that there is no matrix associated to Hecke operators on modular abelian varieties. For a matrix, instead consider, e.g., the Hecke operator on integral or rational homology.

```
sage: t2.action_on_homology().matrix()
[-1  1  1 -1]
[ 1 -1  1  0]
[ 0  0 -2  1]
[ 0  0  0  0]
```

hecke_polynomial(*n, var='x'*)

Return the characteristic polynomial of the n^{th} Hecke operator T_n acting on self. Raises an ArithmeticError if self is not Hecke equivariant.

INPUT:

- n - integer ≥ 1
- var - string (default: 'x'); valid variable name

EXAMPLES:

```
sage: J0(33).hecke_polynomial(2)
x^3 + 3*x^2 - 4
sage: f = J0(33).hecke_polynomial(2, 'y'); f
y^3 + 3*y^2 - 4
sage: f.parent()
Univariate Polynomial Ring in y over Rational Field
sage: J0(33)[2].hecke_polynomial(3)
x + 1
sage: J0(33)[0].hecke_polynomial(5)
x - 1
sage: J0(33)[0].hecke_polynomial(11)
x - 1
sage: J0(33)[0].hecke_polynomial(3)
...
ArithmeticError: subspace is not invariant under matrix
```

homology(*base_ring=Integer Ring*)

Return the homology of this modular abelian variety.

Warning: For efficiency reasons the basis of the integral homology need not be the same as the basis for the rational homology.

EXAMPLES:

```
sage: J0(389).homology(GF(7))
Homology with coefficients in Finite Field of size 7 of Abelian variety J0(389) of dimension
sage: J0(389).homology(QQ)
Rational Homology of Abelian variety J0(389) of dimension 32
sage: J0(389).homology(ZZ)
Integral Homology of Abelian variety J0(389) of dimension 32
```

in_same_ambient_variety(*other*)

Return True if self and other are abelian subvarieties of the same ambient product Jacobian.

EXAMPLES:

```
sage: A,B,C = J0(33)
sage: A.in_same_ambient_variety(B)
True
sage: A.in_same_ambient_variety(J0(11))
False
```

integral_homology()

Return the integral homology of this modular abelian variety.

EXAMPLES:

```
sage: H = J0(43).integral_homology(); H
Integral Homology of Abelian variety J0(43) of dimension 3
sage: H.rank()
6
sage: H = J1(17).integral_homology(); H
Integral Homology of Abelian variety J1(17) of dimension 5
sage: H.rank()
10
```

If you just ask for the rank of the homology, no serious calculations are done, so the following is fast:

```
sage: H = J0(50000).integral_homology(); H
Integral Homology of Abelian variety J0(50000) of dimension 7351
sage: H.rank()
14702
```

A product:

```
sage: H = (J0(11) * J1(13)).integral_homology()
sage: H.hecke_operator(2)
Hecke operator T_2 on Integral Homology of Abelian variety J0(11) x J1(13) of dimension 3
sage: H.hecke_operator(2).matrix()
[-2  0  0  0  0  0]
[ 0 -2  0  0  0  0]
[ 0  0 -2  0 -1  1]
[ 0  0  1 -1  0 -1]
[ 0  0  1  1 -2  0]
[ 0  0  0  1 -1 -1]
```

intersection(*other*)

Returns the intersection of self and other inside a common ambient Jacobian product.

INPUT:

- `other` - a modular abelian variety or a finite group

OUTPUT: If other is a modular abelian variety:

- `G` - finite subgroup of self
- `A` - abelian variety (identity component of intersection) If other is a finite group:
- `G` - a finite group

EXAMPLES: We intersect some abelian varieties with finite intersection.

```
sage: J = J0(37)
sage: J[0].intersection(J[1])
(Finite subgroup with invariants [2, 2] over QQ of Simple abelian subvariety 37a(1,37) of di

sage: D = list(J0(65)); D
[Simple abelian subvariety 65a(1,65) of dimension 1 of J0(65), Simple abelian subvariety 65b
sage: D[0].intersection(D[1])
(Finite subgroup with invariants [2] over QQ of Simple abelian subvariety 65a(1,65) of dimen
sage: (D[0]+D[1]).intersection(D[1]+D[2])
(Finite subgroup with invariants [2] over QQbar of Abelian subvariety of dimension 3 of J0(6

sage: J = J0(33)
sage: J[0].intersection(J[1])
(Finite subgroup with invariants [5] over QQ of Simple abelian subvariety 11a(1,33) of dimen
```

Next we intersect two abelian varieties with non-finite intersection:

```
sage: J = J0(67); D = J.decomposition(); D
[
Simple abelian subvariety 67a(1,67) of dimension 1 of J0(67),
Simple abelian subvariety 67b(1,67) of dimension 2 of J0(67),
Simple abelian subvariety 67c(1,67) of dimension 2 of J0(67)
]
sage: (D[0] + D[1]).intersection(D[1] + D[2])
(Finite subgroup with invariants [5, 10] over QQbar of Abelian subvariety of dimension 3 of
```

is_ambient()

Return True if self equals the ambient product Jacobian.

OUTPUT: bool

EXAMPLES:

```
sage: A,B,C = J0(33)
sage: A.is_ambient()
False
sage: J0(33).is_ambient()
True
sage: (A+B).is_ambient()
False
sage: (A+B+C).is_ambient()
True
```

is_hecke_stable()

Return True if self is stable under the Hecke operators of its ambient Jacobian.

OUTPUT: bool

EXAMPLES:

```
sage: J0(11).is_hecke_stable()
True
sage: J0(33)[2].is_hecke_stable()
True
sage: J0(33)[0].is_hecke_stable()
False
sage: (J0(33)[0] + J0(33)[1]).is_hecke_stable()
True
```

is_simple(*none_if_not_known=False*)

Return whether or not this modular abelian variety is simple, i.e., has no proper nonzero abelian subvarieties.

INPUT:

- `none_if_not_known` - bool (default: False); if True then this function may return None instead of True of False if we don't already know whether or not self is simple.

EXAMPLES:

```
sage: J0(5).is_simple(none_if_not_known=True) is None  # this may fail if J0(5) comes up els
True
sage: J0(33).is_simple()
False
sage: J0(33).is_simple(none_if_not_known=True)
False
sage: J0(33)[1].is_simple()
True
sage: J1(17).is_simple()
False
```

is_subvariety(*other*)

Return True if self is a subvariety of other as they sit in a common ambient modular Jacobian. In particular, this function will only return True if self and other have exactly the same ambient Jacobians.

EXAMPLES:

```
sage: J = J0(37); J
Abelian variety J0(37) of dimension 2
sage: A = J[0]; A
Simple abelian subvariety 37a(1,37) of dimension 1 of J0(37)
sage: A.is_subvariety(A)
True
sage: A.is_subvariety(J)
True
```

is_subvariety_of_ambient_jacobian()

Return True if self is (presented as) a subvariety of the ambient product Jacobian.

Every abelian variety in Sage is a quotient of a subvariety of an ambient Jacobian product by a finite subgroup.

EXAMPLES:

```
sage: J0(33).is_subvariety_of_ambient_jacobian()
True
sage: A = J0(33)[0]; A
Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33)
sage: A.is_subvariety_of_ambient_jacobian()
True
sage: B, phi = A / A.torsion_subgroup(2)
sage: B
Abelian variety factor of dimension 1 of J0(33)
sage: phi.matrix()
[2 0]
[0 2]
sage: B.is_subvariety_of_ambient_jacobian()
False
```

isogeny_number(*none_if_not_known=False*)

Return the number (starting at 0) of the isogeny class of new simple abelian varieties that self is in. If self is not simple, raises a ValueError exception.

INPUT:

- `none_if_not_known` - bool (default: False); if True then this function may return None instead of True of False if we don't already know the isogeny number of self.

EXAMPLES: We test the none_if_not_known flag first:

```
sage: J0(33).isogeny_number(none_if_not_known=True) is None
True
```

Of course, $J_0(33)$ is not simple, so this function raises a ValueError:

```
sage: J0(33).isogeny_number()
...
ValueError: self must be simple
```

Each simple factor has isogeny number 1, since that's the number at which the factor is new.

```
sage: J0(33)[1].isogeny_number()
0
sage: J0(33)[2].isogeny_number()
0
```

Next consider $J_0(37)$ where there are two distinct newform factors:

```
sage: J0(37)[1].isogeny_number()
1
```

label()

Return the label associated to this modular abelian variety.

The format of the label is [level][isogeny class][group](t, ambient level)

If this abelian variety B has the above label, this implies only that B is isogenous to the newform abelan variety A_f associated to the newform with label [level][isogeny class][group]. The [group] is empty for $\Gamma_0(N)$, is G1 for $\Gamma_1(N)$ and is GH[...] for $\Gamma_H(N)$.

Warning: The sum of $\delta_s(A_f)$ for all $s \mid t$ contains A, but no sum for a proper divisor of t contains A. It need *not* be the case that B is equal to $\delta_t(A_f)$!!!

OUTPUT: string

EXAMPLES:

```
sage: J0(11).label()
'11a(1,11)'
sage: J0(11)[0].label()
'11a(1,11)'
sage: J0(33)[2].label()
'33a(1,33)'
sage: J0(22).label()
...
ValueError: self must be simple
```

We illustrate that self need not equal $\delta_t(A_f)$:

```
sage: J = J0(11); phi = J.degeneracy_map(33, 1) + J.degeneracy_map(33,3)
sage: B = phi.image(); B
Abelian subvariety of dimension 1 of J0(33)
sage: B.decomposition()
[
Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33)
]
sage: C = J.degeneracy_map(33,3).image(); C
Abelian subvariety of dimension 1 of J0(33)
sage: C == B
False
```

lattice()

Return lattice in ambient cuspidal modular symbols product that defines this modular abelian variety.

This must be defined in each derived class.

OUTPUT: a free module over $\mathbf{Z}$

EXAMPLES:

```
sage: A = sage.modular.abvar.abvar.ModularAbelianVariety_abstract((Gamma0(37),), QQ)
sage: A
...
NotImplementedError: BUG -- lattice method must be defined in derived class
```

level()

Return the level of this modular abelian variety, which is an integer N (usually minimal) such that this modular abelian variety is a quotient of $J_1(N)$. In the case that the ambient variety of self is a product of Jacobians, return the LCM of their levels.

EXAMPLES:

```
sage: J1(5077).level()
5077
sage: JH(389,[4]).level()
389
sage: (J0(11)*J0(17)).level()
187
```

lseries()

Return the complex L-series of this modular abelian variety.

EXAMPLES:

```
sage: A = J0(37)
sage: A.lseries()
Complex L-series attached to Abelian variety J0(37) of dimension 2
```

modular_degree()
Return the modular degree of this abelian variety, which is the square root of the degree of the modular kernel.

EXAMPLES:

```
sage: A = AbelianVariety('37a')
sage: A.modular_degree()
2
```

modular_kernel()
Return the modular kernel of this abelian variety, which is the kernel of the canonical polarization of self.

EXAMPLES:

```
sage: A = AbelianVariety('33a'); A
Newform abelian subvariety 33a of dimension 1 of J0(33)
sage: A.modular_kernel()
Finite subgroup with invariants [3, 3] over QQ of Newform abelian subvariety 33a of dimensio
```

newform_label()
Return the label [level][isogeny class][group] of the newform f such that this abelian variety is isogenous to the newform abelian variety A_f. If this abelian variety is not simple, raise a ValueError.

OUTPUT: string

EXAMPLES:

```
sage: J0(11).newform_label()
'11a'
sage: J0(33)[2].newform_label()
'33a'
```

The following fails since $J_0(33)$ is not simple:

```
sage: J0(33).newform_label()
...
ValueError: self must be simple
```

newform_level(*none_if_not_known=False*)
Write self as a product (up to isogeny) of newform abelian varieties A_f. Then this function return the least common multiple of the levels of the newforms f, along with the corresponding group or list of groups (the groups do not appear with multiplicity).

INPUT:

- `none_if_not_known` - (default: False) if True, return None instead of attempting to compute the newform level, if it isn't already known. This None result is not cached.

OUTPUT: integer group or list of distinct groups

EXAMPLES:

```
sage: J0(33)[0].newform_level()
(11, Congruence Subgroup Gamma0(33))
sage: J0(33)[0].newform_level(none_if_not_known=True)
(11, Congruence Subgroup Gamma0(33))
```

Here there are multiple groups since there are in fact multiple newforms:

```
sage: (J0(11) * J1(13)).newform_level()
(143, [Congruence Subgroup Gamma0(11), Congruence Subgroup Gamma1(13)])
```

padic_lseries(p)
Return the p-adic L-series of this modular abelian variety.

EXAMPLES:

```
sage: A = J0(37)
sage: A.padic_lseries(7)
7-adic L-series attached to Abelian variety J0(37) of dimension 2
```

project_to_factor(*n*)

If self is an ambient product of Jacobians, return a projection from self to the nth such Jacobian.

EXAMPLES:

```
sage: J = J0(33)
sage: J.project_to_factor(0)
Abelian variety endomorphism of Abelian variety J0(33) of dimension 3

sage: J = J0(33) * J0(37) * J0(11)
sage: J.project_to_factor(2)
Abelian variety morphism:
  From: Abelian variety J0(33) x J0(37) x J0(11) of dimension 6
  To:   Abelian variety J0(11) of dimension 1
sage: J.project_to_factor(2).matrix()
[0 0]
[0 0]
[0 0]
[0 0]
[0 0]
[0 0]
[0 0]
[0 0]
[0 0]
[0 0]
[1 0]
[0 1]
```

projection(*A, check=True*)

Given an abelian subvariety A of self, return a projection morphism from self to A. Note that this morphism need not be unique.

INPUT:

- A - an abelian variety

OUTPUT: a morphism

EXAMPLES:

```
sage: a,b,c = J0(33)
sage: pi = J0(33).projection(a); pi.matrix()
[ 3 -2]
[-5  5]
[-4  1]
[ 3 -2]
[ 5  0]
[ 1  1]
sage: pi = (a+b).projection(a); pi.matrix()
[ 0  0]
[-3  2]
[-4  1]
[-1 -1]
sage: pi = a.projection(a); pi.matrix()
[1 0]
[0 1]
```

We project onto a factor in a product of two Jacobians:

```
sage: A = J0(11)*J0(11); A
Abelian variety J0(11) x J0(11) of dimension 2
sage: A[0]
Simple abelian subvariety 11a(1,11) of dimension 1 of J0(11) x J0(11)
sage: A.projection(A[0])
Abelian variety morphism:
  From: Abelian variety J0(11) x J0(11) of dimension 2
  To:   Simple abelian subvariety 11a(1,11) of dimension 1 of J0(11) x J0(11)
sage: A.projection(A[0]).matrix()
[0 0]
[0 0]
[1 0]
[0 1]
sage: A.projection(A[1]).matrix()
[1 0]
[0 1]
[0 0]
[0 0]
```

qbar_torsion_subgroup()

Return the group of all points of finite order in the algebraic closure of this abelian variety.

EXAMPLES:

```
sage: T = J0(33).qbar_torsion_subgroup(); T
Group of all torsion points in QQbar on Abelian variety J0(33) of dimension 3
```

The field of definition is the same as the base field of the abelian variety.

```
sage: T.field_of_definition()
Rational Field
```

On the other hand, T is a module over $\mathbf{Z}$.

```
sage: T.base_ring()
Integer Ring
```

quotient(*other*)

Compute the quotient of self and other, where other is either an abelian subvariety of self or a finite subgroup of self.

INPUT:

- `other` - a finite subgroup or subvariety

OUTPUT: a pair (A, phi) with phi the quotient map from self to A

EXAMPLES: We quotient $J_0(33)$ out by an abelian subvariety:

```
sage: Q, f = J0(33).quotient(J0(33)[0])
sage: Q
Abelian variety factor of dimension 2 of J0(33)
sage: f
Abelian variety morphism:
  From: Abelian variety J0(33) of dimension 3
  To:   Abelian variety factor of dimension 2 of J0(33)
```

We quotient $J_0(33)$ by the cuspidal subgroup:

```
sage: C = J0(33).cuspidal_subgroup()
sage: Q, f = J0(33).quotient(C)
sage: Q
Abelian variety factor of dimension 3 of J0(33)
sage: f.kernel()[0]
```

```
Finite subgroup with invariants [10, 10] over QQ of Abelian variety J0(33) of dimension 3
sage: C
Finite subgroup with invariants [10, 10] over QQ of Abelian variety J0(33) of dimension 3
sage: J0(11).direct_product(J1(13))
Abelian variety J0(11) x J1(13) of dimension 3
```

rank()

Return the rank of the underlying lattice of self.

EXAMPLES:

```
sage: J = J0(33)
sage: J.rank()
6
sage: J[1]
Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33)
sage: (J[1] * J[1]).rank()
4
```

rational_cusp_subgroup()

Return the subgroup of this modular abelian variety generated by rational cusps.

This is a subgroup of the group of rational points in the cuspidal subgroup.

Warning: This is only currently implemented for $\Gamma_0(N)$.

EXAMPLES:

```
sage: J = J0(54)
sage: CQ = J.rational_cusp_subgroup(); CQ
Finite subgroup with invariants [3, 3, 9] over QQ of Abelian variety J0(54) of dimension 4
sage: CQ.gens()
[[(1/3, 0, 0, 1/3, 2/3, 1/3, 0, 1/3)], [(0, 0, 1/9, 1/9, 7/9, 7/9, 1/9, 8/9)], [(0, 0, 0, 0,
sage: factor(CQ.order())
3^4
sage: CQ.invariants()
[3, 3, 9]
```

In this example the rational cuspidal subgroup and the cuspidal subgroup differ by a lot.

```
sage: J = J0(49)
sage: J.cuspidal_subgroup()
Finite subgroup with invariants [2, 14] over QQ of Abelian variety J0(49) of dimension 1
sage: J.rational_cusp_subgroup()
Finite subgroup with invariants [2] over QQ of Abelian variety J0(49) of dimension 1
```

Note that computation of the rational cusp subgroup isn't implemented for Γ_1.

```
sage: J = J1(13)
sage: J.cuspidal_subgroup()
Finite subgroup with invariants [19, 19] over QQ of Abelian variety J1(13) of dimension 2
sage: J.rational_cusp_subgroup()
...
NotImplementedError: computation of rational cusps only implemented in Gamma0 case.
```

rational_homology()

Return the rational homology of this modular abelian variety.

EXAMPLES:

```
sage: H = J0(37).rational_homology(); H
Rational Homology of Abelian variety J0(37) of dimension 2
sage: H.rank()
```

```
4
sage: H.base_ring()
Rational Field
sage: H = J1(17).rational_homology(); H
Rational Homology of Abelian variety J1(17) of dimension 5
sage: H.rank()
10
sage: H.base_ring()
Rational Field
```

rational_torsion_subgroup()

Return the maximal torsion subgroup of self defined over QQ.

EXAMPLES:

```
sage: J = J0(33)
sage: A = J.new_subvariety()
sage: A
Abelian subvariety of dimension 1 of J0(33)
sage: t = A.rational_torsion_subgroup()
sage: t.multiple_of_order()
4
sage: t.divisor_of_order()
4
sage: t.order()
4
sage: t.gens()
[[(1/2, 0, 0, -1/2, 0, 0)], [(0, 0, 1/2, 0, 1/2, -1/2)]]
sage: t
Torsion subgroup of Abelian subvariety of dimension 1 of J0(33)
```

sturm_bound()

Return a bound B such that all Hecke operators T_n for $n \leq B$ generate the Hecke algebra.

OUTPUT: integer

EXAMPLES:

```
sage: J0(11).sturm_bound()
2
sage: J0(33).sturm_bound()
8
sage: J1(17).sturm_bound()
48
sage: J1(123456).sturm_bound()
1693483008
sage: JH(37,[2,3]).sturm_bound()
7
sage: J1(37).sturm_bound()
228
```

torsion_subgroup(n)

If n is an integer, return the subgroup of points of order n. Return the n-torsion subgroup of elements of order dividing n of this modular abelian variety A, i.e., the group $A[n]$.

EXAMPLES:

```
sage: J1(13).torsion_subgroup(19)
Finite subgroup with invariants [19, 19, 19, 19] over QQ of Abelian variety J1(13) of dimens
```

```
sage: A = J0(23)
sage: G = A.torsion_subgroup(5); G
```

```
Finite subgroup with invariants [5, 5, 5, 5] over QQ of Abelian variety J0(23) of dimension
sage: G.order()
625
sage: G.gens()
[[(1/5, 0, 0, 0)], [(0, 1/5, 0, 0)], [(0, 0, 1/5, 0)], [(0, 0, 0, 1/5)]]
sage: A = J0(23)
sage: A.torsion_subgroup(2).order()
16
```

vector_space()

Return vector space corresponding to the modular abelian variety.

This is the lattice tensored with $\mathbf{Q}$.

EXAMPLES:

```
sage: J0(37).vector_space()
Vector space of dimension 4 over Rational Field
sage: J0(37)[0].vector_space()
Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[   1   -1    0  1/2]
[   0    0    1 -1/2]
```

zero_subgroup()

Return the zero subgroup of this modular abelian variety, as a finite group.

EXAMPLES:

```
sage: A =J0(54); G = A.zero_subgroup(); G
Finite subgroup with invariants [] over QQ of Abelian variety J0(54) of dimension 4
sage: G.is_subgroup(A)
True
```

zero_subvariety()

Return the zero subvariety of self.

EXAMPLES:

```
sage: J = J0(37)
sage: J.zero_subvariety()
Simple abelian subvariety of dimension 0 of J0(37)
sage: J.zero_subvariety().level()
37
sage: J.zero_subvariety().newform_level()
(1, [])
```

class **ModularAbelianVariety_modsym**(*modsym, lattice=None, newform_level=None, is_simple=None, isogeny_number=None, number=None, check=True*)

brandt_module(*p*)

Return the Brandt module at p that corresponds to self. This is the factor of the vector space on the ideal class set in an order of level N in the quaternion algebra ramified at p and infinity.

INPUT: • p – prime that exactly divides the level

OUTPUT: • Brandt module space that corresponds to self.

EXAMPLES:

```
sage: J0(43)[1].brandt_module(43)
Subspace of dimension 2 of Brandt module of dimension 4 of level 43 of weight 2 over Rationa
sage: J0(43)[1].brandt_module(43).basis()
((1, 0, -1/2, -1/2), (0, 1, -1/2, -1/2))
sage: J0(43)[0].brandt_module(43).basis()
```

```
((0, 0, 1, -1),)
sage: J0(35)[0].brandt_module(5).basis()
((1, 0, -1, 0),)
sage: J0(35)[0].brandt_module(7).basis()
((1, -1, 1, -1),)
```

component_group_order(*p*)

Return the order of the component group of the special fiber at p of the Neron model of self.

NOTE: For bad primes, this is only implemented when the group if Gamma0 and p exactly divides the level.

NOTE: the input abelian variety must be simple

ALGORITHM: See "Component Groups of Quotients of J0(N)" by Kohel and Stein. That paper is about optimal quotients; however, section 4.1 of Conrad-Stein "Component Groups of Purely Toric Quotients", one sees that the component group of an optimal quotient is the same as the component group of its dual (which is the subvariety).

INPUT: • p – a prime number

OUTPUT: • Integer

EXAMPLES:

```
sage: A = J0(37)[1]
sage: A.component_group_order(37)
3
sage: A = J0(43)[1]
sage: A.component_group_order(37)
1
sage: A.component_group_order(43)
7
sage: A = J0(23)[0]
sage: A.component_group_order(23)
11
```

tamagawa_number(*p*)

Return the Tamagawa number of this abelianv variety at p.

NOTE: For bad primes, this is only implemented when the group if Gamma0 and p exactly divides the level and Atkin-Lehner acts diagonally on this abelian variety (e.g., if this variety is new and simple). See the self.component_group command for more information.

NOTE: the input abelian variety must be simple

In cases where this function doesn't work, consider using the self.tamagawa_number_bounds functions.

INPUT: • p – a prime number

OUTPUT: • Integer

EXAMPLES:

```
sage: A = J0(37)[1]
sage: A.tamagawa_number(37)
3
sage: A = J0(43)[1]
sage: A.tamagawa_number(37)
1
sage: A.tamagawa_number(43)
7
sage: A = J0(23)[0]
sage: A.tamagawa_number(23)
11
```

tamagawa_number_bounds(*p*)

Return a divisor and multiple of the Tamagawa number of self at p.

NOTE: the input abelian variety must be simple

INPUT: • p – a prime number

OUTPUT: • div – integer; divisor of Tamagawa number at p
- mul – integer; multiple of Tamagawa number at p
- mul_primes – tuple; in case mul==0, a list of all primes that can possibly divide the Tamagawa number at p.

EXAMPLES:

```
sage: A = J0(63).new_subvariety()[1]; A
Simple abelian subvariety 63b(1,63) of dimension 2 of J0(63)
sage: A.tamagawa_number_bounds(7)
(3, 3, ())
sage: A.tamagawa_number_bounds(3)
(1, 0, (2, 3, 5))
```

class ModularAbelianVariety_modsym_abstract(*groups, base_field, is_simple=None, newform_level=None, isogeny_number=None, number=None, check=True*)

decomposition(*simple=True, bound=None*)

Decompose this modular abelian variety as a product of abelian subvarieties, up to isogeny.

INPUT: simple- bool (default: True) if True, all factors are simple. If False, each factor returned is isogenous to a power of a simple and the simples in each factor are distinct.

•`bound` - int (default: None) if given, only use Hecke operators up to this bound when decomposing. This can give wrong answers, so use with caution!

EXAMPLES:

```
sage: J = J0(33)
sage: J.decomposition()
[
Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33),
Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33),
Simple abelian subvariety 33a(1,33) of dimension 1 of J0(33)
]
sage: J1(17).decomposition()
[
Simple abelian subvariety 17aG1(1,17) of dimension 1 of J1(17),
Simple abelian subvariety 17bG1(1,17) of dimension 4 of J1(17)
]
```

dimension()

Return the dimension of this modular abelian variety.

EXAMPLES:

```
sage: J0(37)[0].dimension()
1
sage: J0(43)[1].dimension()
2
sage: J1(17)[1].dimension()
4
```

group()

Return the congruence subgroup associated that this modular abelian variety is associated to.

EXAMPLES:

```
sage: J0(13).group()
Congruence Subgroup Gamma0(13)
```

```
sage: J1(997).group()
Congruence Subgroup Gamma1(997)
sage: JH(37,[3]).group()
Congruence Subgroup Gamma_H(37) with H generated by [3]
sage: J0(37)[1].groups()
(Congruence Subgroup Gamma0(37),)
```

groups()

Return the tuple of groups associated to the modular symbols abelian variety. This is always a 1-tuple.

OUTPUT: tuple

EXAMPLES:

```
sage: A = ModularSymbols(33).cuspidal_submodule().abelian_variety(); A
Abelian variety J0(33) of dimension 3
sage: A.groups()
(Congruence Subgroup Gamma0(33),)
sage: type(A)
<class 'sage.modular.abvar.abvar.ModularAbelianVariety_modsym'>
```

is_ambient()

Return True if this abelian variety attached to a modular symbols space space is attached to the cuspidal subspace of the ambient modular symbols space.

OUTPUT: bool

EXAMPLES:

```
sage: A = ModularSymbols(43).cuspidal_subspace().abelian_variety(); A
Abelian variety J0(43) of dimension 3
sage: A.is_ambient()
True
sage: type(A)
<class 'sage.modular.abvar.abvar.ModularAbelianVariety_modsym'>
sage: A = ModularSymbols(43).cuspidal_subspace()[1].abelian_variety(); A
Abelian subvariety of dimension 2 of J0(43)
sage: A.is_ambient()
False
```

is_subvariety(*other*)

Return True if self is a subvariety of other.

EXAMPLES:

```
sage: J = J0(37); J
Abelian variety J0(37) of dimension 2
sage: A = J[0]; A
Simple abelian subvariety 37a(1,37) of dimension 1 of J0(37)
sage: A.is_subvariety(J)
True
sage: A.is_subvariety(J0(11))
False
```

There may be a way to map A into $J_0(74)$, but A is not equipped with any special structure of an embedding.

```
sage: A.is_subvariety(J0(74))
False
```

Some ambient examples:

```
sage: J = J0(37)
sage: J.is_subvariety(J)
```

```
True
sage: J.is_subvariety(25)
False
```

More examples:

```
sage: A = J0(42); D = A.decomposition(); D
[
Simple abelian subvariety 14a(1,42) of dimension 1 of J0(42),
Simple abelian subvariety 14a(3,42) of dimension 1 of J0(42),
Simple abelian subvariety 21a(1,42) of dimension 1 of J0(42),
Simple abelian subvariety 21a(2,42) of dimension 1 of J0(42),
Simple abelian subvariety 42a(1,42) of dimension 1 of J0(42)
]
sage: D[0].is_subvariety(A)
True
sage: D[1].is_subvariety(D[0] + D[1])
True
sage: D[2].is_subvariety(D[0] + D[1])
False
```

lattice()

Return the lattice the defines this modular symbols modular abelian variety.

OUTPUT: a free $\mathbf{Z}$-module embedded in an ambient $\mathbf{Q}$-vector space

EXAMPLES:

```
sage: A = ModularSymbols(33).cuspidal_submodule()[0].abelian_variety(); A
Abelian subvariety of dimension 1 of J0(33)
sage: A.lattice()
Free module of degree 6 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0  0 -1  0  0]
[ 0  0  1  0  1 -1]
sage: type(A)
<class 'sage.modular.abvar.abvar.ModularAbelianVariety_modsym'>
```

modular_symbols(*sign=0*)

Return space of modular symbols (with given sign) associated to this modular abelian variety, if it can be found by cutting down using Hecke operators. Otherwise raise a RuntimeError exception.

EXAMPLES:

```
sage: A = J0(37)
sage: A.modular_symbols()
Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 5 for Gamma_0(
sage: A.modular_symbols(1)
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 3 for Gamma_0(
```

More examples:

```
sage: J0(11).modular_symbols()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 3 for Gamma_0(
sage: J0(11).modular_symbols(sign=1)
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 for Gamma_0(
sage: J0(11).modular_symbols(sign=0)
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 3 for Gamma_0(
sage: J0(11).modular_symbols(sign=-1)
Modular Symbols space of dimension 1 for Gamma_0(11) of weight 2 with sign -1 over Rational
```

Even more examples:

```
sage: A = J0(33)[1]; A
Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33)
sage: A.modular_symbols()
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 9 for Gamma_0(
```

It is not always possible to determine the sign subspaces:

```
sage: A.modular_symbols(1)
...
RuntimeError: unable to determine sign (=1) space of modular symbols

sage: A.modular_symbols(-1)
...
RuntimeError: unable to determine sign (=-1) space of modular symbols
```

new_subvariety(*p=None*)

Return the new or p-new subvariety of self.

INPUT:

- `self` - a modular abelian variety
- `p` - prime number or None (default); if p is a prime, return the p-new subvariety. Otherwise return the full new subvariety.

EXAMPLES:

```
sage: J0(33).new_subvariety()
Abelian subvariety of dimension 1 of J0(33)
sage: J0(100).new_subvariety()
Abelian subvariety of dimension 1 of J0(100)
sage: J1(13).new_subvariety()
Abelian variety J1(13) of dimension 2
```

old_subvariety(*p=None*)

Return the old or p-old abelian variety of self.

INPUT:

- `self` - a modular abelian variety
- `p` - prime number or None (default); if p is a prime, return the p-old subvariety. Otherwise return the full old subvariety.

EXAMPLES:

```
sage: J0(33).old_subvariety()
Abelian subvariety of dimension 2 of J0(33)
sage: J0(100).old_subvariety()
Abelian subvariety of dimension 6 of J0(100)
sage: J1(13).old_subvariety()
Abelian subvariety of dimension 0 of J1(13)
```

factor_modsym_space_new_factors(*M*)

Given an ambient modular symbols space, return complete factorization of it.

INPUT:

- `M` - modular symbols space

OUTPUT: list of decompositions corresponding to each new space.

EXAMPLES:

```
sage: M = ModularSymbols(33)
sage: sage.modular.abvar.abvar.factor_modsym_space_new_factors(M)
[[
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 3 for Gamma_0(11)
],
 [
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 9 for Gamma_0(33)
]]
```

factor_new_space(*M*)

Given a new space M of modular symbols, return the decomposition into simple of M under the Hecke operators.

INPUT:

- M - modular symbols space

OUTPUT: list of factors

EXAMPLES:

```
sage: M = ModularSymbols(37).cuspidal_subspace()
sage: sage.modular.abvar.abvar.factor_new_space(M)
[
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 5 for Gamma_0(37)
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 5 for Gamma_0(37)
]
```

is_ModularAbelianVariety(*x*)

Return True if x is a modular abelian variety.

INPUT:

- x - object

EXAMPLES:

```
sage: from sage.modular.abvar.abvar import is_ModularAbelianVariety
sage: is_ModularAbelianVariety(5)
False
sage: is_ModularAbelianVariety(J0(37))
True
```

Returning True is a statement about the data type not whether or not some abelian variety is modular:

```
sage: is_ModularAbelianVariety(EllipticCurve('37a'))
False
```

modsym_lattices(*M, factors*)

Append lattice information to the output of simple_factorization_of_modsym_space.

INPUT:

- M - modular symbols spaces
- factors - Sequence (simple_factorization_of_modsym_space)

OUTPUT: sequence with more information for each factor (the lattice)

EXAMPLES:

```
sage: M = ModularSymbols(33)
sage: factors = sage.modular.abvar.abvar.simple_factorization_of_modsym_space(M, simple=False)
sage: sage.modular.abvar.abvar.modsym_lattices(M, factors)
[
(11, 0, None, Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 9 fo
Echelon basis matrix:
[ 1  0  0  0 -1  2]
[ 0  1  0  0 -1  1]
[ 0  0  1  0 -2  2]
[ 0  0  0  1 -1 -1]),
(33, 0, None, Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 9 fo
Echelon basis matrix:
[ 1  0  0 -1  0  0]
[ 0  0  1  0  1 -1])
]
```

random_hecke_operator(*M, t=None, p=2*)

Return a random Hecke operator acting on M, got by adding to t a random multiple of T_p

INPUT:

- `M` - modular symbols space
- `t` - None or a Hecke operator
- `p` - a prime

OUTPUT: Hecke operator prime

EXAMPLES:

```
sage: M = ModularSymbols(11).cuspidal_subspace()
sage: t, p = sage.modular.abvar.abvar.random_hecke_operator(M)
sage: p
3
sage: t, p = sage.modular.abvar.abvar.random_hecke_operator(M, t, p)
sage: p
5
```

simple_factorization_of_modsym_space(*M, simple=True*)

Return factorization of M. If simple is False, return powers of simples.

INPUT:

- `M` - modular symbols space
- `simple` - bool (default: True)

OUTPUT: sequence

EXAMPLES:

```
sage: M = ModularSymbols(33)
sage: sage.modular.abvar.abvar.simple_factorization_of_modsym_space(M)
[
(11, 0, 1, Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 9 for G
(11, 0, 3, Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 9 for G
(33, 0, 1, Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 9 for G
]
sage: sage.modular.abvar.abvar.simple_factorization_of_modsym_space(M, simple=False)
[
(11, 0, None, Modular Symbols subspace of dimension 4 of Modular Symbols space of dimension 9 fo
```

```
(33, 0, None, Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 9 fc
]
```

sqrt_poly(*f*)

Return the square root of the polynomial f.

Note: At some point something like this should be a member of the polynomial class. For now this is just used internally by some charpoly functions above.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = (x-1)*(x+2)*(x^2 + 1/3*x + 5)
sage: f
x^4 + 4/3*x^3 + 10/3*x^2 + 13/3*x - 10
sage: sage.modular.abvar.abvar.sqrt_poly(f^2)
x^4 + 4/3*x^3 + 10/3*x^2 + 13/3*x - 10
sage: sage.modular.abvar.abvar.sqrt_poly(f)
...
ValueError: f must be a perfect square
sage: sage.modular.abvar.abvar.sqrt_poly(2*f^2)
...
ValueError: f must be monic
```

45.3 Ambient Jacobian Abelian Variety

TESTS:

```
sage: loads(dumps(J0(37))) == J0(37)
True
sage: loads(dumps(J1(13))) == J1(13)
True
```

ModAbVar_ambient_jacobian(*group*)

Return the ambient Jacobian attached to a given congruence subgroup.

The result is cached using a weakref. This function is called internally by modular abelian variety constructors.

INPUT:

- `group` - a congruence subgroup.

OUTPUT: a modular abelian variety attached

EXAMPLES:

```
sage: import sage.modular.abvar.abvar_ambient_jacobian as abvar_ambient_jacobian
sage: A = abvar_ambient_jacobian.ModAbVar_ambient_jacobian(Gamma0(11))
sage: A
Abelian variety J0(11) of dimension 1
sage: B = abvar_ambient_jacobian.ModAbVar_ambient_jacobian(Gamma0(11))
sage: A is B
True
```

You can get access to and/or clear the cache as follows:

```
sage: abvar_ambient_jacobian._cache = {}
sage: B = abvar_ambient_jacobian.ModAbVar_ambient_jacobian(Gamma0(11))
sage: A is B
False
```

class **ModAbVar_ambient_jacobian_class**(*group*)

An ambient Jacobian modular abelian variety attached to a congruence subgroup.

ambient_variety()

Return the ambient modular abelian variety that contains self. Since self is a Jacobian modular abelian variety, this is just self.

OUTPUT: abelian variety

EXAMPLES:

```
sage: A = J0(17)
sage: A.ambient_variety()
Abelian variety J0(17) of dimension 1
sage: A is A.ambient_variety()
True
```

decomposition(*simple=True, bound=None*)

Decompose this ambient Jacobian as a product of abelian subvarieties, up to isogeny.

EXAMPLES:

```
sage: J0(33).decomposition(simple=False)
[
Abelian subvariety of dimension 2 of J0(33),
Abelian subvariety of dimension 1 of J0(33)
]
sage: J0(33).decomposition(simple=False)[1].is_simple()
True
sage: J0(33).decomposition(simple=False)[0].is_simple()
False
sage: J0(33).decomposition(simple=False)
[
Abelian subvariety of dimension 2 of J0(33),
Simple abelian subvariety 33a(None,33) of dimension 1 of J0(33)
]
sage: J0(33).decomposition(simple=True)
[
Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33),
Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33),
Simple abelian subvariety 33a(1,33) of dimension 1 of J0(33)
]
```

degeneracy_map(*level, t=1, check=True*)

Return the t-th degeneracy map from self to J(level). Here t must be a divisor of either level/self.level() or self.level()/level.

INPUT:

- `level` - integer (multiple or divisor of level of self)
- `t` - divisor of quotient of level of self and level
- `check` - bool (default: True); if True do some checks on the input

OUTPUT: a morphism

EXAMPLES:

```
sage: J0(11).degeneracy_map(33)
Degeneracy map from Abelian variety J0(11) of dimension 1 to Abelian variety J0(33) of dimen
sage: J0(11).degeneracy_map(33).matrix()
[ 0 -3  2  1 -2  0]
[ 1 -2  0  1  0 -1]
sage: J0(11).degeneracy_map(33,3).matrix()
[-1  0  0  0  1 -2]
[-1 -1  1 -1  1  0]
sage: J0(33).degeneracy_map(11,1).matrix()
[ 0  1]
[ 0 -1]
[ 1 -1]
[ 0  1]
[-1  1]
[ 0  0]
sage: J0(11).degeneracy_map(33,1).matrix() * J0(33).degeneracy_map(11,1).matrix()
[4 0]
[0 4]
```

dimension()

Return the dimension of this modular abelian variety.

EXAMPLES:

```
sage: J0(2007).dimension()
221
sage: J1(13).dimension()
2
sage: J1(997).dimension()
40920
sage: J0(389).dimension()
32
sage: JH(389,[4]).dimension()
64
sage: J1(389).dimension()
6112
```

group()

Return the group that this Jacobian modular abelian variety is attached to.

EXAMPLES:

```
sage: J1(37).group()
Congruence Subgroup Gamma1(37)
sage: J0(5077).group()
Congruence Subgroup Gamma0(5077)
sage: J = GammaH(11,[3]).modular_abelian_variety(); J
Abelian variety JH(11,[3]) of dimension 1
sage: J.group()
Congruence Subgroup Gamma_H(11) with H generated by [3]
```

groups()

Return the tuple of congruence subgroups attached to this ambient Jacobian. This is always a tuple of length 1.

OUTPUT: tuple

EXAMPLES:

```
sage: J0(37).groups()
(Congruence Subgroup Gamma0(37),)
```

45.4 Finite subgroups of modular abelian varieties

Sage can compute with fairly general finite subgroups of modular abelian varieties. Elements of finite order are represented by equivalence classes of elements in $H_1(A, \mathbf{Q})$ modulo $H_1(A, \mathbf{Z})$. A finite subgroup can be defined by giving generators and via various other constructions. Given a finite subgroup, one can compute generators, as well as the structure as an abstract group. Arithmetic on subgroups is also supported, including adding two subgroups together, checking inclusion, etc.

TODO: Intersection, action of Hecke operators.

AUTHORS:

- William Stein (2007-03)

EXAMPLES:

```
sage: J = J0(33)
sage: C = J.cuspidal_subgroup()
sage: C
Finite subgroup with invariants [10, 10] over QQ of Abelian variety J0(33) of dimension 3
sage: C.order()
100
sage: C.gens()
[[(1/10, 0, 1/10, 1/10, 1/10, 3/10)], [(0, 1/5, 1/10, 0, 1/10, 9/10)], [(0, 0, 1/2, 0, 1/2, 1/2)]]
sage: C.0 + C.1
[(1/10, 1/5, 1/5, 1/10, 1/5, 6/5)]
sage: 10*(C.0 + C.1)
[(0, 0, 0, 0, 0, 0)]
sage: G = C.subgroup([C.0 + C.1]); G
Finite subgroup with invariants [10] over QQbar of Abelian variety J0(33) of dimension 3
sage: G.gens()
[[(1/10, 1/5, 1/5, 1/10, 1/5, 1/5)]]
sage: G.order()
10
sage: G <= C
True
sage: G >= C
False
```

We make a table of the order of the cuspidal subgroup for the first few levels:

```
sage: for N in range(11,40): print N, J0(N).cuspidal_subgroup().order()
...
11 5
12 1
13 1
14 6
15 8
16 1
17 4
18 1
19 3
20 6
21 8
22 25
23 11
24 8
```

```
25 1
26 21
27 9
28 36
29 7
30 192
31 5
32 8
33 100
34 48
35 48
36 12
37 3
38 135
39 56
```

TESTS:

```
sage: G = J0(11).finite_subgroup([[1/3,0], [0,1/5]]); G
Finite subgroup with invariants [15] over QQbar of Abelian variety J0(11) of dimension 1
sage: loads(dumps(G)) == G
True
sage: loads(dumps(G.0)) == G.0
True
```

class FiniteSubgroup(*abvar, field_of_definition=Rational Field*)

abelian_variety()

Return the abelian variety that this is a finite subgroup of.

EXAMPLES:

```
sage: J = J0(42)
sage: G = J.rational_torsion_subgroup(); G
Torsion subgroup of Abelian variety J0(42) of dimension 5
sage: G.abelian_variety()
Abelian variety J0(42) of dimension 5
```

exponent()

Return the exponent of this finite abelian group.

OUTPUT: Integer

EXAMPLES:

```
sage: t = J0(33).hecke_operator(7)
sage: G = t.kernel()[0]; G
Finite subgroup with invariants [2, 2, 2, 2, 4, 4] over QQ of Abelian variety J0(33) of dime
sage: G.exponent()
4
```

field_of_definition()

Return the field over which this finite modular abelian variety subgroup is defined. This is a field over which this subgroup is defined.

EXAMPLES:

```
sage: J = J0(42)
sage: G = J.rational_torsion_subgroup(); G
Torsion subgroup of Abelian variety J0(42) of dimension 5
```

```
sage: G.field_of_definition()
Rational Field
```

gen(*n*)

Return n^{th} generator of self.

EXAMPLES:

```
sage: J = J0(23)
sage: C = J.torsion_subgroup(3)
sage: C.gens()
[[(1/3, 0, 0, 0)], [(0, 1/3, 0, 0)], [(0, 0, 1/3, 0)], [(0, 0, 0, 1/3)]]
sage: C.gen(0)
[(1/3, 0, 0, 0)]
sage: C.gen(3)
[(0, 0, 0, 1/3)]
sage: C.gen(4)
...
IndexError: list index out of range
```

Negative indices wrap around:

```
sage: C.gen(-1)
[(0, 0, 0, 1/3)]
```

gens()

Return generators for this finite subgroup.

EXAMPLES: We list generators for several cuspidal subgroups:

```
sage: J0(11).cuspidal_subgroup().gens()
[[(0, 1/5)]]
sage: J0(37).cuspidal_subgroup().gens()
[[(0, 0, 0, 1/3)]]
sage: J0(43).cuspidal_subgroup().gens()
[[(0, 1/7, 0, 6/7, 0, 5/7)]]
sage: J1(13).cuspidal_subgroup().gens()
[[(1/19, 0, 0, 9/19)], [(0, 1/19, 1/19, 18/19)]]
sage: J0(22).torsion_subgroup(6).gens()
[[(1/6, 0, 0, 0)], [(0, 1/6, 0, 0)], [(0, 0, 1/6, 0)], [(0, 0, 0, 1/6)]]
```

intersection(*other*)

Return the intersection of the finite subgroups self and other.

INPUT:

- `other` - a finite group

OUTPUT: a finite group

EXAMPLES:

```
sage: E11a0, E11a1, B = J0(33)
sage: G = E11a0.torsion_subgroup(6); H = E11a0.torsion_subgroup(9)
sage: G.intersection(H)
Finite subgroup with invariants [3, 3] over QQ of Simple abelian subvariety 11a(1,33) of dim
sage: W = E11a1.torsion_subgroup(15)
sage: G.intersection(W)
Finite subgroup with invariants [] over QQ of Simple abelian subvariety 11a(1,33) of dimensi
sage: E11a0.intersection(E11a1)[0]
Finite subgroup with invariants [5] over QQ of Simple abelian subvariety 11a(1,33) of dimens
```

We intersect subgroups of different abelian varieties.

```
sage: E11a0, E11a1, B = J0(33)
sage: G = E11a0.torsion_subgroup(5); H = E11a1.torsion_subgroup(5)
sage: G.intersection(H)
Finite subgroup with invariants [5] over QQ of Simple abelian subvariety 11a(1,33) of dimens
sage: E11a0.intersection(E11a1)[0]
Finite subgroup with invariants [5] over QQ of Simple abelian subvariety 11a(1,33) of dimens
```

We intersect abelian varieties with subgroups:

```
sage: t = J0(33).hecke_operator(7)
sage: G = t.kernel()[0]; G
Finite subgroup with invariants [2, 2, 2, 2, 4, 4] over QQ of Abelian variety J0(33) of dime
sage: A = J0(33).old_subvariety()
sage: A.intersection(G)
Finite subgroup with invariants [2, 2, 2, 2] over QQ of Abelian subvariety of dimension 2 of
sage: A.hecke_operator(7).kernel()[0]
Finite subgroup with invariants [2, 2, 2, 2] over QQ of Abelian subvariety of dimension 2 of
sage: B = J0(33).new_subvariety()
sage: B.intersection(G)
Finite subgroup with invariants [4, 4] over QQ of Abelian subvariety of dimension 1 of J0(33
sage: B.hecke_operator(7).kernel()[0]
Finite subgroup with invariants [4, 4] over QQ of Abelian subvariety of dimension 1 of J0(33
sage: A.intersection(B)[0]
Finite subgroup with invariants [3, 3] over QQ of Abelian subvariety of dimension 2 of J0(33
```

invariants()

Return elementary invariants of this abelian group, by which we mean a nondecreasing (immutable) sequence of integers n_i, $1 \leq i \leq k$, with n_i dividing n_{i+1}, and such that this group is abstractly isomorphic to $\mathbf{Z}/n_1\mathbf{Z} \times \cdots \times \mathbf{Z}/n_k\mathbf{Z}$.

EXAMPLES:

```
sage: J = J0(38)
sage: C = J.cuspidal_subgroup(); C
Finite subgroup with invariants [3, 45] over QQ of Abelian variety J0(38) of dimension 4
sage: v = C.invariants(); v
[3, 45]
sage: v[0] = 5
...
ValueError: object is immutable; please change a copy instead.
sage: type(v[0])
<type 'sage.rings.integer.Integer'>

sage: C * 3
Finite subgroup with invariants [15] over QQ of Abelian variety J0(38) of dimension 4
```

An example involving another cuspidal subgroup:

```
sage: C = J0(22).cuspidal_subgroup(); C
Finite subgroup with invariants [5, 5] over QQ of Abelian variety J0(22) of dimension 2
sage: C.lattice()
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[1/5 1/5 4/5   0]
[  0   1   0   0]
[  0   0   1   0]
[  0   0   0 1/5]
sage: C.invariants()
[5, 5]
```

is_subgroup(*other*)

Return True exactly if self is a subgroup of other, and both are defined as subgroups of the same ambient abelian variety.

EXAMPLES:

```
sage: C = J0(22).cuspidal_subgroup()
sage: H = C.subgroup([C.0])
sage: K = C.subgroup([C.1])
sage: H.is_subgroup(K)
False
sage: K.is_subgroup(H)
False
sage: K.is_subgroup(C)
True
sage: H.is_subgroup(C)
True
```

lattice()

Return the lattice corresponding to this subgroup in the rational homology of the modular Jacobian product. The elements of the subgroup are represented by vecotors in the ambient vector space (the rational homology), and this returns the lattice they span. EXAMPLES:

```
sage: J = J0(33); C = J[0].cuspidal_subgroup(); C
Finite subgroup with invariants [5] over QQ of Simple abelian subvariety 11a(1,33) of dimens
sage: C.lattice()
Free module of degree 6 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1/5 13/5   -2 -4/5    2 -1/5]
[   0    3   -2   -1    2    0]
```

order()

Return the order (number of elements) of this finite subgroup.

EXAMPLES:

```
sage: J = J0(42)
sage: C = J.cuspidal_subgroup()
sage: C.order()
2304
```

subgroup(*gens*)

Return the subgroup of self spanned by the given generators, which all must be elements of self.

EXAMPLES:

```
sage: J = J0(23)
sage: G = J.torsion_subgroup(11); G
Finite subgroup with invariants [11, 11, 11, 11] over QQ of Abelian variety J0(23) of dimens
```

We create the subgroup of the 11-torsion subgroup of $J_0(23)$ generated by the first 11-torsion point:

```
sage: H = G.subgroup([G.0]); H
Finite subgroup with invariants [11] over QQbar of Abelian variety J0(23) of dimension 2
sage: H.invariants()
[11]
```

We can also create a subgroup from a list of objects that coerce into the ambient rational homology.

```
sage: H == G.subgroup([[1/11,0,0,0]])
True
```

class FiniteSubgroup_lattice(*abvar, lattice, field_of_definition=Algebraic Field, check=True*)

lattice()
Return lattice that defines this finite subgroup.
EXAMPLES:

```
sage: J = J0(11)
sage: G = J.finite_subgroup([[1/3,0], [0,1/5]]); G
Finite subgroup with invariants [15] over QQbar of Abelian variety J0(11) of dimension 1
sage: G.lattice()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1/3   0]
[  0 1/5]
```

class TorsionPoint (*parent, element, check=True*)

additive_order()
Return the additive order of this element.
EXAMPLES:

```
sage: J = J0(11); G = J.finite_subgroup([[1/3,0], [0,1/5]])
sage: G.0.additive_order()
3
sage: G.1.additive_order()
5
sage: (G.0 + G.1).additive_order()
15
sage: (3*G.0).additive_order()
1
```

element()
Return an underlying QQ-vector space element that defines this element of a modular abelian variety. This is a vector in the ambient Jacobian variety's rational homology.
EXAMPLES: We create some elements of $J_0(11)$:

```
sage: J = J0(11)
sage: G = J.finite_subgroup([[1/3,0], [0,1/5]]); G
Finite subgroup with invariants [15] over QQbar of Abelian variety J0(11) of dimension 1
sage: G.0.element()
(1/3, 0)
```

The underlying element is a vector over the rational numbers:

```
sage: v = (G.0-G.1).element(); v
(1/3, -1/5)
sage: type(v)
<type 'sage.modules.vector_rational_dense.Vector_rational_dense'>
```

45.5 Torsion subgroups of modular abelian varieties.

Sage can compute information about the structure of the torsion subgroup of a modular abelian variety. Sage computes a multiple of the order by computing the greatest common divisor of the orders of the torsion subgroup of the reduction of the abelian variety modulo p for various primes p. Sage computes a divisor of the order by computing the rational cuspidal subgroup. When these two bounds agree (which is often the case), we determine the exact structure of the torsion subgroup.

AUTHORS:

- William Stein (2007-03)

EXAMPLES: First we consider $J_0(50)$ where everything works out nicely:

```
sage: J = J0(50)
sage: T = J.rational_torsion_subgroup(); T
Torsion subgroup of Abelian variety J0(50) of dimension 2
sage: T.multiple_of_order()
15
sage: T.divisor_of_order()
15
sage: T.gens()
[[(1/15, 3/5, 2/5, 14/15)]]
sage: T.invariants()
[15]
sage: d = J.decomposition(); d
[
Simple abelian subvariety 50a(1,50) of dimension 1 of J0(50),
Simple abelian subvariety 50b(1,50) of dimension 1 of J0(50)
]
sage: d[0].rational_torsion_subgroup().order()
3
sage: d[1].rational_torsion_subgroup().order()
5
```

Next we make a table of the upper and lower bounds for each new factor.

```
sage: for N in range(1,38):
...     for A in J0(N).new_subvariety().decomposition():
...         T = A.rational_torsion_subgroup()
...         print '%-5s%-5s%-5s%-5s'%(N, A.dimension(), T.divisor_of_order(), T.multiple_of_order())
11   1    5    5
14   1    6    6
15   1    8    8
17   1    4    4
19   1    3    3
20   1    6    6
21   1    8    8
23   2    11   11
24   1    8    8
26   1    3    3
26   1    7    7
27   1    3    3
29   2    7    7
30   1    6    12
31   2    5    5
32   1    4    4
33   1    4    4
34   1    6    6
35   1    3    3
35   2    16   16
36   1    6    6
37   1    1    1
37   1    3    3
```

TESTS:

```
sage: T = J0(54).rational_torsion_subgroup()
sage: loads(dumps(T)) == T
True
```

class QQbarTorsionSubgroup(*abvar*)

abelian_variety()

Return the abelian variety that this is the set of all torsion points on.

OUTPUT: abelian variety

EXAMPLES:

```
sage: J0(23).qbar_torsion_subgroup().abelian_variety()
Abelian variety J0(23) of dimension 2
```

field_of_definition()

Return the field of definition of this subgroup. Since this is the group of all torsion it is defined over the base field of this abelian variety.

OUTPUT: a field

EXAMPLES:

```
sage: J0(23).qbar_torsion_subgroup().field_of_definition()
Rational Field
```

class RationalTorsionSubgroup(*abvar*)

The torsion subgroup of a modular abelian variety.

divisor_of_order()

Return a divisor of the order of this torsion subgroup of a modular abelian variety.

EXAMPLES:

```
sage: t = J0(37)[1].rational_torsion_subgroup()
sage: t.divisor_of_order()
3
```

lattice()

Return lattice that defines this torsion subgroup, if possible.

Warning: There is no known algorithm in general to compute the rational torsion subgroup. Use rational_cusp_group to obtain a subgroup of the rational torsion subgroup in general.

EXAMPLES:

```
sage: J0(11).rational_torsion_subgroup().lattice()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[  1   0]
[  0 1/5]
```

The following fails because in fact I know of no (reasonable) algorithm to provably compute the torsion subgroup in general.

```
sage: T = J0(33).rational_torsion_subgroup()
sage: T.lattice()
...
NotImplementedError: unable to compute the rational torsion subgroup in this case (there is
```

The problem is that the multiple of the order obtained by counting points over finite fields is twice the divisor of the order got from the rational cuspidal subgroup.

```
sage: T.multiple_of_order(30)
200
sage: J0(33).rational_cusp_subgroup().order()
100
```

multiple_of_order(*maxp=None*)

Return a multiple of the order of this torsion group.

The multiple is computed using characteristic polynomials of Hecke operators of odd index not dividing the level.

INPUT:

- `maxp` - (default: None) If maxp is None (the default), return gcd of best bound computed so far with bound obtained by computing gcd's of orders modulo p until this gcd stabilizes for 3 successive primes. If maxp is given, just use all primes up to and including maxp.

EXAMPLES:

```
sage: J = J0(11)
sage: G = J.rational_torsion_subgroup()
sage: G.multiple_of_order(11)
5
sage: J = J0(389)
sage: G = J.rational_torsion_subgroup(); G
Torsion subgroup of Abelian variety J0(389) of dimension 32
sage: G.multiple_of_order()
97
sage: [G.multiple_of_order(p) for p in prime_range(3,11)]
[92645296242160800, 7275, 291]
sage: [G.multiple_of_order(p) for p in prime_range(3,13)]
[92645296242160800, 7275, 291, 97]
sage: [G.multiple_of_order(p) for p in prime_range(3,19)]
[92645296242160800, 7275, 291, 97, 97, 97]

sage: J = J0(33) * J0(11) ; J.rational_torsion_subgroup().order()
...
NotImplementedError: torsion multiple only implemented for Gamma0
```

The next example illustrates calling this function with a larger input and how the result may be cached when maxp is None:

```
sage: T = J0(43)[1].rational_torsion_subgroup()
sage: T.multiple_of_order()
14
sage: T.multiple_of_order(50)
7
sage: T.multiple_of_order()
7
```

order()

Return the order of the torsion subgroup of this modular abelian variety.

This may fail if the multiple obtained by counting points modulo p exceeds the divisor obtained from the rational cuspidal subgroup.

EXAMPLES:

```
sage: a = J0(11)
sage: a.rational_torsion_subgroup().order()
5
sage: a = J0(23)
sage: a.rational_torsion_subgroup().order()
```

```
11
sage: t = J0(37)[1].rational_torsion_subgroup()
sage: t.order()
3
```

possible_orders()

Return the possible orders of this torsion subgroup, computed from a known divisor and multiple of the order.

EXAMPLES:

```
sage: J0(11).rational_torsion_subgroup().possible_orders()
[5]
sage: J0(33).rational_torsion_subgroup().possible_orders()
[100, 200]
```

Note that this function has not been implemented for $J_1(N)$, though it should be reasonably easy to do so soon (see Conrad, Edixhoven, Stein):

```
sage: J1(13).rational_torsion_subgroup().possible_orders()
...
NotImplementedError: torsion multiple only implemented for Gamma0
```

45.6 Cuspidal subgroups of modular abelian varieties

AUTHORS:

- William Stein (2007-03, 2008-02)

EXAMPLES: We compute the cuspidal subgroup of $J_1(13)$:

```
sage: A = J1(13)
sage: C = A.cuspidal_subgroup(); C
Finite subgroup with invariants [19, 19] over QQ of Abelian variety J1(13) of dimension 2
sage: C.gens()
[[(1/19, 0, 0, 9/19)], [(0, 1/19, 1/19, 18/19)]]
sage: C.order()
361
sage: C.invariants()
[19, 19]
```

We compute the cuspidal subgroup of $J_0(54)$:

```
sage: A = J0(54)
sage: C = A.cuspidal_subgroup(); C
Finite subgroup with invariants [3, 3, 3, 3, 3, 9] over QQ of Abelian variety J0(54) of dimension 4
sage: C.gens()
[[(1/3, 0, 0, 0, 0, 1/3, 0, 2/3)], [(0, 1/3, 0, 0, 0, 2/3, 0, 1/3)], [(0, 0, 1/9, 1/9, 1/9, 1/9, 1/9,
sage: C.order()
2187
sage: C.invariants()
[3, 3, 3, 3, 3, 9]
```

We compute the subgroup of the cuspidal subgroup generated by rational cusps.

```
sage: C = J0(54).rational_cusp_subgroup(); C
Finite subgroup with invariants [3, 3, 9] over QQ of Abelian variety J0(54) of dimension 4
sage: C.gens()
[[(1/3, 0, 0, 1/3, 2/3, 1/3, 0, 1/3)], [(0, 0, 1/9, 1/9, 7/9, 7/9, 1/9, 8/9)], [(0, 0, 0, 0, 0, 0, 1/
sage: C.order()
81
sage: C.invariants()
[3, 3, 9]
```

This might not give us the exact rational torsion subgroup, since it might be bigger than order 81:

```
sage: J0(54).rational_torsion_subgroup().multiple_of_order()
243
```

TESTS:

```
sage: C = J0(54).cuspidal_subgroup()
sage: loads(dumps(C)) == C
True
sage: D = J0(54).rational_cusp_subgroup()
sage: loads(dumps(D)) == D
True
```

class **CuspidalSubgroup**(*abvar, field_of_definition=Rational Field*)

EXAMPLES:

```
sage: a = J0(65)[2]
sage: t = a.cuspidal_subgroup()
sage: t.order()
6
```

lattice()

Returned cached tuple of vectors that define elements of the rational homology that generate this finite subgroup.

OUTPUT:

- `tuple` - cached

EXAMPLES:

```
sage: J = J0(27)
sage: G = J.cuspidal_subgroup()
sage: G.lattice()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1/3   0]
[  0 1/3]
```

Test that the result is cached:

```
sage: G.lattice() is G.lattice()
True
```

class **CuspidalSubgroup_generic**(*abvar, field_of_definition=Rational Field*)

class **RationalCuspidalSubgroup**(*abvar, field_of_definition=Rational Field*)

EXAMPLES:

```
sage: a = J0(65)[2]
sage: t = a.rational_cusp_subgroup()
sage: t.order()
6
```

lattice()

Return lattice that defines this group.

OUTPUT: lattice

EXAMPLES:

```
sage: G = J0(27).rational_cusp_subgroup()
sage: G.lattice()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1/3   0]
[  0   1]
```

Test that the result is cached.

```
sage: G.lattice() is G.lattice()
True
```

is_rational_cusp_gamma0(*c, N, data*)

Return True if the rational number c is a rational cusp of level N. This uses remarks in Glenn Steven's Ph.D. thesis.

INPUT:

- `c` - a cusp
- `N` - a positive integer
- `data` - the list [n for n in range(2,N) if gcd(n,N) == 1], which is passed in as a parameter purely for efficiency reasons.

EXAMPLES:

```
sage: from sage.modular.abvar.cuspidal_subgroup import is_rational_cusp_gamma0
sage: N = 27
sage: data = [n for n in range(2,N) if gcd(n,N) == 1]
sage: is_rational_cusp_gamma0(Cusp(1/3), N, data)
False
sage: is_rational_cusp_gamma0(Cusp(1), N, data)
True
sage: is_rational_cusp_gamma0(Cusp(oo), N, data)
True
sage: is_rational_cusp_gamma0(Cusp(2/9), N, data)
False
```

45.7 Homology of modular abelian varieties.

Sage can compute with homology groups associated to modular abelian varieties with coefficients in any commutative ring. Supported operations include computing matrices and characteristic polynomials of Hecke operators, rank, and rational decomposition as a direct sum of factors (obtained by cutting out kernels of Hecke operators).

AUTHORS:

- William Stein (2007-03)

EXAMPLES:

```
sage: J = J0(43)
sage: H = J.integral_homology()
sage: H
Integral Homology of Abelian variety J0(43) of dimension 3
sage: H.hecke_matrix(19)
[ 0  0 -2  0  2  0]
[ 2 -4 -2  0  2  0]
[ 0  0 -2 -2  0  0]
[ 2  0 -2 -4  2 -2]
[ 0  2  0 -2 -2  0]
[ 0  2  0 -2  0  0]
sage: H.base_ring()
Integer Ring
sage: d = H.decomposition(); d
[
Submodule of rank 2 of Integral Homology of Abelian variety J0(43) of dimension 3,
Submodule of rank 4 of Integral Homology of Abelian variety J0(43) of dimension 3
]
sage: a = d[0]
sage: a.hecke_matrix(5)
[-4  0]
[ 0 -4]
sage: a.T(7)
Hecke operator T_7 on Submodule of rank 2 of Integral Homology of Abelian variety J0(43) of dimensio
```

class Homology(*base_ring, level, weight*)

A homology group of an abelian variety, equipped with a Hecke action.

hecke_polynomial(*n, var='x'*)

Return the n-th Hecke polynomial in the given variable.

INPUT:

- `n` - positive integer
- `var` - string (default: 'x') the variable name

OUTPUT: a polynomial over ZZ in the given variable

EXAMPLES:

```
sage: H = J0(43).integral_homology(); H
Integral Homology of Abelian variety J0(43) of dimension 3
sage: f = H.hecke_polynomial(3); f
x^6 + 4*x^5 - 16*x^3 - 12*x^2 + 16*x + 16
sage: parent(f)
Univariate Polynomial Ring in x over Integer Ring
sage: H.hecke_polynomial(3,'w')
w^6 + 4*w^5 - 16*w^3 - 12*w^2 + 16*w + 16
```

class Homology_abvar(*abvar, base*)

The homology of a modular abelian variety.

abelian_variety()

Return the abelian variety that this is the homology of.

EXAMPLES:

```
sage: H = J0(48).homology()
sage: H.abelian_variety()
Abelian variety J0(48) of dimension 3
```

ambient_hecke_module()

Return the ambient Hecke module that this homology is contained in.

EXAMPLES:

```
sage: H = J0(48).homology(); H
Integral Homology of Abelian variety J0(48) of dimension 3
sage: H.ambient_hecke_module()
Integral Homology of Abelian variety J0(48) of dimension 3
```

free_module()

Return the underlying free module of this homology group.

EXAMPLES:

```
sage: H = J0(48).homology()
sage: H.free_module()
Ambient free module of rank 6 over the principal ideal domain Integer Ring
```

gen(n)

Return n^{th} generator of self.

This is not yet implemented!

EXAMPLES:

```
sage: H = J0(37).homology()
sage: H.gen(0)     # this will change
...
NotImplementedError: homology classes not yet implemented
```

gens()

Return generators of self.

This is not yet implemented!

EXAMPLES:

```
sage: H = J0(37).homology()
sage: H.gens()     # this will change
...
NotImplementedError: homology classes not yet implemented
```

hecke_bound()

Return bound on the number of Hecke operators needed to generate the Hecke algebra as a $\mathbf{Z}$-module acting on this space.

EXAMPLES:

```
sage: J0(48).homology().hecke_bound()
16
sage: J1(15).homology().hecke_bound()
32
```

hecke_matrix(n)

Return the matrix of the n-th Hecke operator acting on this homology group.

INPUT:

- n - a positive integer

OUTPUT: a matrix over the coefficient ring of this homology group

EXAMPLES:

```
sage: H = J0(23).integral_homology()
sage: H.hecke_matrix(3)
[-1 -2  2  0]
[ 0 -3  2 -2]
```

```
[ 2 -4  3 -2]
[ 2 -2  0  1]
```

The matrix is over the coefficient ring:

```
sage: J = J0(23)
sage: J.homology(QQ[I]).hecke_matrix(3).parent()
Full MatrixSpace of 4 by 4 dense matrices over Number Field in I with defining polynomial x^
```

rank()

Return the rank as a module or vector space of this homology group.

EXAMPLES:

```
sage: H = J0(5077).homology(); H
Integral Homology of Abelian variety J0(5077) of dimension 422
sage: H.rank()
844
```

submodule(*U, check=True*)

Return the submodule of this homology group given by U, which should be a submodule of the free module associated to this homology group.

INPUT:

- `U` - submodule of ambient free module (or something that defines one)
- `check` - currently ignored.

Note: We do *not* check that U is invariant under all Hecke operators.

EXAMPLES:

```
sage: H = J0(23).homology(); H
Integral Homology of Abelian variety J0(23) of dimension 2
sage: F = H.free_module()
sage: U = F.span([[1,2,3,4]])
sage: M = H.submodule(U); M
Submodule of rank 1 of Integral Homology of Abelian variety J0(23) of dimension 2
```

Note that the submodule command doesn't actually check that the object defined is a homology group or is invariant under the Hecke operators. For example, the fairly random M that we just defined is not invariant under the Hecke operators, so it is not a Hecke submodule - it is only a $\mathbf{Z}$-submodule.

```
sage: M.hecke_matrix(3)
...
ArithmeticError: subspace is not invariant under matrix
```

class **Homology_over_base**(*abvar, base_ring*)

The homology over a modular abelian variety over an arbitrary base commutative ring (not $\mathbf{Z}$ or $\mathbf{Q}$).

hecke_matrix(*n*)

Return the matrix of the n-th Hecke operator acting on this homology group.

EXAMPLES:

```
sage: t = J1(13).homology(GF(3)).hecke_matrix(3); t
[0 0 2 1]
[1 1 0 2]
[1 1 0 0]
[0 1 2 1]
sage: t.base_ring()
Finite Field of size 3
```

class **Homology_submodule**(*ambient, submodule*)

A submodule of the homology of a modular abelian variety.

ambient_hecke_module()

Return the ambient Hecke module that this homology is contained in.

EXAMPLES:

```
sage: H = J0(48).homology(); H
Integral Homology of Abelian variety J0(48) of dimension 3
sage: d = H.decomposition(); d
[
Submodule of rank 2 of Integral Homology of Abelian variety J0(48) of dimension 3,
Submodule of rank 4 of Integral Homology of Abelian variety J0(48) of dimension 3
]
sage: d[0].ambient_hecke_module()
Integral Homology of Abelian variety J0(48) of dimension 3
```

free_module()

Return the underlying free module of the homology group.

EXAMPLES:

```
sage: H = J0(48).homology()
sage: K = H.decomposition()[1]; K
Submodule of rank 4 of Integral Homology of Abelian variety J0(48) of dimension 3
sage: K.free_module()
Free module of degree 6 and rank 4 over Integer Ring
Echelon basis matrix:
[ 1  0  0  0  0  0]
[ 0  1  0  0  1 -1]
[ 0  0  1  0 -1  1]
[ 0  0  0  1  0 -1]
```

hecke_bound()

Return a bound on the number of Hecke operators needed to generate the Hecke algebra acting on this homology group.

EXAMPLES:

```
sage: d = J0(43).homology().decomposition(2); d
[
Submodule of rank 2 of Integral Homology of Abelian variety J0(43) of dimension 3,
Submodule of rank 4 of Integral Homology of Abelian variety J0(43) of dimension 3
]
```

Because the first factor has dimension 2 it corresponds to an elliptic curve, so we have a Hecke bound of 1.

```
sage: d[0].hecke_bound()
1
sage: d[1].hecke_bound()
8
```

hecke_matrix(*n*)

Return the matrix of the n-th Hecke operator acting on this homology group.

EXAMPLES:

```
sage: d = J0(125).homology(GF(17)).decomposition(2); d
[
Submodule of rank 4 of Homology with coefficients in Finite Field of size 17 of Abelian vari
Submodule of rank 4 of Homology with coefficients in Finite Field of size 17 of Abelian vari
Submodule of rank 8 of Homology with coefficients in Finite Field of size 17 of Abelian vari
]
sage: t = d[0].hecke_matrix(17); t
[16 15 15  0]
```

```
[ 0  5  0  2]
[ 2  0  5 15]
[ 0 15  0 16]
sage: t.base_ring()
Finite Field of size 17
sage: t.fcp()
(x^2 + 13*x + 16)^2
```

rank()

Return the rank of this homology group.

EXAMPLES:

```
sage: d = J0(43).homology().decomposition(2)
sage: [H.rank() for H in d]
[2, 4]
```

class **IntegralHomology**(*abvar*)

The integral homology $H_1(A, \mathbf{Z})$ of a modular abelian variety.

hecke_matrix(*n*)

Return the matrix of the n-th Hecke operator acting on this homology group.

EXAMPLES:

```
sage: J0(48).integral_homology().hecke_bound()
16
sage: t = J1(13).integral_homology().hecke_matrix(3); t
[ 0  0  2 -2]
[-2 -2  0  2]
[-2 -2  0  0]
[ 0 -2  2 -2]
sage: t.base_ring()
Integer Ring
```

hecke_polynomial(*n, var='x'*)

Return the n-th Hecke polynomial on this integral homology group.

EXAMPLES:

```
sage: f = J0(43).integral_homology().hecke_polynomial(2)
sage: f.base_ring()
Integer Ring
sage: factor(f)
(x + 2)^2 * (x^2 - 2)^2
```

class **RationalHomology**(*abvar*)

The rational homology $H_1(A, \mathbf{Q})$ of a modular abelian variety.

hecke_matrix(*n*)

Return the matrix of the n-th Hecke operator acting on this homology group.

EXAMPLES:

```
sage: t = J1(13).homology(QQ).hecke_matrix(3); t
[ 0  0  2 -2]
[-2 -2  0  2]
[-2 -2  0  0]
[ 0 -2  2 -2]
sage: t.base_ring()
Rational Field
sage: t = J1(13).homology(GF(3)).hecke_matrix(3); t
[0 0 2 1]
```

```
[1 1 0 2]
[1 1 0 0]
[0 1 2 1]
sage: t.base_ring()
Finite Field of size 3
```

hecke_polynomial(*n, var='x'*)

Return the n-th Hecke polynomial on this rational homology group.

EXAMPLES:

```
sage: f = J0(43).rational_homology().hecke_polynomial(2)
sage: f.base_ring()
Rational Field
sage: factor(f)
(x + 2) * (x^2 - 2)
```

45.8 Spaces of homomorphisms between modular abelian varieties.

EXAMPLES:

First, we consider J0(37). This jacobian has two simple factors, corresponding to distinct newforms. These two intersect nontrivially in J0(37).

```
sage: J = J0(37)
sage: D = J.decomposition() ; D
[
Simple abelian subvariety 37a(1,37) of dimension 1 of J0(37),
Simple abelian subvariety 37b(1,37) of dimension 1 of J0(37)
]
sage: D[0].intersection(D[1])
(Finite subgroup with invariants [2, 2] over QQ of Simple abelian subvariety 37a(1,37) of dimension 1
 Simple abelian subvariety of dimension 0 of J0(37))
```

As an abstract product, since these newforms are distinct, the corresponding simple abelian varieties are not isogenous, and so there are no maps between them. The endomorphism ring of the corresponding product is thus isomorphic to the direct sum of the endomorphism rings for each factor. Since the factors correspond to abelian varieties of dimension 1, these endomorphism rings are each isomorphic to ZZ.

```
sage: Hom(D[0],D[1]).gens()
()
sage: A = D[0] * D[1] ; A
Abelian subvariety of dimension 2 of J0(37) x J0(37)
sage: A.endomorphism_ring().gens()
(Abelian variety endomorphism of Abelian subvariety of dimension 2 of J0(37) x J0(37),
 Abelian variety endomorphism of Abelian subvariety of dimension 2 of J0(37) x J0(37))
sage: [ x.matrix() for x in A.endomorphism_ring().gens() ]
[[1 0 0 0]
[0 1 0 0]
[0 0 0 0]
[0 0 0 0],
 [0 0 0 0]
[0 0 0 0]
[0 0 1 0]
[0 0 0 1]]
```

However, these two newforms have a congruence between them modulo 2, which gives rise to interesting endomorphisms of J0(37).

```
sage: E = J.endomorphism_ring()
sage: E.gens()
(Abelian variety endomorphism of Abelian variety J0(37) of dimension 2,
 Abelian variety endomorphism of Abelian variety J0(37) of dimension 2)
sage: [ x.matrix() for x in E.gens() ]
[[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1],
 [ 0  1  1 -1]
[ 1  0  1  0]
[ 0  0 -1  1]
[ 0  0  0  1]]
sage: (-1*E.gens()[0] + E.gens()[1]).matrix()
[-1  1  1 -1]
[ 1 -1  1  0]
[ 0  0 -2  1]
[ 0  0  0  0]
```

Of course, these endomorphisms will be reflected in the Hecke algebra, which is in fact the full endomorphism ring of J0(37) in this case:

```
sage: J.hecke_operator(2).matrix()
[-1  1  1 -1]
[ 1 -1  1  0]
[ 0  0 -2  1]
[ 0  0  0  0]
sage: T = E.image_of_hecke_algebra()
sage: T.gens()
(Abelian variety endomorphism of Abelian variety J0(37) of dimension 2,
 Abelian variety endomorphism of Abelian variety J0(37) of dimension 2)
sage: [ x.matrix() for x in T.gens() ]
[[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1],
 [ 0  1  1 -1]
[ 1  0  1  0]
[ 0  0 -1  1]
[ 0  0  0  1]]
sage: T.index_in(E)
1
```

Next, we consider J0(33). In this case, we have both oldforms and newforms. There are two copies of J0(11), one for each degeneracy map from J0(11) to J0(33). There is also one newform at level 33. The images of the two degeneracy maps are, of course, isogenous.

```
sage: J = J0(33)
sage: D = J.decomposition()
sage: D
[
Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33),
Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33),
Simple abelian subvariety 33a(1,33) of dimension 1 of J0(33)
```

```
]
sage: Hom(D[0],D[1]).gens()
(Abelian variety morphism:
  From: Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33)
  To:   Simple abelian subvariety 11a(3,33) of dimension 1 of J0(33),)
sage: Hom(D[0],D[1]).gens()[0].matrix()
[ 0  1]
[-1  0]
```

Then this gives that the component corresponding to the sum of the oldforms will have a rank 4 endomorphism ring. We also have a rank one endomorphism ring for the newform 33a (since it is again 1-dimensional), which gives a rank 5 endomorphism ring for J0(33).

```
sage: DD = J.decomposition(simple=False) ; DD
[
Abelian subvariety of dimension 2 of J0(33),
Abelian subvariety of dimension 1 of J0(33)
]
sage: A, B = DD
sage: A == D[0] + D[1]
True
sage: A.endomorphism_ring().gens()
(Abelian variety endomorphism of Abelian subvariety of dimension 2 of J0(33),
 Abelian variety endomorphism of Abelian subvariety of dimension 2 of J0(33),
 Abelian variety endomorphism of Abelian subvariety of dimension 2 of J0(33),
 Abelian variety endomorphism of Abelian subvariety of dimension 2 of J0(33))
sage: B.endomorphism_ring().gens()
(Abelian variety endomorphism of Abelian subvariety of dimension 1 of J0(33),)
sage: E = J.endomorphism_ring() ; E.gens()
(Abelian variety endomorphism of Abelian variety J0(33) of dimension 3,
 Abelian variety endomorphism of Abelian variety J0(33) of dimension 3,
 Abelian variety endomorphism of Abelian variety J0(33) of dimension 3,
 Abelian variety endomorphism of Abelian variety J0(33) of dimension 3,
 Abelian variety endomorphism of Abelian variety J0(33) of dimension 3)
```

In this case, the image of the Hecke algebra will only have rank 3, so that it is of infinite index in the full endomorphism ring. However, if we call this image T, we can still ask about the index of T in its saturation, which is 1 in this case.

```
sage: T = E.image_of_hecke_algebra()
sage: T.gens()
(Abelian variety endomorphism of Abelian variety J0(33) of dimension 3,
 Abelian variety endomorphism of Abelian variety J0(33) of dimension 3,
 Abelian variety endomorphism of Abelian variety J0(33) of dimension 3)
sage: T.index_in(E)
+Infinity
sage: T.index_in_saturation()
1
```

AUTHORS:

- William Stein (2007-03)
- Craig Citro, Robert Bradshaw (2008-03): Rewrote with modabvar overhaul

class EndomorphismSubring(*A, gens=None*)

abelian_variety()

Return the abelian variety that this endomorphism ring is attached to.

EXAMPLES:

```
sage: J0(11).endomorphism_ring().abelian_variety()
Abelian variety J0(11) of dimension 1
```

discriminant()

Return the discriminant of this ring, which is the discriminant of the trace pairing.

Note: One knows that for modular abelian varieties, the endomorphism ring should be isomorphic to an order in a number field. However, the discriminant returned by this function will be 2^n (n = self.dimension()) times the discriminant of that order, since the elements are represented as 2d x 2d matrices. Notice, for example, that the case of a one dimensional abelian variety, whose endomorphism ring must be ZZ, has discriminant 2, as in the example below.

EXAMPLES:

```
sage: J0(33).endomorphism_ring().discriminant()
-64800
sage: J0(46).endomorphism_ring().discriminant()
24200000000
sage: J0(11).endomorphism_ring().discriminant()
2
```

image_of_hecke_algebra(*check_every=1*)

Compute the image of the Hecke algebra inside this endomorphism subring.

We simply calculate Hecke operators up to the Sturm bound, and look at the submodule spanned by them. While computing, we can check to see if the submodule spanned so far is saturated and of maximal dimension, in which case we may be done. The optional argument check_every determines how many Hecke operators we add in before checking to see if this condition is met.

EXAMPLES:

```
sage: E = J0(33).endomorphism_ring()
sage: E.image_of_hecke_algebra()
Subring of endomorphism ring of Abelian variety J0(33) of dimension 3
sage: E.image_of_hecke_algebra().gens()
(Abelian variety endomorphism of Abelian variety J0(33) of dimension 3,
 Abelian variety endomorphism of Abelian variety J0(33) of dimension 3,
 Abelian variety endomorphism of Abelian variety J0(33) of dimension 3)
sage: [ x.matrix() for x in E.image_of_hecke_algebra().gens() ]
[[1 0 0 0 0 0]
[0 1 0 0 0 0]
[0 0 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 0]
[0 0 0 0 0 1],
 [ 0  2  0 -1  1 -1]
[-1 -2  2 -1  2 -1]
[ 0  0  1 -1  3 -1]
[-2  2  0  1  1 -1]
[-1  1  0  2  0 -3]
[-1  1 -1  1  1 -2],
 [ 0  0  1 -1  1 -1]
[ 0 -1  1  0  1 -1]
[ 0  0  1  0  2 -2]
[-2  0  1  1  1 -1]
[-1  0  1  1  0 -1]
[-1  0  0  1  0 -1]]
sage: J0(33).hecke_operator(2).matrix()
```

```
[-1  0  1 -1  1 -1]
[ 0 -2  1  0  1 -1]
[ 0  0  0  0  2 -2]
[-2  0  1  0  1 -1]
[-1  0  1  1 -1 -1]
[-1  0  0  1  0 -2]
```

index_in(*other, check=True*)

Return the index of self in other.

INPUT:

- `other` - another endomorphism subgring of the same abelian variety
- `check` - bool (default: True); whether to do some type and other consistency checks

EXAMPLES:

```
sage: R = J0(33).endomorphism_ring()
sage: R.index_in(R)
1
sage: J = J0(37) ; E = J.endomorphism_ring() ; T = E.image_of_hecke_algebra()
sage: T.index_in(E)
1
sage: J = J0(22) ; E = J.endomorphism_ring() ; T = E.image_of_hecke_algebra()
sage: T.index_in(E)
+Infinity
```

index_in_saturation()

Given a Hecke algebra T, compute its index in its saturation.

EXAMPLES:

```
sage: End(J0(23)).image_of_hecke_algebra().index_in_saturation()
1
sage: End(J0(44)).image_of_hecke_algebra().index_in_saturation()
2
```

class Homspace(*domain, codomain, cat*)

A space of homomorphisms between two modular abelian varieties.

calculate_generators()

If generators haven't already been computed, calculate generators for this homspace. If they have been computed, do nothing.

EXAMPLES:

```
sage: E = End(J0(11))
sage: E.calculate_generators()
```

free_module()

Return this endomorphism ring as a free submodule of a big $\mathbf{Z}^{4nm}$, where n is the dimension of the domain abelian variety and m the dimension of the codomain.

OUTPUT: free module

EXAMPLES:

```
sage: E = Hom(J0(11), J0(22))
sage: E.free_module()
Free module of degree 8 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -3  1  1  1 -1 -1]
[ 0  1 -3  1  1  1 -1  0]
```

gen(*i=0*)

Return i-th generator of self.

INPUT:

- `i` - an integer

OUTPUT: a morphism

EXAMPLES:

```
sage: E = End(J0(22))
sage: E.gen(0).matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

gens()

Return tuple of generators for this endomorphism ring.

EXAMPLES:

```
sage: E = End(J0(22))
sage: E.gens()
(Abelian variety endomorphism of Abelian variety J0(22) of dimension 2,
 Abelian variety endomorphism of Abelian variety J0(22) of dimension 2,
 Abelian variety endomorphism of Abelian variety J0(22) of dimension 2,
 Abelian variety endomorphism of Abelian variety J0(22) of dimension 2)
```

matrix_space()

Return the underlying matrix space that we view this endomorphism ring as being embedded into.

EXAMPLES:

```
sage: E = End(J0(22))
sage: E.matrix_space()
Full MatrixSpace of 4 by 4 dense matrices over Integer Ring
```

ngens()

Return number of generators of self.

OUTPUT: integer

EXAMPLES:

```
sage: E = End(J0(22))
sage: E.ngens()
4
```

45.9 Morphisms between modular abelian varieties, including Hecke operators acting on modular abelian varieties.

Sage can compute with Hecke operators on modular abelian varieties. A Hecke operator is defined by given a modular abelian variety and an index. Given a Hecke operator, Sage can compute the characteristic polynomial, and the action of the Hecke operator on various homology groups.

AUTHORS:

- William Stein (2007-03)
- Craig Citro (2008-03)

EXAMPLES:

```
sage: A = J0(54)
sage: t5 = A.hecke_operator(5); t5
Hecke operator T_5 on Abelian variety J0(54) of dimension 4
sage: t5.charpoly().factor()
(x - 3) * (x + 3) * x^2
sage: B = A.new_subvariety(); B
Abelian subvariety of dimension 2 of J0(54)
sage: t5 = B.hecke_operator(5); t5
Hecke operator T_5 on Abelian subvariety of dimension 2 of J0(54)
sage: t5.charpoly().factor()
(x - 3) * (x + 3)
sage: t5.action_on_homology().matrix()
[ 0  3  3 -3]
[-3  3  3  0]
[ 3  3  0 -3]
[-3  6  3 -3]
```

class DegeneracyMap (*parent, A, t*)

t()

Return the list of indices defining self.

EXAMPLES:

```
sage: J0(22).degeneracy_map(44).t()
[1]
sage: J = J0(22) * J0(11)
sage: J.degeneracy_map([44,44], [2,1])
Degeneracy map from Abelian variety J0(22) x J0(11) of dimension 3 to Abelian variety J0(44)
sage: J.degeneracy_map([44,44], [2,1]).t()
[2, 1]
```

class HeckeOperator (*abvar, n*)

A Hecke operator acting on a modular abelian variety.

action_on_homology (*R=Integer Ring*)

Return the action of this Hecke operator on the homology $H_1(A; R)$ of this abelian variety with coefficients in R.

EXAMPLES:

```
sage: A = J0(43)
sage: t2 = A.hecke_operator(2); t2
Hecke operator T_2 on Abelian variety J0(43) of dimension 3
sage: h2 = t2.action_on_homology(); h2
Hecke operator T_2 on Integral Homology of Abelian variety J0(43) of dimension 3
sage: h2.matrix()
[-2  1  0  0  0  0]
[-1  1  1  0 -1  0]
[-1  0 -1  2 -1  1]
[-1  0  1  1 -1  1]
[ 0 -2  0  2 -2  1]
[ 0 -1  0  1  0 -1]
sage: h2 = t2.action_on_homology(GF(2)); h2
Hecke operator T_2 on Homology with coefficients in Finite Field of size 2 of Abelian variet
sage: h2.matrix()
[0 1 0 0 0 0]
[1 1 1 0 1 0]
```

```
[1 0 1 0 1 1]
[1 0 1 1 1 1]
[0 0 0 0 0 1]
[0 1 0 1 0 1]
```

characteristic_polynomial(*var='x'*)

Return the characteristic polynomial of this Hecke operator in the given variable.

INPUT:

- `var` - a string (default: 'x')

OUTPUT: a polynomial in var over the rational numbers.

EXAMPLES:

```
sage: A = J0(43)[1]; A
Simple abelian subvariety 43b(1,43) of dimension 2 of J0(43)
sage: t2 = A.hecke_operator(2); t2
Hecke operator T_2 on Simple abelian subvariety 43b(1,43) of dimension 2 of J0(43)
sage: f = t2.characteristic_polynomial(); f
x^2 - 2
sage: f.parent()
Univariate Polynomial Ring in x over Integer Ring
sage: f.factor()
x^2 - 2
sage: t2.characteristic_polynomial('y')
y^2 - 2
```

charpoly(*var='x'*)

Synonym for `self.characteristic_polynomial(var)`.

INPUT:

- `var` - string (default: 'x')

EXAMPLES:

```
sage: A = J1(13)
sage: t2 = A.hecke_operator(2); t2
Hecke operator T_2 on Abelian variety J1(13) of dimension 2
sage: f = t2.charpoly(); f
x^2 + 3*x + 3
sage: f.factor()
x^2 + 3*x + 3
sage: t2.charpoly('y')
y^2 + 3*y + 3
```

index()

Return the index of this Hecke operator. (For example, if this is the operator T_n, then the index is the integer n.)

OUTPUT:

- `n` - a (Sage) Integer

EXAMPLES:

```
sage: J = J0(15)
sage: t = J.hecke_operator(53)
sage: t
Hecke operator T_53 on Abelian variety J0(15) of dimension 1
sage: t.index()
53
sage: t = J.hecke_operator(54)
sage: t
```

```
Hecke operator T_54 on Abelian variety J0(15) of dimension 1
sage: t.index()
54

sage: J = J1(12345)
sage: t = J.hecke_operator(997) ; t
Hecke operator T_997 on Abelian variety J1(12345) of dimension 5405473
sage: t.index()
997
sage: type(t.index())
<type 'sage.rings.integer.Integer'>
```

matrix()

Return the matrix of self acting on the homology $H_1(A, ZZ)$ of this abelian variety with coefficients in **Z**.

EXAMPLES:

```
sage: J0(47).hecke_operator(3).matrix()
[ 0  0  1 -2  1  0 -1  0]
[ 0  0  1  0 -1  0  0  0]
[-1  2  0  0  2 -2  1 -1]
[-2  1  1 -1  3 -1 -1  0]
[-1 -1  1  0  1  0 -1  1]
[-1  0  0 -1  2  0 -1  0]
[-1 -1  2 -2  2  0 -1  0]
[ 0 -1  0  0  1  0 -1  1]

sage: J0(11).hecke_operator(7).matrix()
[-2  0]
[ 0 -2]
sage: (J0(11) * J0(33)).hecke_operator(7).matrix()
[-2  0  0  0  0  0  0  0]
[ 0 -2  0  0  0  0  0  0]
[ 0  0  0  0  2 -2  2 -2]
[ 0  0  0 -2  2  0  2 -2]
[ 0  0  0  0  2  0  4 -4]
[ 0  0 -4  0  2  2  2 -2]
[ 0  0 -2  0  2  2  0 -2]
[ 0  0 -2  0  0  2  0 -2]

sage: J0(23).hecke_operator(2).matrix()
[ 0  1 -1  0]
[ 0  1 -1  1]
[-1  2 -2  1]
[-1  1  0 -1]
```

n()

Alias for `self.index()`.

EXAMPLES:

```
sage: J = J0(17)
sage: J.hecke_operator(5).n()
5
```

class Morphism(*parent, A*)

restrict_domain(*sub*)

Restrict self to the subvariety sub of self.domain().

EXAMPLES:

```
sage: J = J0(37) ; A, B = J.decomposition()
sage: A.lattice().matrix()
[ 1 -1  1  0]
[ 0  0  2 -1]
sage: B.lattice().matrix()
[1 1 1 0]
[0 0 0 1]
sage: T = J.hecke_operator(2) ; T.matrix()
[-1  1  1 -1]
[ 1 -1  1  0]
[ 0  0 -2  1]
[ 0  0  0  0]
sage: T.restrict_domain(A)
Abelian variety morphism:
  From: Simple abelian subvariety 37a(1,37) of dimension 1 of J0(37)
  To:   Abelian variety J0(37) of dimension 2
sage: T.restrict_domain(A).matrix()
[-2  2 -2  0]
[ 0  0 -4  2]
sage: T.restrict_domain(B)
Abelian variety morphism:
  From: Simple abelian subvariety 37b(1,37) of dimension 1 of J0(37)
  To:   Abelian variety J0(37) of dimension 2
sage: T.restrict_domain(B).matrix()
[0 0 0 0]
[0 0 0 0]
```

class **Morphism_abstract** (*parent*)

A morphism between modular abelian varieties. EXAMPLES:

```
sage: t = J0(11).hecke_operator(2)
sage: from sage.modular.abvar.morphism import Morphism
sage: isinstance(t, Morphism)
True
```

cokernel ()

Return the cokernel of self.

OUTPUT:

- A - an abelian variety (the cokernel)
- phi - a quotient map from self.codomain() to the cokernel of self

EXAMPLES:

```
sage: t = J0(33).hecke_operator(2)
sage: (t-1).cokernel()
(Abelian subvariety of dimension 1 of J0(33),
 Abelian variety morphism:
  From: Abelian variety J0(33) of dimension 3
  To:   Abelian subvariety of dimension 1 of J0(33))
```

Projection will always have cokernel zero.

```
sage: J0(37).projection(J0(37)[0]).cokernel()
(Simple abelian subvariety of dimension 0 of J0(37),
 Abelian variety morphism:
  From: Simple abelian subvariety 37a(1,37) of dimension 1 of J0(37)
  To:   Simple abelian subvariety of dimension 0 of J0(37))
```

Here we have a nontrivial cokernel of a Hecke operator, as the T_2-eigenvalue for the newform 37b is 0.

```
sage: J0(37).hecke_operator(2).cokernel()
(Abelian subvariety of dimension 1 of J0(37),
 Abelian variety morphism:
  From: Abelian variety J0(37) of dimension 2
  To:   Abelian subvariety of dimension 1 of J0(37))
sage: AbelianVariety('37b').newform().q_expansion(5)
q + q^3 - 2*q^4 + O(q^5)
```

complementary_isogeny()

Returns the complementary isogeny of self.

EXAMPLES:

```
sage: J = J0(43)
sage: A = J[1]
sage: T5 = A.hecke_operator(5)
sage: T5.is_isogeny()
True
sage: T5.complementary_isogeny()
Abelian variety endomorphism of Simple abelian subvariety 43b(1,43) of dimension 2 of J0(43)
sage: (T5.complementary_isogeny() * T5).matrix()
[2 0 0 0]
[0 2 0 0]
[0 0 2 0]
[0 0 0 2]
```

factor_out_component_group()

View self as a morphism $f : A \to B$. Then $\ker(f)$ is an extension of an abelian variety C by a finite component group G. This function constructs a morphism g with domain A and codomain Q isogenous to C such that $\ker(g)$ is equal to C.

OUTPUT: a morphism

EXAMPLES:

```
sage: A,B,C = J0(33)
sage: pi = J0(33).projection(A)
sage: pi.kernel()
(Finite subgroup with invariants [5] over QQbar of Abelian variety J0(33) of dimension 3,
 Abelian subvariety of dimension 2 of J0(33))
sage: psi = pi.factor_out_component_group()
sage: psi.kernel()
(Finite subgroup with invariants [] over QQbar of Abelian variety J0(33) of dimension 3,
 Abelian subvariety of dimension 2 of J0(33))
```

ALGORITHM: We compute a subgroup G of B so that the composition $h : A \to B \to B/G$ has kernel that contains $A[n]$ and component group isomorphic to $(\mathbf{Z}/n\mathbf{Z})^{2d}$, where d is the dimension of A. Then h factors through multiplication by n, so there is a morphism $g : A \to B/G$ such that $g \circ [n] = h$. Then g is the desired morphism. We give more details below about how to transform this into linear algebra.

image()

Return the image of this morphism.

OUTPUT: an abelian variety

EXAMPLES: We compute the image of projection onto a factor of $J_0(33)$:

```
sage: A,B,C = J0(33)
sage: A
Simple abelian subvariety 11a(1,33) of dimension 1 of J0(33)
sage: f = J0(33).projection(A)
sage: f.image()
Abelian subvariety of dimension 1 of J0(33)
```

```
sage: f.image() == A
True
```

We compute the image of a Hecke operator:

```
sage: t2 = J0(33).hecke_operator(2); t2.fcp()
(x - 1) * (x + 2)^2
sage: phi = t2 + 2
sage: phi.image()
Abelian subvariety of dimension 1 of J0(33)
```

The sum of the image and the kernel is the whole space:

```
sage: phi.kernel()[1] + phi.image() == J0(33)
True
```

is_isogeny()

Return True if this morphism is an isogeny of abelian varieties.

EXAMPLES:

```
sage: J = J0(39)
sage: Id = J.hecke_operator(1)
sage: Id.is_isogeny()
True
sage: J.hecke_operator(19).is_isogeny()
False
```

kernel()

Return the kernel of this morphism.

OUTPUT:

- G - a finite group
- A - an abelian variety (identity component of the kernel)

EXAMPLES: We compute the kernel of a projection map. Notice that the kernel has a nontrivial abelian variety part.

```
sage: A, B, C = J0(33)
sage: pi = J0(33).projection(B)
sage: pi.kernel()
(Finite subgroup with invariants [20] over QQbar of Abelian variety J0(33) of dimension 3,
 Abelian subvariety of dimension 2 of J0(33))
```

We compute the kernels of some Hecke operators:

```
sage: t2 = J0(33).hecke_operator(2)
sage: t2
Hecke operator T_2 on Abelian variety J0(33) of dimension 3
sage: t2.kernel()
(Finite subgroup with invariants [2, 2, 2, 2] over QQ of Abelian variety J0(33) of dimensior
 Abelian subvariety of dimension 0 of J0(33))
sage: t3 = J0(33).hecke_operator(3)
sage: t3.kernel()
(Finite subgroup with invariants [3, 3] over QQ of Abelian variety J0(33) of dimension 3,
 Abelian subvariety of dimension 0 of J0(33))
```

45.10 Abelian varieties attached to newforms

TESTS:

```
sage: A = AbelianVariety('23a')
sage: loads(dumps(A)) == A
True
```

class ModularAbelianVariety_newform(*f, internal_name=False*)

A modular abelian variety attached to a specific newform.

endomorphism_ring()

Return the endomorphism ring of this newform abelian variety.

EXAMPLES:

```
sage: A = AbelianVariety('23a')
sage: E = A.endomorphism_ring(); E
Endomorphism ring of Newform abelian subvariety 23a of dimension 2 of J0(23)
```

We display the matrices of these two basis matrices:

```
sage: E.0.matrix()
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: E.1.matrix()
[ 0  1 -1  0]
[ 0  1 -1  1]
[-1  2 -2  1]
[-1  1  0 -1]
```

The result is cached:

```
sage: E is A.endomorphism_ring()
True
```

factor_number()

Return factor number.

OUTPUT: int

EXAMPLES:

```
sage: A = AbelianVariety('43b')
sage: A.factor_number()
1
```

label()

Return canonical label that defines this newform modular abelian variety.

OUTPUT: string

EXAMPLES:

```
sage: A = AbelianVariety('43b')
sage: A.label()
'43b'
```

newform(*names=None*)

Return the newform that this modular abelian variety is attached to.

EXAMPLES:

```
sage: f = Newform('37a')
sage: A = f.abelian_variety()
sage: A.newform()
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + O(q^6)
```

```
sage: A.newform() is f
True
```

If the a variable name has not ben specified, we must specify one:

```
sage: A = AbelianVariety('67b')
sage: A.newform()
...
TypeError: You must specify the name of the generator.
sage: A.newform('alpha')
q + alpha*q^2 + (-alpha - 3)*q^3 + (-3*alpha - 3)*q^4 - 3*q^5 + O(q^6)
```

If the eigenform is actually over $\mathbf{Q}$ then we don't have to specify the name:

```
sage: A = AbelianVariety('67a')
sage: A.newform()
q + 2*q^2 - 2*q^3 + 2*q^4 + 2*q^5 + O(q^6)
```

45.11 L-series of modular abelian varieties

At the moment very little functionality is implemented – this is mostly a placeholder for future planned work.

AUTHOR:

- William Stein (2007-03)

TESTS:

```
sage: L = J0(37)[0].padic_lseries(5)
sage: loads(dumps(L)) == L
True
sage: L = J0(37)[0].lseries()
sage: loads(dumps(L)) == L
True
```

class **Lseries**(*abvar*)

Base class for L-series attached to modular abelian varieties.

abelian_variety()

Return the abelian variety that this L-series is attached to.

OUTPUT: a modular abelian variety

EXAMPLES:

```
sage: J0(11).padic_lseries(7).abelian_variety()
Abelian variety J0(11) of dimension 1
```

class **Lseries_complex**(*abvar*)

A complex L-series attached to a modular abelian variety.

EXAMPLES:

```
sage: A = J0(37)
sage: A.lseries()
Complex L-series attached to Abelian variety J0(37) of dimension 2
```

rational_part()

Return the rational part of this L-function at the central critical value 1.

NOTE: This is not yet implemented.

EXAMPLES:

```
sage: J0(37).lseries().rational_part()
...
NotImplementedError
```

class **Lseries_padic**(*abvar, p*)

A p-adic L-series attached to a modular abelian variety.

power_series(*n=2, prec=5*)

Return the n-th approximation to this p-adic L-series as a power series in T. Each coefficient is a p-adic number whose precision is provably correct.

NOTE: This is not yet implemented.

EXAMPLES:

```
sage: L = J0(37)[0].padic_lseries(5)
sage: L.power_series()
...
NotImplementedError
sage: L.power_series(3,7)
...
NotImplementedError
```

prime()

Return the prime p of this p-adic L-series.

EXAMPLES:

```
sage: J0(11).padic_lseries(7).prime()
7
```

CHAPTER

FORTYSIX

MISCELLANEOUS MODULAR-FORM-RELATED MODULES

46.1 Dirichlet characters

A `DirichletCharacter` is the extension of a homomorphism

$$(\mathbf{Z}/N\mathbf{Z})^* \to R^*,$$

for some ring R, to the map $\mathbf{Z}/N\mathbf{Z} \to R$ obtained by sending those $x \in \mathbf{Z}/N\mathbf{Z}$ with $\gcd(N, x) > 1$ to 0.

EXAMPLES:

```
sage: G = DirichletGroup(35)
sage: x = G.gens()
sage: e = x[0]*x[1]^2; e
[zeta12^3, zeta12^2 - 1]
sage: e.order()
12
```

This illustrates a canonical coercion.

```
sage: e = DirichletGroup(5, QQ).0
sage: f = DirichletGroup(5,CyclotomicField(4)).0
sage: e*f
[-zeta4]
```

AUTHORS:

- William Stein (2005-09-02): Fixed bug in comparison of Dirichlet characters. It was checking that their values were the same, but not checking that they had the same level!
- William Stein (2006-01-07): added more examples
- William Stein (2006-05-21): added examples of everything; fix a *lot* of tiny bugs and design problem that became clear when creating examples.
- Craig Citro (2008-02-16): speed up __call__ method for Dirichlet characters, miscellaneous fixes

class `DirichletCharacter` (*parent, x, check=True*)
A Dirichlet character

bar()
Return the complex conjugate of this Dirichlet character.
EXAMPLES:

```
sage: e = DirichletGroup(5).0
sage: e
[zeta4]
sage: e.bar()
[-zeta4]
```

base_ring()
Returns the base ring of this Dirichlet character.
EXAMPLES:

```
sage: G = DirichletGroup(11)
sage: G.gen(0).base_ring()
Cyclotomic Field of order 10 and degree 4
sage: G = DirichletGroup(11, RationalField())
sage: G.gen(0).base_ring()
Rational Field
```

bernoulli(*k, algorithm='recurrence', cache=True, **opts*)
Returns the generalized Bernoulli number $B_{k,eps}$.
INPUT:

- `k` - an integer
- `algorithm` - string (default: 'recurrence'); either 'recurrence' or 'definition'. The 'recurrence' algorithm expresses generalized Bernoulli numbers in terms of classical Bernoulli numbers using a recurrence formula and is usually optimal. In this case `**opts` is passed onto the bernoulli function.
- `cache` - if True, cache answers

Let eps be this character (not necessarily primitive), and let $k \geq 0$ be an integer weight. This function computes the (generalized) Bernoulli number $B_{k,eps}$, e.g., as defined on page 44 of Diamond-Im:

$$\sum_{a=1}^{N} \varepsilon(a) t * e^{at}/(e^{Nt} - 1) = sum_{k=0}^{\infty} B_{k,eps}/k! t^k.$$

where N is the modulus of ε.
The default algorithm is the recurrence on page 656 of Cohen's GTM 'Number Theory and Diophantine Equations', section 9.
EXAMPLES:

```
sage: G = DirichletGroup(13)
sage: e = G.0
sage: e.bernoulli(5)
7430/13*zeta12^3 - 34750/13*zeta12^2 - 11380/13*zeta12 + 9110/13
sage: eps = DirichletGroup(9).0
sage: eps.bernoulli(3)
10*zeta6 + 4
sage: eps.bernoulli(3, algorithm="definition")
10*zeta6 + 4
```

change_ring(*R*)
Returns the base extension of self to the ring R.
EXAMPLE:

```
sage: e = DirichletGroup(7, QQ).0
sage: f = e.change_ring(QuadraticField(3, 'a'))
sage: f.parent()
Group of Dirichlet characters of modulus 7 over Number Field in a with defining polynomial x
```

```
sage: e = DirichletGroup(13).0
sage: e.change_ring(QQ)
...
ValueError: cannot coerce element of order 12 into self
```

conductor()

Computes and returns the conductor of this character.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: a.conductor()
4
sage: b.conductor()
5
sage: (a*b).conductor()
20
```

decomposition()

Return the decomposition of self as a product of Dirichlet characters of prime power modulus, where the prime powers exactly divide the modulus of this character.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: c = a*b
sage: d = c.decomposition(); d
[[-1], [zeta4]]
sage: d[0].parent()
Group of Dirichlet characters of modulus 4 over Cyclotomic Field of order 4 and degree 2
sage: d[1].parent()
Group of Dirichlet characters of modulus 5 over Cyclotomic Field of order 4 and degree 2
```

We can't multiply directly, since coercion of one element into the other parent fails in both cases:

```
sage: d[0]*d[1] == c
...
TypeError: unsupported operand parent(s) for '*': 'Group of Dirichlet characters of modulus
```

We can multiply if we're explicit about where we want the multiplication to take place.

```
sage: G(d[0])*G(d[1]) == c
True
```

Conductors that are divisible by various powers of 2 present some problems as the multiplicative group modulo 2^k is trivial for $k = 1$ and non-cyclic for $k \geq 3$:

```
sage: (DirichletGroup(18).0).decomposition()
[[], [zeta6]]
sage: (DirichletGroup(36).0).decomposition()
[[-1], [1]]
sage: (DirichletGroup(72).0).decomposition()
[[-1, 1], [1]]
```

element()

Return the underlying $\mathbf{Z}/n\mathbf{Z}$-module vector of exponents.

Warning: Please do not change the entries of the returned vector; this vector is mutable *only* because immutable vectors are implemented yet.

EXAMPLE:

```
sage: G.<a,b> = DirichletGroup(20)
sage: a.element()
(2, 0)
sage: b.element()
(0, 1)
```

extend(*M*)

Returns the extension of this character to a Dirichlet character modulo the multiple M of the modulus.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: H.<c> = DirichletGroup(4)
sage: c.extend(20)
[-1, 1]
sage: a
[-1, 1]
sage: c.extend(20) == a
True
```

galois_orbit(*sort=True*)

Return the orbit of this character under the action of the absolute Galois group of the prime subfield of the base ring.

EXAMPLES:

```
sage: G = DirichletGroup(30); e = G.1
sage: e.galois_orbit()
[[1, zeta4], [1, -zeta4]]
```

Another example:

```
sage: G = DirichletGroup(13)
sage: G.galois_orbits()
[
[[1]],
[[zeta12], [zeta12^3 - zeta12], [-zeta12], [-zeta12^3 + zeta12]],
[[zeta12^2], [-zeta12^2 + 1]],
[[zeta12^3], [-zeta12^3]],
[[zeta12^2 - 1], [-zeta12^2]],
[[-1]]
]
sage: e = G.0
sage: e
[zeta12]
sage: e.galois_orbit()
[[zeta12], [zeta12^3 - zeta12], [-zeta12], [-zeta12^3 + zeta12]]
sage: e = G.0^2; e
[zeta12^2]
sage: e.galois_orbit()
[[zeta12^2], [-zeta12^2 + 1]]
```

A non-example:

```
sage: chi = DirichletGroup(7, Integers(9), zeta = Integers(9)(2)).0
sage: chi.galois_orbit()
...
TypeError: Galois orbits only defined if base ring is an integral domain
```

gauss_sum(*a=1*)

Return a Gauss sum associated to this Dirichlet character.

The Gauss sum associated to χ is

$$g_a(\chi) = \sum_{r \in \mathbf{Z}/m\mathbf{Z}} \chi(r)\,\zeta^{ar},$$

where m is the modulus of χ and ζ is a primitive m^{th} root of unity, i.e., ζ is `self.parent().zeta()`.
FACTS: If the modulus is a prime p and the character is nontrivial, then the Gauss sum has absolute value $\sqrt{p}$.
CACHING: Computed Gauss sums are *not* cached with this character.
EXAMPLES:

```
sage: G = DirichletGroup(3)
sage: e = G([-1])
sage: e.gauss_sum(1)
2*zeta6 - 1
sage: e.gauss_sum(2)
-2*zeta6 + 1
sage: norm(e.gauss_sum())
3

sage: G = DirichletGroup(13)
sage: e = G.0
sage: e.gauss_sum()
-zeta156^46 + zeta156^45 + zeta156^42 + zeta156^41 + 2*zeta156^40 + zeta156^37 - zeta156^36
sage: factor(norm(e.gauss_sum()))
13^24
```

gauss_sum_numerical(*prec=53, a=1*)

Return a Gauss sum associated to this Dirichlet character as an approximate complex number with prec bits of precision.
INPUT:

- `prec` - integer (deafault: 53), *bits* of precision
- `a` - integer, as for gauss_sum.

The Gauss sum associated to χ is

$$g_a(\chi) = \sum_{r \in \mathbf{Z}/m\mathbf{Z}} \chi(r)\,\zeta^{ar},$$

where m is the modulus of χ and ζ is a primitive m^{th} root of unity, i.e., ζ is `self.parent().zeta()`.
EXAMPLES:

```
sage: G = DirichletGroup(3)
sage: e = G.0
sage: abs(e.gauss_sum_numerical())
1.7320508075...
sage: sqrt(3.0)
1.73205080756888
sage: e.gauss_sum_numerical(a=2)
-...e-15 - 1.7320508075...*I
sage: e.gauss_sum_numerical(a=2, prec=100)
4.7331654313260708324703713917e-30 - 1.7320508075688772935274463415*I
sage: G = DirichletGroup(13)
sage: e = G.0
sage: e.gauss_sum_numerical()
-3.07497205... + 1.8826966926...*I
sage: abs(e.gauss_sum_numerical())
3.60555127546...
```

```
sage: sqrt(13.0)
3.60555127546399
```

is_even()

Return `True` if and only if $\varepsilon(-1) = 1$.

EXAMPLES:

```
sage: G = DirichletGroup(13)
sage: e = G.0
sage: e.is_even()
False
sage: e(-1)
-1
sage: [e.is_even() for e in G]
[True, False, True, False, True, False, True, False, True, False, True, False]
```

Note that `is_even` need not be the negation of is_odd, e.g., in characteristic 2:

```
sage: G.<e> = DirichletGroup(13, GF(4,'a'))
sage: e.is_even()
True
sage: e.is_odd()
True
```

is_odd()

Return `True` if and only if $\varepsilon(-1) = -1$.

EXAMPLES:

```
sage: G = DirichletGroup(13)
sage: e = G.0
sage: e.is_odd()
True
sage: [e.is_odd() for e in G]
[False, True, False, True, False, True, False, True, False, True, False, True]
```

Note that `is_even` need not be the negation of is_odd, e.g., in characteristic 2:

```
sage: G.<e> = DirichletGroup(13, GF(4,'a'))
sage: e.is_even()
True
sage: e.is_odd()
True
```

is_primitive()

Return `True` if and only if this character is primitive, i.e., its conductor equals its modulus.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: a.is_primitive()
False
sage: b.is_primitive()
False
sage: (a*b).is_primitive()
True
```

is_trivial()

Returns `True` if this is the trivial character, i.e., has order 1.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: a.is_trivial()
False
sage: (a^2).is_trivial()
True
```

jacobi_sum(*char, check=True*)

Return the Jacobi sum associated to these Dirichlet characters (i.e., J(self,char)).

EXAMPLES:

```
sage: D = DirichletGroup(13)
sage: e = D.0
sage: f = D[-2]
sage: e.jacobi_sum(f)
3*zeta156^26 + 2*zeta156^13 - 3
sage: f.jacobi_sum(e)
3*zeta156^26 + 2*zeta156^13 - 3
sage: p = 7
sage: DP = DirichletGroup(p)
sage: f = DP.0
sage: e.jacobi_sum(f)
...
NotImplementedError: Characters must be from the same Dirichlet Group.
sage: all_jacobi_sums = [(DP[i],DP[j],DP[i].jacobi_sum(DP[j]))              ...
...
sage: for s in all_jacobi_sums:
...       print s
([1], [1], 5)
([1], [zeta6], 0)
([1], [zeta6 - 1], 0)
([1], [-1], 0)
([1], [-zeta6], 0)
([1], [-zeta6 + 1], 0)
([zeta6], [zeta6], -zeta42^7 + 3)
([zeta6], [zeta6 - 1], 2*zeta42^7 + 1)
([zeta6], [-1], -2*zeta42^7 - 1)
([zeta6], [-zeta6], zeta42^7 - 3)
([zeta6], [-zeta6 + 1], 1)
([zeta6 - 1], [zeta6 - 1], -3*zeta42^7 + 2)
([zeta6 - 1], [-1], 2*zeta42^7 + 1)
([zeta6 - 1], [-zeta6], -1)
([zeta6 - 1], [-zeta6 + 1], -zeta42^7 - 2)
([-1], [-1], 1)
([-1], [-zeta6], -2*zeta42^7 + 3)
([-1], [-zeta6 + 1], 2*zeta42^7 - 3)
([-zeta6], [-zeta6], 3*zeta42^7 - 1)
([-zeta6], [-zeta6 + 1], -2*zeta42^7 + 3)
([-zeta6 + 1], [-zeta6 + 1], zeta42^7 + 2)
```

Let's check that trivial sums are being calculated correctly:

```
sage: N = 13
sage: D = DirichletGroup(N)
sage: g = D(1)
sage: g.jacobi_sum(g)
11
sage: sum([g(x)*g(1-x) for x in IntegerModRing(N)])
11
```

Now let's take a look at a non-prime modulus. One reason we don't like non-prime moduli is that certain

identities that are used in the code are not valid for non-prime moduli:

```
sage: N = 9
sage: D = DirichletGroup(N)
sage: g = D(1)
sage: g.jacobi_sum(g)
...
ValueError: Characters must have prime moduli at this time -- use check=False to allow non-p
sage: g.jacobi_sum(g, check=False)
5
sage: sum([g(x)*g(1-x) for x in IntegerModRing(N)])
3
```

TODO: Implement Jacobi sums for characters with output in finite fields GF(q) for q non-prime.

kernel()

Return the kernel of this character.

OUTPUT: Currently the kernel is returned as a list. This may change.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20)
sage: a.kernel()
[1, 9, 13, 17]
sage: b.kernel()
[1, 11]
```

kloosterman_sum(*a=1, b=0*)

Return the "twisted" Kloosterman sum associated to this Dirichlet character. This includes Gauss sums, classical Kloosterman sums, Salie sums, etc.

The Kloosterman sum associated to chi and the integers a,b is $$ K(a,b,chi) = sum_{r in (Z/mZ)^times} chi(r),zeta^{ar+br^{-1}}, $$ where m is the modulus of chi and $zeta$ is a primitive mth root of unity, i.e., $zeta$ is code{self.parent().zeta()}. This reduces to to the Gauss sum if b=0.

CACHING: Computed Kloosterman sums are emph{not} cached with this character.

EXAMPLES:

```
sage: G = DirichletGroup(3)
sage: e = G([-1])
sage: e.kloosterman_sum(3,5)
-2*zeta6 + 1
sage: G = DirichletGroup(20)
sage: e = G([1 for  u in G.unit_gens()])
sage: e.kloosterman_sum(7,17)
-2*zeta20^6 + 2*zeta20^4 + 4
```

kloosterman_sum_numerical(*prec=53, a=1, b=0*)

Return the Kloosterman sum associated to this Dirichlet character as an approximate complex number with prec bits of precision.

INPUT: - prec – integer (deafault: 53), *bits* of precision - a – integer, as for kloosterman_sum - b – integer, as for kloosterman_sum.

EXAMPLES:

```
sage: G = DirichletGroup(3)
sage: e = G.0
```

The real component of the numerical value of e is near zero:

```
sage: v=e.kloosterman_sum_numerical()
sage: v.real() < 1.0e15
True
sage: v.imag()
```

```
1.73205080757
sage: G = DirichletGroup(20)
sage: e = G.1
sage: e.kloosterman_sum_numerical(53,3,11)
3.80422606518 - 3.80422606518*I
```

level()

Synonym for modulus.

EXAMPLE:

```
sage: e = DirichletGroup(100, QQ).0
sage: e.level()
100
```

maximize_base_ring()

Let

$$\varepsilon : (\mathbf{Z}/N\mathbf{Z})^* \to \mathbf{Q}(\zeta_n)$$

be a Dirichlet character. This function returns an equal Dirichlet character

$$\chi : (\mathbf{Z}/N\mathbf{Z})^* \to \mathbf{Q}(\zeta_m)$$

where m is the least common multiple of n and the exponent of $(\mathbf{Z}/N\mathbf{Z})^*$.

EXAMPLES:

```
sage: G.<a,b> = DirichletGroup(20,QQ)
sage: b.maximize_base_ring()
[1, -1]
sage: b.maximize_base_ring().base_ring()
Cyclotomic Field of order 4 and degree 2
sage: DirichletGroup(20).base_ring()
Cyclotomic Field of order 4 and degree 2
```

minimize_base_ring()

Return a Dirichlet character that equals this one, but over as small a subfield (or subring) of the base ring as possible.

Note: This function is currently only implemented when the base ring is a number field. It's the identity function in characteristic p.

EXAMPLES:

```
sage: G = DirichletGroup(13)
sage: e = DirichletGroup(13).0
sage: e.base_ring()
Cyclotomic Field of order 12 and degree 4
sage: e.minimize_base_ring().base_ring()
Cyclotomic Field of order 12 and degree 4
sage: (e^2).minimize_base_ring().base_ring()
Cyclotomic Field of order 6 and degree 2
sage: (e^3).minimize_base_ring().base_ring()
Cyclotomic Field of order 4 and degree 2
sage: (e^12).minimize_base_ring().base_ring()
Rational Field
```

modulus()

The modulus of this character.

EXAMPLES:

```
sage: e = DirichletGroup(100, QQ).0
sage: e.modulus()
100
sage: e.conductor()
4
```

multiplicative_order()

The order of this character.

EXAMPLES:

```
sage: e = DirichletGroup(100).1
sage: e.order()    # same as multiplicative_order, since group is multiplicative
20
sage: e.multiplicative_order()
20
sage: e = DirichletGroup(100).0
sage: e.multiplicative_order()
2
```

primitive_character()

Returns the primitive character associated to self.

EXAMPLES:

```
sage: e = DirichletGroup(100).0; e
[-1, 1]
sage: e.conductor()
4
sage: f = e.primitive_character(); f
[-1]
sage: f.modulus()
4
```

restrict(*M*)

Returns the restriction of this character to a Dirichlet character modulo the divisor M of the modulus, which must also be a multiple of the conductor of this character.

EXAMPLES:

```
sage: e = DirichletGroup(100).0
sage: e.modulus()
100
sage: e.conductor()
4
sage: e.restrict(20)
[-1, 1]
sage: e.restrict(4)
[-1]
sage: e.restrict(50)
...
ValueError: conductor(=4) must divide M(=50)
```

values()

Returns a list of the values of this character on each integer between 0 and the modulus.

EXAMPLES:

```
sage: e = DirichletGroup(20)(1)
sage: e.values()
[0, 1, 0, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 0, 1, 0, 1]
sage: e = DirichletGroup(20).gen(0)
sage: print e.values()
```

```
[0, 1, 0, -1, 0, 0, 0, -1, 0, 1, 0, -1, 0, 1, 0, 0, 0, 1, 0, -1]
sage: e = DirichletGroup(20).gen(1)
sage: e.values()
[0, 1, 0, -zeta4, 0, 0, 0, zeta4, 0, -1, 0, 1, 0, -zeta4, 0, 0, 0, zeta4, 0, -1]
sage: e = DirichletGroup(21).gen(0) ; e.values()
[0, 1, -1, 0, 1, -1, 0, 0, -1, 0, 1, -1, 0, 1, 0, 0, 1, -1, 0, 1, -1]
sage: e = DirichletGroup(21, base_ring=GF(37)).gen(0) ; e.values()
[0, 1, 36, 0, 1, 36, 0, 0, 36, 0, 1, 36, 0, 1, 0, 0, 1, 36, 0, 1, 36]
sage: e = DirichletGroup(21, base_ring=GF(3)).gen(0) ; e.values()
[0, 1, 2, 0, 1, 2, 0, 0, 2, 0, 1, 2, 0, 1, 0, 0, 1, 2, 0, 1, 2]

sage: chi = DirichletGroup(100151, CyclotomicField(10)).0
sage: ls = chi.values() ; ls[0:10]
[0,
1,
-zeta10^3,
-zeta10,
-zeta10,
1,
zeta10^3 - zeta10^2 + zeta10 - 1,
zeta10,
zeta10^3 - zeta10^2 + zeta10 - 1,
zeta10^2]
```

values_on_gens()

Returns a tuple of the values of this character on each of the minimal generators of $(\mathbf{Z}/N\mathbf{Z})^*$, where N is the modulus.

EXAMPLES:

```
sage: e = DirichletGroup(16)([-1, 1])
sage: e.values_on_gens ()
(-1, 1)
```

DirichletGroup(*modulus, base_ring=None, zeta=None, zeta_order=None, names=None, integral=False*)

The group of Dirichlet characters modulo N with values in the subgroup $\langle \zeta_n \rangle$ of the multiplicative group of the `base_ring`. If the base_ring is omitted then we use $\mathbf{Q}(\zeta_n)$, where n is the exponent of $(\mathbf{Z}/N\mathbf{Z})^*$. If ζ is omitted then we compute and use a maximal-order zeta in base_ring, if possible.

INPUT:

- `modulus` - int
- `base_ring` - Ring (optional), where characters take their values (should be an integral domain).
- `zeta` - Element (optional), element of base_ring; zeta is a root of unity
- `zeta_order` - int (optional), the order of zeta
- `names` - ignored (needed so G.... = DirichletGroup(...) notation works)
- `integral` - boolean (default: `False`). If `True`, return the group with base_ring the ring of integers in the smallest choice of CyclotomicField. Ignored if base_ring is not `None`.

OUTPUT:

- `DirichletGroup` - a group of Dirichlet characters.

EXAMPLES:

The default base ring is a cyclotomic field of order the exponent of $(\mathbf{Z}/N\mathbf{Z})^*$.

```
sage: DirichletGroup(20)
Group of Dirichlet characters of modulus 20 over Cyclotomic Field of order 4 and degree 2
```

We create the group of Dirichlet character mod 20 with values in the rational numbers:

```
sage: G = DirichletGroup(20, QQ); G
Group of Dirichlet characters of modulus 20 over Rational Field
sage: G.order()
4
sage: G.base_ring()
Rational Field
```

The elements of G print as lists giving the values of the character on the generators of $(Z/NZ)^*$:

```
sage: list(G)
[[1, 1], [-1, 1], [1, -1], [-1, -1]]
```

Next we construct the group of Dirichlet character mod 20, but with values in Q(zeta_n):

```
sage: G = DirichletGroup(20)
sage: G.list()
[[1, 1], [-1, 1], [1, zeta4], [-1, zeta4], [1, -1], [-1, -1], [1, -zeta4], [-1, -zeta4]]
```

We next compute several invariants of G:

```
sage: G.gens()
([-1, 1], [1, zeta4])
sage: G.unit_gens()
[11, 17]
sage: G.zeta()
zeta4
sage: G.zeta_order()
4
```

In this example we create a Dirichlet character with values in a number field. We have to give zeta, but not its order.

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^4 + 1)
sage: G = DirichletGroup(5, K, a); G
Group of Dirichlet characters of modulus 5 over Number Field in a with defining polynomial x^4 +
sage: G.list()
[[1], [a^2], [-1], [-a^2]]


sage: G.<e> = DirichletGroup(13)
sage: loads(G.dumps()) == G
True


sage: G = DirichletGroup(19, GF(5))
sage: loads(G.dumps()) == G
True
```

We compute a Dirichlet group over a large prime field.

```
sage: p = next_prime(10^40)
sage: g = DirichletGroup(19, GF(p)); g
Group of Dirichlet characters of modulus 19 over Finite Field of size 10000000000000000000000000
```

Note that the root of unity has small order, i.e., it is not the largest order root of unity in the field.

```
sage: g.zeta_order()
2

sage: r4 = CyclotomicField(4).ring_of_integers()
sage: G = DirichletGroup(60, r4)
sage: G.gens()
([-1, 1, 1], [1, -1, 1], [1, 1, zeta4])
sage: val = G.gens()[2].values_on_gens()[2] ; val
zeta4
sage: parent(val)
Maximal Order in Cyclotomic Field of order 4 and degree 2
sage: r4.residue_field(r4.ideal(29).factor()[0][0])(val)
17
sage: r4.residue_field(r4.ideal(29).factor()[0][0])(val) * GF(29)(3)
22
sage: r4.residue_field(r4.ideal(29).factor()[0][0])(G.gens()[2].values_on_gens()[2]) * 3
22
sage: parent(r4.residue_field(r4.ideal(29).factor()[0][0])(G.gens()[2].values_on_gens()[2]) * 3)
Residue field of Fractional ideal (-2*zeta4 + 5)

sage: DirichletGroup(60, integral=True)
Group of Dirichlet characters of modulus 60 over Maximal Order in Cyclotomic Field of order 4 an
sage: parent(DirichletGroup(60, integral=True).gens()[2].values_on_gens()[2])
Maximal Order in Cyclotomic Field of order 4 and degree 2
```

class **DirichletGroup_class**(*modulus, zeta, zeta_order*)

Group of Dirichlet characters modulo N over a given base ring R.

base_extend(*R*)

Returns the Dirichlet group over R obtained by extending scalars, with the same modulus and root of unity as self.

EXAMPLES:

```
sage: G = DirichletGroup(7,QQ); G
Group of Dirichlet characters of modulus 7 over Rational Field
sage: H = G.base_extend(CyclotomicField(6)); H
Group of Dirichlet characters of modulus 7 over Cyclotomic Field of order 6 and degree 2
sage: H.zeta()
-1
sage: G.base_extend(ZZ)
...
TypeError: No coercion map from 'Rational Field' to 'Integer Ring' is defined.
```

change_ring(*R, zeta=None, zeta_order=None*)

Returns the Dirichlet group over R with the same modulus as self.

EXAMPLES:

```
sage: G = DirichletGroup(7,QQ); G
Group of Dirichlet characters of modulus 7 over Rational Field
sage: G.change_ring(CyclotomicField(6))
Group of Dirichlet characters of modulus 7 over Cyclotomic Field of order 6 and degree 2
```

decomposition()

Returns the Dirichlet groups of prime power modulus corresponding to primes dividing modulus.

(Note that if the modulus is 2 mod 4, there will be a "factor" of $(\mathbf{Z}/2\mathbf{Z})^*$, which is the trivial group.)

EXAMPLES:

```
sage: DirichletGroup(20).decomposition()
[
Group of Dirichlet characters of modulus 4 over Cyclotomic Field of order 4 and degree 2,
Group of Dirichlet characters of modulus 5 over Cyclotomic Field of order 4 and degree 2
]
sage: DirichletGroup(20,GF(5)).decomposition()
[
Group of Dirichlet characters of modulus 4 over Finite Field of size 5,
Group of Dirichlet characters of modulus 5 over Finite Field of size 5
]
```

exponent()

Return the exponent of this group.

EXAMPLES:

```
sage: DirichletGroup(20).exponent()
4
sage: DirichletGroup(20,GF(3)).exponent()
2
sage: DirichletGroup(20,GF(2)).exponent()
1
sage: DirichletGroup(37).exponent()
36
```

galois_orbits(*v=None, reps_only=False, sort=True, check=True*)

Return a list of the Galois orbits of Dirichlet characters in self, or in v if v is not None.

INPUT:

- `v` - (optional) list of elements of self
- `reps_only` - (optional: default False) if True only returns representatives for the orbits.
- `sort` - (optional: default True) whether to sort the list of orbits and the orbits themselves (slightly faster if False).
- `check` - (optional, default: True) whether or not to explicitly coerce each element of v into self.

The Galois group is the absolute Galois group of the prime subfield of Frac(R). If R is not a domain, an error will be raised.

EXAMPLES:

```
sage: DirichletGroup(20).galois_orbits()
[
[[1, 1]],
[[1, zeta4], [1, -zeta4]],
[[1, -1]],
[[-1, 1]],
[[-1, zeta4], [-1, -zeta4]],
[[-1, -1]]
]
sage: DirichletGroup(17, Integers(6), zeta=Integers(6)(5)).galois_orbits()
...
TypeError: Galois orbits only defined if base ring is an integral domain
sage: DirichletGroup(17, Integers(9), zeta=Integers(9)(2)).galois_orbits()
...
TypeError: Galois orbits only defined if base ring is an integral domain
```

gen(*n=0*)

Return the n-th generator of self.

EXAMPLES:

```
sage: G = DirichletGroup(20)
sage: G.gen(0)
[-1, 1]
sage: G.gen(1)
[1, zeta4]
sage: G.gen(2)
...
IndexError: n(=2) must be between 0 and 1

sage: G.gen(-1)
...
IndexError: n(=-1) must be between 0 and 1
```

gens()

Returns generators of self.

EXAMPLES:

```
sage: G = DirichletGroup(20)
sage: G.gens()
([-1, 1], [1, zeta4])
```

integers_mod()

Returns the group of integers $\mathbf{Z}/N\mathbf{Z}$ where N is the modulus of self.

EXAMPLES:

```
sage: G = DirichletGroup(20)
sage: G.integers_mod()
Ring of integers modulo 20
```

modulus()

Returns the modulus of self.

EXAMPLES:

```
sage: G = DirichletGroup(20)
sage: G.modulus()
20
```

ngens()

Returns the number of generators of self.

EXAMPLES:

```
sage: G = DirichletGroup(20)
sage: G.ngens()
2
```

order()

Return the number of elements of self. This is the same as len(self).

EXAMPLES:

```
sage: DirichletGroup(20).order()
8
sage: DirichletGroup(37).order()
36
```

random_element()

Return a random element of self.

The element is computed by multiplying a random power of each generator together, where the power is between 0 and the order of the generator minus 1, inclusive.

EXAMPLES:

```
sage: DirichletGroup(37).random_element()
[zeta36^4]
sage: DirichletGroup(20).random_element()
[-1, 1]
sage: DirichletGroup(60).random_element()
[1, -1, 1]
```

unit_gens()

Returns the minimal generators for the units of $(\mathbf{Z}/N\mathbf{Z})^*$, where N is the modulus of self.

EXAMPLES:

```
sage: DirichletGroup(37).unit_gens()
[2]
sage: DirichletGroup(20).unit_gens()
[11, 17]
sage: DirichletGroup(60).unit_gens()
[31, 41, 37]
sage: DirichletGroup(20,QQ).unit_gens()
[11, 17]
```

zeta()

Returns the chosen root zeta of unity in the base ring R.

EXAMPLES:

```
sage: DirichletGroup(37).zeta()
zeta36
sage: DirichletGroup(20).zeta()
zeta4
sage: DirichletGroup(60).zeta()
zeta4
sage: DirichletGroup(60,QQ).zeta()
-1
sage: DirichletGroup(60, GF(25,'a')).zeta()
2
```

zeta_order()

Returns the order of the chosen root zeta of unity in the base ring R.

EXAMPLES:

```
sage: DirichletGroup(20).zeta_order()
4
sage: DirichletGroup(60).zeta_order()
4
sage: DirichletGroup(60, GF(25,'a')).zeta_order()
4
sage: DirichletGroup(19).zeta_order()
18
```

TrivialCharacter(*N, base_ring=Rational Field*)

Return the trivial character of the given modulus, with values in the given base ring.

EXAMPLE:

```
sage: t = trivial_character(7)
sage: [t(x) for x in [0..20]]
[0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
sage: t(1).parent()
Rational Field
sage: trivial_character(7, Integers(3))(1).parent()
Ring of integers modulo 3
```

is_DirichletCharacter(*x*)
Return True if x is of type DirichletCharacter.

EXAMPLES:

```
sage: from sage.modular.dirichlet import is_DirichletCharacter
sage: is_DirichletCharacter(trivial_character(3))
True
sage: is_DirichletCharacter([1])
False
```

is_DirichletGroup(*x*)
Returns True if x is a Dirichlet group.

EXAMPLES:

```
sage: from sage.modular.dirichlet import is_DirichletGroup
sage: is_DirichletGroup(DirichletGroup(11))
True
sage: is_DirichletGroup(11)
False
sage: is_DirichletGroup(DirichletGroup(11).0)
False
```

kronecker_character(*d*)
Returns the quadratic Dirichlet character (d/.) of minimal conductor.

EXAMPLES:

```
sage: kronecker_character(97*389*997^2)
[-1, -1]
```

```
sage: a = kronecker_character(1)
sage: b = DirichletGroup(2401,QQ)(a)     # NOTE -- over QQ!
sage: b.modulus()
2401
```

AUTHORS:

- Jon Hanke (2006-08-06)

kronecker_character_upside_down(*d*)
Returns the quadratic Dirichlet character (./d) of conductor d, for d0.

EXAMPLES:

```
sage: kronecker_character_upside_down(97*389*997^2)
[-1, -1, 1]
```

AUTHORS:

- Jon Hanke (2006-08-06)

trivial_character(*N, base_ring=Rational Field*)
Return the trivial character of the given modulus, with values in the given base ring.

EXAMPLE:

```
sage: t = trivial_character(7)
sage: [t(x) for x in [0..20]]
[0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
sage: t(1).parent()
Rational Field
sage: trivial_character(7, Integers(3))(1).parent()
Ring of integers modulo 3
```

46.2 The set $\mathbb{P}^1(\mathbb{Q})$ of cusps

EXAMPLES:

```
sage: Cusps
Set P^1(QQ) of all cusps


sage: Cusp(oo)
Infinity
```

class Cusp(*a, b=None, parent=None, check=True*)

A cusp.

A cusp is either a rational number or infinity, i.e., an element of the projective line over Q. A Cusp is stored as a pair (a,b), where gcd(a,b)=1 and a,b are of type Integer.

EXAMPLES:

```
sage: a = Cusp(2/3); b = Cusp(oo)
sage: a.parent()
Set P^1(QQ) of all cusps
sage: a.parent() is b.parent()
True
```

apply(*g*)

Return g(self), where g=[a,b,c,d] is a list of length 4, which we view as a linear fractional transformation.

EXAMPLES: Apply the identity matrix:

```
sage: Cusp(0).apply([1,0,0,1])
0
sage: Cusp(0).apply([0,-1,1,0])
Infinity
sage: Cusp(0).apply([1,-3,0,1])
-3
```

denominator()

Return the denominator of the cusp a/b.

EXAMPLES:

```
sage: x=Cusp(6,9); x
2/3
sage: x.denominator()
3
sage: Cusp(oo).denominator()
0
sage: Cusp(-5/10).denominator()
2
```

galois_action(*t, N*)

Suppose this cusp is α, G is a congruence subgroup of level N, and σ is the automorphism in the Galois group of $\mathbf{Q}(\zeta_N)/\mathbf{Q}$ that sends ζ_N to ζ_N^t. Then this function computes a cusp β such that $\sigma([\alpha]) = [\beta]$, where $[\alpha]$ is the equivalence class of α modulo G.

INPUT:

- t – integer that is coprime to N
- N – positive integer (level)

OUTPUT:

- a cusp

EXAMPLES:

```
sage: Cusp(1/10).galois_action(3, 50)
1/170
sage: Cusp(oo).galois_action(3, 50)
Infinity
sage: Cusp(0).galois_action(3, 50)
0
```

Here we compute explicitly the permutations of the action for t=3 on cusps for Gamma0(50):

```
sage: N = 50; t=3; G = Gamma0(N); C = G.cusps()
sage: cl = lambda z: exists(C, lambda y:y.is_gamma0_equiv(z, N))[1]
sage: for i in range(5): print i, t^i, [cl(alpha.galois_action(t^i,N)) for alpha in C]
0 1 [0, 1/25, 1/10, 1/5, 3/10, 2/5, 1/2, 3/5, 7/10, 4/5, 9/10, Infinity]
1 3 [0, 1/25, 7/10, 2/5, 1/10, 4/5, 1/2, 1/5, 9/10, 3/5, 3/10, Infinity]
2 9 [0, 1/25, 9/10, 4/5, 7/10, 3/5, 1/2, 2/5, 3/10, 1/5, 1/10, Infinity]
3 27 [0, 1/25, 3/10, 3/5, 9/10, 1/5, 1/2, 4/5, 1/10, 2/5, 7/10, Infinity]
4 81 [0, 1/25, 1/10, 1/5, 3/10, 2/5, 1/2, 3/5, 7/10, 4/5, 9/10, Infinity]
```

REFERENCES:

- Section 1.3 of Glenn Stevens, "Arithmetic on Modular Curves"
- There is a long comment about our algorithm in the source code for this function.

WARNING: In some cases N must fit in a long long, i.e., there are cases where this algorithm isn't fully implemented.

AUTHORS:

- William Stein, 2009-04-18

is_gamma0_equiv(*other, N, transformation=False*)

Return whether self and other are equivalent modulo the action of $\Gamma_0(N)$ via linear fractional transformations.

INPUT:

- `other` - Cusp
- `N` - an integer (specifies the group Gamma_0(N))
- `transformation` - bool (default: False), if True, also return upper left entry of a matrix in Gamma_0(N) that sends self to other.

OUTPUT:

- `bool` - True if self and other are equivalent
- `integer` - returned only if transformation is True

EXAMPLES:

```
sage: x = Cusp(2,3)
sage: y = Cusp(4,5)
sage: x.is_gamma0_equiv(y, 2)
```

```
True
sage: x.is_gamma0_equiv(y, 2, True)
(True, 1)
sage: x.is_gamma0_equiv(y, 3)
False
sage: x.is_gamma0_equiv(y, 3, True)
(False, None)
sage: Cusp(1,0)
Infinity
sage: z = Cusp(1,0)
sage: x.is_gamma0_equiv(z, 3, True)
(True, 2)
```

ALGORITHM: See Proposition 2.2.3 of Cremona's book "Algorithms for Modular Elliptic Curves", or Prop 2.27 of Stein's Ph.D. thesis.

is_gamma1_equiv(*other, N*)

Return whether self and other are equivalent modulo the action of Gamma_1(N) via linear fractional transformations.

INPUT:

- `other` - Cusp
- `N` - an integer (specifies the group Gamma_1(N))

OUTPUT:

- `bool` - True if self and other are equivalent
- `int` - 0, 1 or -1, gives further information about the equivalence: If the two cusps are u1/v1 and u2/v2, then they are equivalent if and only if v1 = v2 (mod N) and u1 = u2 (mod gcd(v1,N)) or v1 = -v2 (mod N) and u1 = -u2 (mod gcd(v1,N)) The sign is +1 for the first and -1 for the second. If the two cusps are not equivalent then 0 is returned.

EXAMPLES:

```
sage: x = Cusp(2,3)
sage: y = Cusp(4,5)
sage: x.is_gamma1_equiv(y,2)
(True, 1)
sage: x.is_gamma1_equiv(y,3)
(False, 0)
sage: z = Cusp(QQ(x) + 10)
sage: x.is_gamma1_equiv(z,10)
(True, 1)
sage: z = Cusp(1,0)
sage: x.is_gamma1_equiv(z, 3)
(True, -1)
sage: Cusp(0).is_gamma1_equiv(oo, 1)
(True, 1)
sage: Cusp(0).is_gamma1_equiv(oo, 3)
(False, 0)
```

is_gamma_h_equiv(*other, G*)

Return a pair (b, t), where b is True or False as self and other are equivalent under the action of G, and t is 1 or -1, as described below.

Two cusps $u1/v1$ and $u2/v2$ are equivalent modulo Gamma_H(N) if and only if $v1 = h*v2 (\mathrm{mod} N)$ and $u1 = h^{(-1)}*u2 (\mathrm{mod} gcd(v1,N))$ or $v1 = -h*v2 (mod N)$ and $u1 = -h^{(-1)}*u2 (\mathrm{mod} gcd(v1,N))$ for some $h \in H$. Then t is 1 or -1 as c and c' fall into the first or second case, respectively.

INPUT:

- `other` - Cusp

- G - a congruence subgroup Gamma_H(N)

OUTPUT:

- bool - True if self and other are equivalent
- int - -1, 0, 1; extra info

EXAMPLES:

```
sage: x = Cusp(2,3)
sage: y = Cusp(4,5)
sage: x.is_gamma_h_equiv(y,GammaH(13,[2]))
(True, 1)
sage: x.is_gamma_h_equiv(y,GammaH(13,[5]))
(False, 0)
sage: x.is_gamma_h_equiv(y,GammaH(5,[]))
(False, 0)
sage: x.is_gamma_h_equiv(y,GammaH(23,[4]))
(True, -1)
```

Enumerating the cusps for a space of modular symbols uses this function.

```
sage: G = GammaH(25,[6]) ; M = G.modular_symbols() ; M
Modular Symbols space of dimension 11 for Congruence Subgroup Gamma_H(25) with H generated b
sage: M.cusps()
[37/75, 1/2, 31/125, 1/4, -2/5, 2/5, -1/5, 1/10, -3/10, 1/15, 7/15, 9/20]
sage: len(M.cusps())
12
```

This is always one more than the associated space of weight 2 Eisenstein series.

```
sage: G.dimension_eis(2)
11
sage: M.cuspidal_subspace()
Modular Symbols subspace of dimension 0 of Modular Symbols space of dimension 11 for Congrue
sage: G.dimension_cusp_forms(2)
0
```

is_infinity()

Returns True if this is the cusp infinity.

EXAMPLES:

```
sage: Cusp(3/5).is_infinity()
False
sage: Cusp(1,0).is_infinity()
True
sage: Cusp(0,1).is_infinity()
False
```

numerator()

Return the numerator of the cusp a/b.

EXAMPLES:

```
sage: x=Cusp(6,9); x
2/3
sage: x.numerator()
2
sage: Cusp(oo).numerator()
1
sage: Cusp(-5/10).numerator()
-1
```

class Cusps_class()
The set of cusps.

EXAMPLES:

```
sage: C = Cusps; C
Set P^1(QQ) of all cusps
sage: loads(C.dumps()) == C
True
```

46.3 Dimensions of spaces of modular forms

AUTHORS:

- William Stein
- Jordi Quer

ACKNOWLEDGEMENT: The dimension formulas and implementations in this module grew out of a program that Bruce Kaskel wrote (around 1996) in PARI, which Kevin Buzzard subsequently extended. I (William Stein) then implemented it in C++ for Hecke. I also implemented it in Magma. Also, the functions for dimensions of spaces with nontrivial character are based on a paper (that has no proofs) by Cohen and Oesterle (Springer Lecture notes in math, volume 627, pages 69-78). The formulas for $\Gamma_H(N)$ were found and implemented by Jordi Quer.

The formulas here are more complete than in Hecke or Magma.

Currently the input to each function below is an integer and either a Dirichlet character ε or a finite index subgroup of $\mathrm{SL}_2(\mathbf{Z})$. If the input is a Dirichlet character ε, the dimensions are for subspaces of $M_k(\Gamma_1(N), \varepsilon)$, where N is the modulus of ε.

These functions mostly call the methods dimension_cusp_forms, dimension_modular_forms and so on of the corresponding congruence subgroup classes.

CO_delta(*r, p, N, eps*)
This is used as an intermediate value in computations related to the paper of Cohen-Oesterle.

INPUT:

- `r` - positive integer
- `p` - a prime
- `N` - positive integer
- `eps` - character

OUTPUT: element of the base ring of the character

EXAMPLES:

```
sage: G.<eps> = DirichletGroup(7)
sage: sage.modular.dims.CO_delta(1,5,7,eps^3)
2
```

CO_nu(*r, p, N, eps*)
This is used as an intermediate value in computations related to the paper of Cohen-Oesterle.

INPUT:

- `r` - positive integer
- `p` - a prime

- N - positive integer
- eps - character

OUTPUT: element of the base ring of the character

EXAMPLES:

```
sage: G.<eps> = DirichletGroup(7)
sage: G.<eps> = DirichletGroup(7)
sage: sage.modular.dims.CO_nu(1,7,7,eps)
-1
```

CohenOesterle(*eps, k*)

Compute the Cohen-Oesterle function associate to eps, k. This is a summand in the formula for the dimension of the space of cusp forms of weight 2 with character ε.

INPUT:

- eps - Dirichlet character
- k - integer

OUTPUT: element of the base ring of eps.

EXAMPLES:

```
sage: G.<eps> = DirichletGroup(7)
sage: sage.modular.dims.CohenOesterle(eps, 2)
-2/3
sage: sage.modular.dims.CohenOesterle(eps, 4)
-1
```

dimension_cusp_forms(*X, k=2*)

The dimension of the space of cusp forms for the given congruence subgroup or Dirichlet character.

INPUT:

- X - congruence subgroup or Dirichlet character or integer
- k - weight (integer)

EXAMPLES:

```
sage: dimension_cusp_forms(5,4)
1


sage: dimension_cusp_forms(Gamma0(11),2)
1
sage: dimension_cusp_forms(Gamma1(13),2)
2


sage: dimension_cusp_forms(DirichletGroup(13).0^2,2)
1
sage: dimension_cusp_forms(DirichletGroup(13).0,3)
1


sage: dimension_cusp_forms(Gamma0(11),2)
1
sage: dimension_cusp_forms(Gamma0(11),0)
0
sage: dimension_cusp_forms(Gamma0(1),12)
```

```
1
sage: dimension_cusp_forms(Gamma0(1),2)
0
sage: dimension_cusp_forms(Gamma0(1),4)
0

sage: dimension_cusp_forms(Gamma0(389),2)
32
sage: dimension_cusp_forms(Gamma0(389),4)
97
sage: dimension_cusp_forms(Gamma0(2005),2)
199
sage: dimension_cusp_forms(Gamma0(11),1)
0

sage: dimension_cusp_forms(Gamma1(11),2)
1
sage: dimension_cusp_forms(Gamma1(1),12)
1
sage: dimension_cusp_forms(Gamma1(1),2)
0
sage: dimension_cusp_forms(Gamma1(1),4)
0

sage: dimension_cusp_forms(Gamma1(389),2)
6112
sage: dimension_cusp_forms(Gamma1(389),4)
18721
sage: dimension_cusp_forms(Gamma1(2005),2)
159201

sage: dimension_cusp_forms(Gamma1(11),1)
0

sage: e = DirichletGroup(13).0
sage: e.order()
12
sage: dimension_cusp_forms(e,2)
0
sage: dimension_cusp_forms(e^2,2)
1
```

dimension_eis(*X*, *k=2*)

The dimension of the space of eisenstein series for the given congruence subgroup.

INPUT:

- X - congruence subgroup or Dirichlet character or integer
- k - weight (integer)

EXAMPLES:

```
sage: dimension_eis(5,4)
2
```

```
sage: dimension_eis(Gamma0(11),2)
1
sage: dimension_eis(Gamma1(13),2)
11
sage: dimension_eis(Gamma1(2006),2)
3711

sage: e = DirichletGroup(13).0
sage: e.order()
12
sage: dimension_eis(e,2)
0
sage: dimension_eis(e^2,2)
2

sage: e = DirichletGroup(13).0
sage: e.order()
12
sage: dimension_eis(e,2)
0
sage: dimension_eis(e^2,2)
2
sage: dimension_eis(e,13)
2

sage: G = DirichletGroup(20)
sage: dimension_eis(G.0,3)
4
sage: dimension_eis(G.1,3)
6
sage: dimension_eis(G.1^2,2)
6

sage: G = DirichletGroup(200)
sage: e = prod(G.gens(), G(1))
sage: e.conductor()
200
sage: dimension_eis(e,2)
4

sage: dimension_modular_forms(Gamma1(4), 11)
6
```

dimension_modular_forms(*X*, *k=2*)

The dimension of the space of cusp forms for the given congruence subgroup (either $\Gamma_0(N)$, $\Gamma_1(N)$, or $\Gamma_H(N)$) or Dirichlet character.

INPUT:

- X - congruence subgroup or Dirichlet character
- k - weight (integer)

EXAMPLES:

```
sage: dimension_modular_forms(Gamma0(11),2)
2
sage: dimension_modular_forms(Gamma0(11),0)
1
sage: dimension_modular_forms(Gamma1(13),2)
13
sage: dimension_modular_forms(GammaH(11, [10]), 2)
10
sage: dimension_modular_forms(GammaH(11, [10]))
10
sage: dimension_modular_forms(GammaH(11, [10]), 4)
20
sage: e = DirichletGroup(20).1
sage: dimension_modular_forms(e,3)
9
sage: dimension_cusp_forms(e,3)
3
sage: dimension_eis(e,3)
6
sage: dimension_modular_forms(11,2)
2
```

dimension_new_cusp_forms (*X, k=2, p=0*)

Return the dimension of the new (or p-new) subspace of cusp forms for the character or group X.

INPUT:

- X - integer, congruence subgroup or Dirichlet character
- k - weight (integer)
- p - 0 or a prime

EXAMPLES:

```
sage: dimension_new_cusp_forms(100,2)
1


sage: dimension_new_cusp_forms(Gamma0(100),2)
1
sage: dimension_new_cusp_forms(Gamma0(100),4)
5


sage: dimension_new_cusp_forms(Gamma1(100),2)
141
sage: dimension_new_cusp_forms(Gamma1(100),4)
463


sage: dimension_new_cusp_forms(DirichletGroup(100).1^2,2)
2
sage: dimension_new_cusp_forms(DirichletGroup(100).1^2,4)
8


sage: sum(dimension_new_cusp_forms(e,3) for e in DirichletGroup(30))
12
sage: dimension_new_cusp_forms(Gamma1(30),3)
12
```

eisen(*p*)

Return the Eisenstein number n which is the numerator of $(p-1)/12$.

INPUT:

- p - a prime

OUTPUT: Integer

EXAMPLES:

```
sage: [(p,sage.modular.dims.eisen(p)) for p in prime_range(24)]
[(2, 1), (3, 1), (5, 1), (7, 1), (11, 5), (13, 1), (17, 4), (19, 3), (23, 11)]
```

sturm_bound(*level, weight=2*)

Returns the Sturm bound for modular forms with given level and weight. For more details, see the documentation for the sturm_bound method of sage.modular.arithgroup.CongruenceSubgroup objects.

INPUT:

- level - an integer (interpreted as a level for Gamma0) or a congruence subgroup
- weight - an integer ≥ 2 (default: 2)

EXAMPLES:

```
sage: sturm_bound(11,2)
2
sage: sturm_bound(389,2)
65
sage: sturm_bound(1,12)
1
sage: sturm_bound(100,2)
30
sage: sturm_bound(1,36)
3
sage: sturm_bound(11)
2
```

46.4 Conjectural Slopes of Hecke Polynomial

Interface to Kevin Buzzard's PARI program for computing conjectural slopes of characteristic polynomials of Hecke operators.

AUTHORS:

- William Stein (2006-03-05): Sage interface
- Kevin Buzzard: PARI program that implements underlying functionality

buzzard_tpslopes(*p, N, kmax*)

Returns a vector of length kmax, whose k'th entry ($0 \leq k \leq k_{max}$) is the conjectural sequence of valuations of eigenvalues of T_p on forms of level N, weight k, and trivial character.

This conjecture is due to Kevin Buzzard, and is only made assuming that p does not divide N and if p is $\Gamma_0(N)$-regular.

EXAMPLES:

```
sage: c = buzzard_tpslopes(2,1,50)
sage: c[50]
[4, 8, 13]
```

Hence Buzzard would conjecture that the 2-adic valuations of the eigenvalues of T_2 on cusp forms of level 1 and weight 50 are $[4, 8, 13]$, which indeed they are, as one can verify by an explicit computation using, e.g., modular symbols:

```
sage: M = ModularSymbols(1,50, sign=1).cuspidal_submodule()
sage: T = M.hecke_operator(2)
sage: f = T.charpoly('x')
sage: f.newton_slopes(2)
[13, 8, 4]
```

AUTHORS:

- Kevin Buzzard: several GP/PARI scripts
- William Stein (2006-03-17): small Sage wrapper of Buzzard's scripts

gp()

Return a copy of the GP interpreter with the appropriate files loaded.

EXAMPLE:

```
sage: sage.modular.buzzard.gp()
GP/PARI interpreter
```

46.5 Eta-products on modular curves $X_0(N)$.

This package provides a class for representing eta-products, which are meromorphic functions on modular curves of the form

$$\prod_{d|N} \eta(q^d)^{r_d}$$

where $\eta(q)$ is Dirichlet's eta function $q^{1/24}\prod_{n=1}^{\infty}(1-q^n)$. These are useful for obtaining explicit models of modular curves.

See trac ticket #3934 for background.

AUTHOR:

- David Loeffler (2008-08-22): initial version

AllCusps(N)

Return a list of CuspFamily objects corresponding to the cusps of $X_0(N)$.

INPUT:

- N - (integer): the level

EXAMPLES:

```
sage: AllCusps(18)
[(Inf), (c_{2}), (c_{3,1}), (c_{3,2}), (c_{6,1}), (c_{6,2}), (c_{9}), (0)]
```

class CuspFamily(*N, width, label=None*)

A family of elliptic curves parametrising a region of $X_0(N)$.

level()

The level of this cusp.

EXAMPLES:

```
sage: e = CuspFamily(10, 1)
sage: e.level()
10
```

sage_cusp()

Return the corresponding element of $\mathbb{P}^1(\mathbf{Q})$.

EXAMPLE:

```
sage: CuspFamily(10, 1).sage_cusp() # not implemented
Infinity
```

width()

The width of this cusp.

EXAMPLES:

```
sage: e = CuspFamily(10, 1)
sage: e.width()
1
```

EtaGroup(*level*)

Create the group of eta products of the given level.

EXAMPLES:

```
sage: EtaGroup(12)
Group of eta products on X_0(12)
sage: EtaGroup(1/2)
...
TypeError: Level (=1/2) must be a positive integer
sage: EtaGroup(0)
...
ValueError: Level (=0) must be a positive integer
```

class EtaGroupElement(*parent, rdict*)

degree()

Return the degree of self as a map $X_0(N) \to \mathbb{P}^1$, which is equal to the sum of all the positive coefficients in the divisor of self.

EXAMPLES:

```
sage: e = EtaProduct(12, {1:-336, 2:576, 3:696, 4:-216, 6:-576, 12:-144})
sage: e.degree()
230
```

divisor()

Return the divisor of self, as a formal sum of CuspFamily objects.

EXAMPLES:

```
sage: e = EtaProduct(12, {1:-336, 2:576, 3:696, 4:-216, 6:-576, 12:-144})
sage: e.divisor() # FormalSum seems to print things in a random order?
-131*(Inf) - 50*(c_{2}) + 11*(0) + 50*(c_{6}) + 169*(c_{4}) - 49*(c_{3})
sage: e = EtaProduct(2^8, {8:1,32:-1})
```

```
sage: e.divisor() # random
-(c_{2}) - (Inf) - (c_{8,2}) - (c_{8,3}) - (c_{8,4}) - (c_{4,2}) - (c_{8,1}) - (c_{4,1}) + (
```

level()

Return the level of this eta product.

EXAMPLES:

```
sage: e = EtaProduct(3, {3:12, 1:-12})
sage: e.level()
3
sage: EtaProduct(12, {6:6, 2:-6}).level() # not the lcm of the d's
12
sage: EtaProduct(36, {6:6, 2:-6}).level() # not minimal
36
```

order_at_cusp(*cusp*)

Return the order of vanishing of self at the given cusp.

INPUT:

- cusp - a CuspFamily object

OUTPUT:

- an integer

EXAMPLES:

```
sage: e = EtaProduct(2, {2:24, 1:-24})
sage: e.order_at_cusp(CuspFamily(2, 1)) # cusp at infinity
1
sage: e.order_at_cusp(CuspFamily(2, 2)) # cusp 0
-1
```

qexp(*n*)

The q-expansion of self at the cusp at infinity.

INPUT:

- n (integer): number of terms to calculate

OUTPUT:

- a power series over $\mathbf{Z}$ in the variable q, with a *relative* precision of $1 + O(q^n)$.

ALGORITHM: Calculates eta to (n/m) terms, where m is the smallest integer dividing self.level() such that self.r(m) != 0. Then multiplies.

EXAMPLES:

```
sage: EtaProduct(36, {6:6, 2:-6}).qexp(10)
q + 6*q^3 + 27*q^5 + 92*q^7 + 279*q^9 + O(q^11)
sage: R.<q> = ZZ[[]]
sage: EtaProduct(2,{2:24,1:-24}).qexp(100) == delta_qexp(101)(q^2)/delta_qexp(101)(q)
True
```

r(*d*)

Return the exponent r_d of $\eta(q^d)$ in self.

EXAMPLES:

```
sage: e = EtaProduct(12, {2:24, 3:-24})
sage: e.r(3)
-24
sage: e.r(4)
0
```

class EtaGroup_class(*level*)

The group of eta products of a given level under multiplication.

basis(*reduce=True*)

Produce a basis for the free abelian group of eta-products of level N (under multiplication), attempting to find basis vectors of the smallest possible degree.

INPUT:

- `reduce` - a boolean (default True) indicating whether or not to apply LLL-reduction to the calculated basis

EXAMPLE:

```
sage: EtaGroup(5).basis()
[Eta product of level 5 : (eta_1)^6 (eta_5)^-6]
sage: EtaGroup(12).basis()
[Eta product of level 12 : (eta_1)^2 (eta_2)^1 (eta_3)^2 (eta_4)^-1 (eta_6)^-7 (eta_12)^3,
Eta product of level 12 : (eta_1)^-2 (eta_2)^3 (eta_3)^6 (eta_4)^-1 (eta_6)^-9 (eta_12)^3,
Eta product of level 12 : (eta_1)^-3 (eta_2)^2 (eta_3)^1 (eta_4)^-1 (eta_6)^-2 (eta_12)^3,
Eta product of level 12 : (eta_1)^1 (eta_2)^-1 (eta_3)^-3 (eta_4)^-2 (eta_6)^7 (eta_12)^-2,
Eta product of level 12 : (eta_1)^-6 (eta_2)^9 (eta_3)^2 (eta_4)^-3 (eta_6)^-3 (eta_12)^1]
sage: EtaGroup(12).basis(reduce=False) # much bigger coefficients
[Eta product of level 12 : (eta_2)^24 (eta_12)^-24,
Eta product of level 12 : (eta_1)^-336 (eta_2)^576 (eta_3)^696 (eta_4)^-216 (eta_6)^-576 (et
Eta product of level 12 : (eta_1)^-8 (eta_2)^-2 (eta_6)^2 (eta_12)^8,
Eta product of level 12 : (eta_1)^1 (eta_2)^9 (eta_3)^13 (eta_4)^-4 (eta_6)^-15 (eta_12)^-4,
Eta product of level 12 : (eta_1)^15 (eta_2)^-24 (eta_3)^-29 (eta_4)^9 (eta_6)^24 (eta_12)^5
```

ALGORITHM: An eta product of level N is uniquely determined by the integers r_d for $d|N$ with $d < N$, since $\sum_{d|N} r_d = 0$. The valid r_d are those that satisfy two congruences modulo 24, and one congruence modulo 2 for every prime divisor of N. We beef up the congruences modulo 2 to congruences modulo 24 by multiplying by 12. To calculate the kernel of the ensuing map $\mathbf{Z}^m \to (\mathbf{Z}/24\mathbf{Z})^n$ we lift it arbitrarily to an integer matrix and calculate its Smith normal form. This gives a basis for the lattice.

This lattice typically contains "large" elements, so by default we pass it to the reduce_basis() function which performs LLL-reduction to give a more manageable basis.

level()

Return the level of self. EXAMPLES:

```
sage: EtaGroup(10).level()
10
```

reduce_basis(*long_etas*)

Produce a more manageable basis via LLL-reduction.

INPUT:

- `long_etas` - a list of EtaGroupElement objects (which should all be of the same level)

OUTPUT:

- a new list of EtaGroupElement objects having hopefully smaller norm

ALGORITHM: We define the norm of an eta-product to be the L^2 norm of its divisor (as an element of the free $\mathbf{Z}$-module with the cusps as basis and the standard inner product). Applying LLL-reduction to this gives a basis of hopefully more tractable elements. Of course we'd like to use the L^1 norm as this is just twice the degree, which is a much more natural invariant, but L^2 norm is easier to work with!

EXAMPLES:

```
sage: EtaGroup(4).reduce_basis([ EtaProduct(4, {1:8,2:24,4:-32}), EtaProduct(4, {1:8, 4:-8})
[Eta product of level 4 : (eta_1)^8 (eta_4)^-8,
Eta product of level 4 : (eta_1)^-8 (eta_2)^24 (eta_4)^-16]
```

EtaProduct (*level, dict*)

Create an EtaGroupElement object representing the function $\prod_{d|N} \eta(q^d)^{r_d}$. Checks the critera of Ligozat to ensure that this product really is the q-expansion of a meromorphic function on X_0(N).

INPUT:

- `level` - (integer): the N such that this eta product is a function on X_0(N).
- `dict` - (dictionary): a dictionary indexed by divisors of N such that the coefficient of $\eta(q^d)$ is r[d]. Only nonzero coefficients need be specified. If Ligozat's criteria are not satisfied, a ValueError will be raised.

OUTPUT:

- an EtaGroupElement object, whose parent is the EtaGroup of level N and whose coefficients are the given dictionary.

Note: The dictionary dict does not uniquely specify N. It is possible for two EtaGroupElements with different N's to be created with the same dictionary, and these represent different objects (although they will have the same q-expansion at the cusp ∞).

EXAMPLES:

```
sage: EtaProduct(3, {3:12, 1:-12})
Eta product of level 3 : (eta_1)^-12 (eta_3)^12
sage: EtaProduct(3, {3:6, 1:-6})
...
ValueError: sum d r_d (=12) is not 0 mod 24
sage: EtaProduct(3, {4:6, 1:-6})
...
ValueError: 4 does not divide 3
```

eta_poly_relations (*eta_elements, degree, labels=, ['x1', 'x2'], verbose=False*)

Find polynomial relations between eta products.

INPUTS:

- `eta_elements` - (list): a list of EtaGroupElement objects. Not implemented unless this list has precisely two elements. degree
- `degree` - (integer): the maximal degree of polynomial to look for.
- `labels` - (list of strings): labels to use for the polynomial returned.
- `verbose` ` - (boolean, default False): if True, prints information as it goes.

OUTPUTS: a list of polynomials which is a Groebner basis for the part of the ideal of relations between eta_elements which is generated by elements up to the given degree; or None, if no relations were found.

ALGORITHM: An expression of the form $\sum_{0 \le i,j \le d} a_{ij} x^i y^j$ is zero if and only if it vanishes at the cusp infinity to degree at least $v = d(deg(x) + deg(y))$. For all terms up to q^v in the q-expansion of this expression to be zero is a system of $v + k$ linear equations in d^2 coefficients, where k is the number of nonzero negative coefficients that can appear.

Solving these equations and calculating a basis for the solution space gives us a set of polynomial relations, but this is generally far from a minimal generating set for the ideal, so we calculate a Groebner basis.

As a test, we calculate five extra terms of q-expansion and check that this doesn't change the answer.

EXAMPLES:

```
sage: t = EtaProduct(26, {2:2,13:2,26:-2,1:-2})
sage: u = EtaProduct(26, {2:4,13:2,26:-4,1:-2})
sage: eta_poly_relations([t, u], 3)
sage: eta_poly_relations([t, u], 4)
[x1^3*x2 - 13*x1^3 - 4*x1^2*x2 - 4*x1*x2 - x2^2 + x2]
```

Use verbose=True to see the details of the computation:

```
sage: eta_poly_relations([t, u], 3, verbose=True)
Trying to find a relation of degree 3
Lowest order of a term at infinity = -12
Highest possible degree of a term = 15
Trying all coefficients from q^-12 to q^15 inclusive
No polynomial relation of order 3 valid for 28 terms
Check: Trying all coefficients from q^-12 to q^20 inclusive
No polynomial relation of order 3 valid for 33 terms


sage: eta_poly_relations([t, u], 4, verbose=True)
Trying to find a relation of degree 4
Lowest order of a term at infinity = -16
Highest possible degree of a term = 20
Trying all coefficients from q^-16 to q^20 inclusive
Check: Trying all coefficients from q^-16 to q^25 inclusive
[x1^3*x2 - 13*x1^3 - 4*x1^2*x2 - 4*x1*x2 - x2^2 + x2]
```

num_cusps_of_width(*N, d*)

Return the number of cusps on $X_0(N)$ of width d.

INPUT:

- N - (integer): the level
- d - (integer): an integer dividing N, the cusp width

EXAMPLES:

```
sage: [num_cusps_of_width(18,d) for d in divisors(18)]
[1, 1, 2, 2, 1, 1]
```

qexp_eta(*ps_ring, n*)

Return the q-expansion of $\eta(q)/q^{1/24}$, where $\eta(q)$ is Dedekind's function

$$\eta(q) = q^{1/24}\prod_{i=1}^{\infty}(1-q^i)$$

as an element of ps_ring, to precision n. Completely naive algorithm.

INPUT:

- ps_ring - (PowerSeriesRing): a power series ring - we pass this as an argument as we frequently need to create multiple series in the same ring.
- n - (integer): the number of terms to compute.

OUTPUT: An element of ps_ring which is the q-expansion of $\eta(q)/q^{1/24}$ truncated to n terms.

ALGORITHM: Multiply out the product $\prod_{i=1}^{n}(1-q^i)$. Could perhaps be speeded up by using the identity

$$\eta(q) = q^{1/24}(1+\sum_{i\geq 1}(-1)^n(q^{n(3n+1)/2}+q^{n(3n-1)/2}),$$

but I'm lazy.

EXAMPLES:

```
sage: qexp_eta(ZZ[['q']], 100)
1 - q - q^2 + q^5 + q^7 - q^12 - q^15 + q^22 + q^26 - q^35 - q^40 + q^51 + q^57 - q^70 - q^77 +
```

46.6 The space of p-adic weights

A p-adic weight is a continuous character $\mathbf{Z}_p^\times \to \mathbf{C}_p^\times$. These are the $\mathbf{C}_p$-points of a rigid space over $\mathbf{Q}_p$, which is isomorphic to a disjoint union of copies (indexed by $(\mathbf{Z}/p\mathbf{Z})^\times$) of the open unit p-adic disc.

Sage supports both "classical points", which are determined by the data of a Dirichlet character modulo p^m for some m and an integer k (corresponding to the character $z \mapsto z^k\chi(z)$) and "non-classical points" which are determined by the data of an element of $(\mathbf{Z}/p\mathbf{Z})^\times$ and an element $w \in \mathbf{C}_p$ with $|w-1| < 1$.

EXAMPLES:

```
sage: W = pAdicWeightSpace(17)
sage: W
Space of 17-adic weight-characters defined over '17-adic Field with capped relative precision 20'
sage: L = Qp(17).extension(x^2 - 17, names='a'); L.rename('L')
sage: W.base_extend(L)
Space of 17-adic weight-characters defined over 'L'
```

We create a simple element of $\mathcal{W}$: the algebraic character, $x \mapsto x^6$:

```
sage: kappa = W(6)
sage: kappa(5)
15625
sage: kappa(5) == 5^6
True
```

A locally algebraic character, $x \mapsto x^6\chi(x)$ for χ a Dirichlet character mod p:

```
sage: kappa2 = W(6, DirichletGroup(17, Qp(17)).0^8)
sage: kappa2(5) == -5^6
True
sage: kappa2(13) == 13^6
True
```

A non-locally-algebraic character, sending the generator 18 of $1 + 17\mathbf{Z}_{17}$ to 35 and acting as $\mu \mapsto \mu^4$ on the group of 16th roots of unity:

```
sage: kappa3 = W(35 + O(17^20), 4, algebraic=False)
sage: kappa3(2)
16 + 8*17 + ... + O(17^20)
```

AUTHORS:

- David Loeffler (2008-9)

class **AlgebraicWeight** (*parent, k, chi=None*)

A point in weight space corresponding to a locally algebraic character, of the form $x \mapsto \chi(x)x^k$ where k is an integer and χ is a Dirichlet character modulo p^n for some n.

TESTS:

```
sage: w = pAdicWeightSpace(23)(12, DirichletGroup(23, QQ).0) # exact
sage: w == loads(dumps(w))
True
sage: w = pAdicWeightSpace(23)(12, DirichletGroup(23, Qp(23)).0) # inexact
sage: w == loads(dumps(w))
```

```
True
sage: w is loads(dumps(w)) # elements are not globally unique
False
```

Lvalue()

Return the value of the p-adic L-function of $\mathbf{Q}$ evaluated at this weight-character. If the character is $x \mapsto x^k\chi(x)$ where $k > 0$ and χ has conductor a power of p, this is an element of the number field generated by the values of χ, equal to the value of the complex L-function $L(1-k,\chi)$. If χ is trivial, it is equal to $(1-p^{k-1})\zeta(1-k)$.

At present this is not implemented in any other cases, except the trivial character (for which the value is ∞).

TODO: Implement this more generally using the Amice transform machinery in sage/schemes/elliptic_curves/padic_lseries.py, which should clearly be factored out into a separate class.

EXAMPLES:

```
sage: pAdicWeightSpace(7)(4).Lvalue() == (1 - 7^3)*zeta__exact(-3)
True
sage: pAdicWeightSpace(7)(5, DirichletGroup(7, Qp(7)).0^4).Lvalue()
0
sage: pAdicWeightSpace(7)(6, DirichletGroup(7, Qp(7)).0^4).Lvalue()
1 + 2*7 + 7^2 + 3*7^3 + 3*7^5 + 4*7^6 + 2*7^7 + 5*7^8 + 2*7^9 + 3*7^10 + 6*7^11 + 2*7^12 + 3
```

chi()

If this character is $x \mapsto x^k\chi(x)$ for an integer k and a Dirichlet character χ, return χ.

EXAMPLE:

```
sage: kappa = pAdicWeightSpace(29)(13, DirichletGroup(29, Qp(29)).0^14)
sage: kappa.chi()
[28 + 28*29 + 28*29^2 + ... + O(29^20)]
```

k()

If this character is $x \mapsto x^k\chi(x)$ for an integer k and a Dirichlet character χ, return k.

EXAMPLE:

```
sage: kappa = pAdicWeightSpace(29)(13, DirichletGroup(29, Qp(29)).0^14)
sage: kappa.k()
13
```

teichmuller_type()

Return the Teichmuller type of this weight-character κ, which is the unique $t \in \mathbf{Z}/(p-1)\mathbf{Z}$ such that $\kappa(\mu) = \mu^t$ for mu a $(p-1)$-st root of 1.

For $p = 2$ this doesn't make sense, but we still want the Teichmuller type to correspond to the index of the component of weight space in which κ lies, so we return 1 if κ is odd and 0 otherwise.

EXAMPLE:

```
sage: pAdicWeightSpace(11)(2, DirichletGroup(11,QQ).0).teichmuller_type()
7
sage: pAdicWeightSpace(29)(13, DirichletGroup(29, Qp(29)).0).teichmuller_type()
14
sage: pAdicWeightSpace(2)(3, DirichletGroup(4,QQ).0).teichmuller_type()
0
```

class **ArbitraryWeight**(*parent, w, t*)

teichmuller_type()

Return the Teichmuller type of this weight-character κ, which is the unique $t \in \mathbf{Z}/(p-1)\mathbf{Z}$ such that $\kappa(\mu) = \mu^t$ for mu a $(p-1)$-st root of 1.

For $p = 2$ this doesn't make sense, but we still want the Teichmuller type to correspond to the index of the component of weight space in which κ lies, so we return 1 if κ is odd and 0 otherwise.

EXAMPLES:

```
sage: pAdicWeightSpace(17)(1 + 3*17 + 2*17^2 + O(17^3), 8, False).teichmuller_type()
8
sage: pAdicWeightSpace(2)(1 + 2 + O(2^2), 1, False).teichmuller_type()
1
```

class **WeightCharacter**(*parent*)

Abstract base class representing an element of the p-adic weight space $Hom(\mathbf{Z}_p^\times, \mathbf{C}_p^\times)$.

Lvalue()

Return the value of the p-adic L-function of $\mathbf{Q}$, which can be regarded as a rigid-analytic function on weight space, evaluated at this character.

EXAMPLES:

```
sage: W = pAdicWeightSpace(11)
sage: sage.modular.overconvergent.weightspace.WeightCharacter(W).Lvalue()
...
NotImplementedError
```

base_extend(*R*)

Extend scalars to the base ring R (which must have a canonical map from the current base ring)

EXAMPLE:

```
sage: w = pAdicWeightSpace(17, QQ)(3)
sage: w.base_extend(Qp(17))
3
```

is_even()

Return True if this weight-character sends -1 to +1.

EXAMPLE:

```
sage: pAdicWeightSpace(17)(0).is_even()
True
sage: pAdicWeightSpace(17)(11).is_even()
False
sage: pAdicWeightSpace(17)(1 + 17 + O(17^20), 3, False).is_even()
False
sage: pAdicWeightSpace(17)(1 + 17 + O(17^20), 4, False).is_even()
True
```

is_trivial()

Return True if and only if this is the trivial character.

EXAMPLES:

```
sage: pAdicWeightSpace(11)(2).is_trivial()
False
sage: pAdicWeightSpace(11)(2, DirichletGroup(11, QQ).0).is_trivial()
False
sage: pAdicWeightSpace(11)(0).is_trivial()
True
```

one_over_Lvalue()

Return the reciprocal of the p-adic L-function evaluated at this weight-character. If the weight-character is odd, then the L-function is zero, so an error will be raised.

EXAMPLES:

```
sage: pAdicWeightSpace(11)(4).one_over_Lvalue()
-12/133
sage: pAdicWeightSpace(11)(3, DirichletGroup(11, QQ).0).one_over_Lvalue()
-1/6
sage: pAdicWeightSpace(11)(3).one_over_Lvalue()
...
ZeroDivisionError: Rational division by zero
sage: pAdicWeightSpace(11)(0).one_over_Lvalue()
0
sage: type(_)
<type 'sage.rings.integer.Integer'>
```

pAdicEisensteinSeries(*ring, prec=20*)

Calculate the q-expansion of the p-adic Eisenstein series of given weight-character, normalised so the constant term is 1.

EXAMPLE:

```
sage: kappa = pAdicWeightSpace(3)(3, DirichletGroup(3,QQ).0)
sage: kappa.pAdicEisensteinSeries(QQ[['q']], 20)
1 - 9*q + 27*q^2 - 9*q^3 - 117*q^4 + 216*q^5 + 27*q^6 - 450*q^7 + 459*q^8 - 9*q^9 - 648*q^10
```

values_on_gens()

If κ is this character, calculate the values $(\kappa(r), t)$ where r is $1 + p$ (or 5 if $p = 2$) and t is the unique element of $\mathbf{Z}/(p-1)\mathbf{Z}$ such that $\kappa(\mu) = \mu^t$ for μ a (p-1)st root of unity. (If $p = 2$, we take t to be 0 or 1 according to whether κ is odd or even.) These two values uniquely determine the character κ.

EXAMPLES:

```
sage: W=pAdicWeightSpace(11); W(2).values_on_gens()
(1 + 2*11 + 11^2 + O(11^20), 2)
sage: W(2, DirichletGroup(11, QQ).0).values_on_gens()
(1 + 2*11 + 11^2 + O(11^20), 7)
sage: W(1 + 2*11 + O(11^5), 4, algebraic = False).values_on_gens()
(1 + 2*11 + O(11^5), 4)
```

class **WeightSpace_class**(*p, base_ring*)

The space of p-adic weight-characters $\mathcal{W} = \mathrm{Hom}(\mathbf{Z}_p^\times, \mathbf{C}_p^\times)$. This isomorphic to a disjoint union of $(p-1)$ open discs of radius 1 (or 2 such discs if $p = 2$), with the parameter on the open disc corresponding to the image of $1 + p$ (or 5 if $p = 2$)

TESTS:

```
sage: W = pAdicWeightSpace(3)
sage: W is loads(dumps(W))
True
```

base_extend(*R*)

Extend scalars to the ring R. There must be a canonical coercion map from the present base ring to R.

EXAMPLE:

```
sage: W = pAdicWeightSpace(3, QQ)
sage: W.base_extend(Qp(3))
Space of 3-adic weight-characters defined over '3-adic Field with capped relative precision
sage: W.base_extend(IntegerModRing(12))
...
TypeError: No coercion map from 'Rational Field' to 'Ring of integers modulo 12' is defined
```

prime()

Return the prime p such that this is a p-adic weight space.

EXAMPLE:

```
sage: pAdicWeightSpace(17).prime()
17
```

WeightSpace_constructor(*p, base_ring=None*)

Construct the p-adic weight space for the given prime p. A p-adic weight is a continuous character $\mathbf{Z}_p^\times \to \mathbf{C}_p^\times$. These are the $\mathbf{C}_p$-points of a rigid space over $\mathbf{Q}_p$, which is isomorphic to a disjoint union of copies (indexed by $(\mathbf{Z}/p\mathbf{Z})^\times$) of the open unit p-adic disc.

Note that the "base ring" of a p-adic weight is the smallest ring containing the image of $\mathbf{Z}$; in particular, although the default base ring is $\mathbf{Q}_p$, base ring $\mathbf{Q}$ will also work.

EXAMPLES:

```
sage: pAdicWeightSpace(3) # indirect doctest
Space of 3-adic weight-characters defined over '3-adic Field with capped relative precision 20'
sage: pAdicWeightSpace(3, QQ)
Space of 3-adic weight-characters defined over 'Rational Field'
sage: pAdicWeightSpace(10)
...
ValueError: p must be prime
```

46.7 Overconvergent p-adic modular forms for small primes

This module implements computations of the U_p-operator on overconvergent modular forms of level $\Gamma_0(p)$, where p is one of the primes $\{2, 3, 5, 7, 13\}$, using the algorithms described in:

David Loeffler, "Spectral expansions of overconvergent modular functions", Int. Math. Res. Not 2007(050).

AUTHORS:

- David Loeffler (August 2008): initial version

class OverconvergentModularFormElement(*parent, gexp=None, qexp=None*)

A class representing an element of a space of overconvergent modular forms.

EXAMPLE:

```
sage: K.<w> = Qp(5).extension(x^7 - 5); s = OverconvergentModularForms(5, 6, 1/21, base_ring=K).
sage: s == loads(dumps(s))
True
```

additive_order()

Return the additive order of this element (required attribute for all elements deriving from sage.modules.ModuleElement)

EXAMPLES:

```
sage: M = OverconvergentModularForms(13, 10, 1/2, base_ring = Qp(13).extension(x^2 - 13,name
sage: M.gen(0).additive_order()
+Infinity
sage: M(0).additive_order()
1
```

base_extend(*R*)

Return a copy of self but with coefficients in the given ring.

EXAMPLES:

```
sage: M = OverconvergentModularForms(7, 10, 1/2, prec=5)
sage: f = M.1
sage: f.base_extend(Qp(7, 4))
7-adic overconvergent modular form of weight-character 10 with q-expansion (7 + O(7^5))*q +
```

eigenvalue()

Return the U_p-eigenvalue of this eigenform. Raises an error unless this element was explicitly flagged as an eigenform, using the _notify_eigen function.

EXAMPLE:

```
sage: M = OverconvergentModularForms(3, 0, 1/2)
sage: f = M.eigenfunctions(3)[1]
sage: f.eigenvalue()
3^2 + 3^4 + 2*3^6 + 3^7 + 3^8 + 2*3^9 + 2*3^10 + 3^12 + 3^16 + 2*3^17 + 3^18 + 3^20 + 2*3^21
sage: M.gen(4).eigenvalue()
...
TypeError: eigenvalue only defined for eigenfunctions
```

gexp()

Return the formal power series in g corresponding to this overconvergent modular form (so the result is F where this modular form is $E_k^* \times F(g)$, where g is the appropriately normalised parameter of $X_0(p)$).

EXAMPLE:

```
sage: M = OverconvergentModularForms(3, 0, 1/2)
sage: f = M.eigenfunctions(3)[1]
sage: f.gexp()
(3^-3 + O(3^91))*g + (3^-1 + 1 + 2*3 + 3^2 + 2*3^3 + 3^5 + 3^7 + 3^10 + 3^11 + 3^14 + 3^15 +
```

governing_term(*r*)

The degree of the series term with largest norm on the r-overconvergent region.

EXAMPLES:

```
sage: o=OverconvergentModularForms(3, 0, 1/2)
sage: f=o.eigenfunctions(10)[1]
sage: f.governing_term(1/2)
1
```

is_eigenform()

Return True if this is an eigenform. At present this returns False unless this element was explicitly flagged as an eigenform, using the _notify_eigen function.

EXAMPLE:

```
sage: M = OverconvergentModularForms(3, 0, 1/2)
sage: f = M.eigenfunctions(3)[1]
sage: f.is_eigenform()
True
sage: M.gen(4).is_eigenform()
False
```

is_integral()

Test whether or not this element has q-expansion coefficients that are p-adically integral. This should always be the case with eigenfunctions, but sometimes if n is very large this breaks down for unknown reasons!

EXAMPLE:

```
sage: M = OverconvergentModularForms(2, 0, 1/3)
sage: q = QQ[['q']].gen()
sage: M(q - 17*q^2 + O(q^3)).is_integral()
```

```
True
sage: M(q - q^2/2 + 6*q^7  + O(q^9)).is_integral()
False
```

prec()

Return the series expansion precision of this overconvergent modular form. (This is not the same as the p-adic precision of the coefficients.)

EXAMPLE:

```
sage: OverconvergentModularForms(5, 6, 1/3,prec=15).gen(1).prec()
15
```

prime()

If this is a p-adic modular form, return p.

EXAMPLE:

```
sage: OverconvergentModularForms(2, 0, 1/2).an_element().prime()
2
```

qexp()

Return the q-expansion of self, to as high precision as it is known.

EXAMPLE:

```
sage: OverconvergentModularForms(3, 4, 1/2).gen(0).qexp()
1 - 120/13*q - 1080/13*q^2 - 120/13*q^3 - 8760/13*q^4 - 15120/13*q^5 - 1080/13*q^6 - 41280/1
```

r_ord(*r*)

The p-adic valuation of self on the r-overconvergent region.

EXAMPLES:

```
sage: o=OverconvergentModularForms(3, 0, 1/2)
sage: t = o([1, 1, 1/3])
sage: t.r_ord(1/2)
1
sage: t.r_ord(2/3)
3
```

slope()

Return the slope of this eigenform, i.e. the valuation of its U_p-eigenvalue. Raises an error unless this element was explicitly flagged as an eigenform, using the _notify_eigen function.

EXAMPLE:

```
sage: M = OverconvergentModularForms(3, 0, 1/2)
sage: f = M.eigenfunctions(3)[1]
sage: f.slope()
2
sage: M.gen(4).slope()
...
TypeError: slope only defined for eigenfunctions
```

valuation_plot()

Draw a graph depicting the growth of the norm of this overconvergent modular form as it approaches the boundary of the overconvergent region.

EXAMPLE:

```
sage: o=OverconvergentModularForms(3, 0, 1/2)
sage: f=o.eigenfunctions(4)[1]
sage: f.valuation_plot()
```

weight()

Return the weight of this overconvergent modular form.

EXAMPLES:

```
sage: M = OverconvergentModularForms(13, 10, 1/2, base_ring = Qp(13).extension(x^2 - 13,name
sage: M.gen(0).weight()
10
```

OverconvergentModularForms(*prime, weight, radius, base_ring=Rational Field, prec=20, char=None*)

Create a space of overconvergent p-adic modular forms of level $\Gamma_0(p)$, over the given base ring. The base ring need not be a p-adic ring (the spaces we compute with typically have bases over $\mathbf{Q}$).

INPUT:

- `prime` - a prime number p, which must be one of the primes $\{2, 3, 5, 7, 13\}$, or the congruence subgroup Gamma0(p) where p is one of these primes.
- `weight` - an integer (which at present must be 0 or ≥ 2), the weight.
- `radius` - a rational number in the interval $\left(0, \frac{p}{p+1}\right)$, the radius of overconvergence.
- `base_ring` (default: $\mathbf{Q}$), a ring over which to compute. This need not be a p-adic ring.
- `prec` - an integer (default: 20), the number of q-expansion terms to compute.
- `char` - a Dirichlet character modulo p or `None` (the default). Here `None` is interpreted as the trivial character modulo p.

The character χ and weight k must satisfy $(-1)^k = \chi(-1)$, and the base ring must contain an element v such that $\mathrm{ord}_p(v) = \frac{12r}{p-1}$ where r is the radius of overconvergence (and ord_p is normalised so $\mathrm{ord}_p(p) = 1$).

EXAMPLES:

```
sage: OverconvergentModularForms(3, 0, 1/2)
Space of 3-adic 1/2-overconvergent modular forms of weight-character 0 over Rational Field
sage: OverconvergentModularForms(3, 16, 1/2)
Space of 3-adic 1/2-overconvergent modular forms of weight-character 16 over Rational Field
sage: OverconvergentModularForms(3, 3, 1/2, char = DirichletGroup(3,QQ).0)
Space of 3-adic 1/2-overconvergent modular forms of weight-character (3, 3, [-1]) over Rational
```

class OverconvergentModularFormsSpace(*prime, weight, radius, base_ring, prec, char*)

A space of overconvergent modular forms of level $\Gamma_0(p)$, where p is a prime such that $X_0(p)$ has genus 0.

Elements are represented as power series, with a formal power series F corresponding to the modular form $E_k^* \times F(g)$ where E_k^* is the p-deprived Eisenstein series of weight-character k, and g is a uniformiser of $X_0(p)$ normalised so that the r-overconvergent region $X_0(p)_{\geq r}$ corresponds to $|g| \leq 1$.

TESTS:

```
sage: K.<w> = Qp(13).extension(x^2-13); M = OverconvergentModularForms(13, 20, radius=1/2, base_
sage: M is loads(dumps(M))
True
```

base_extend(*ring*)

Return the base extension of self to the given base ring. There must be a canonical map to this ring from the current base ring, otherwise a TypeError will be raised.

EXAMPLES:

```
sage: M = OverconvergentModularForms(2, 0, 1/2, base_ring = Qp(2))
sage: M.base_extend(Qp(2).extension(x^2 - 2, names="w"))
Space of 2-adic 1/2-overconvergent modular forms of weight-character 0 over Eisenstein Exter
sage: M.base_extend(QQ)
...
TypeError: Base extension of self (over '2-adic Field with capped relative precision 20') to
```

change_ring(*ring*)

Return the space corresponding to self but over the given base ring.

EXAMPLES:

```
sage: M = OverconvergentModularForms(2, 0, 1/2)
sage: M.change_ring(Qp(2))
Space of 2-adic 1/2-overconvergent modular forms of weight-character 0 over 2-adic Field wit
```

character()

Return the character of self. For overconvergent forms, the weight and the character are unified into the concept of a weight-character, so this returns exactly the same thing as self.weight().

EXAMPLE:

```
sage: OverconvergentModularForms(3, 0, 1/2).character()
0
sage: type(OverconvergentModularForms(3, 0, 1/2).character())
<class '...weightspace.AlgebraicWeight'>
sage: OverconvergentModularForms(3, 3, 1/2, char=DirichletGroup(3,QQ).0).character()
(3, 3, [-1])
```

coordinate_vector(*x*)

Write x as a vector with respect to the basis given by self.basis(). Here x must be an element of this space or something that can be converted into one. If x has precision less than the default precision of self, then the returned vector will be shorter.

EXAMPLES:

```
sage: M = OverconvergentModularForms(Gamma0(3), 0, 1/3, prec=4)
sage: M.coordinate_vector(M.gen(2))
(0, 0, 1, 0)
sage: q = QQ[['q']].gen(); M.coordinate_vector(q - q^2 + O(q^4))
(0, 1/9, -13/81, 74/243)
sage: M.coordinate_vector(q - q^2 + O(q^3))
(0, 1/9, -13/81)
```

cps_u(*n, use_recurrence=False*)

Compute the characteristic power series of U_p acting on self, using an n x n matrix.

EXAMPLES:

```
sage: OverconvergentModularForms(3, 16, 1/2, base_ring=Qp(3)).cps_u(4)
1 + O(3^20) + (2 + 2*3 + 2*3^2 + 2*3^4 + 3^5 + 3^6 + 3^7 + 3^11 + 3^12 + 2*3^14 + 3^16 + 3^1
sage: OverconvergentModularForms(3, 16, 1/2, base_ring=Qp(3), prec=30).cps_u(10)
1 + O(3^20) + (2 + 2*3 + 2*3^2 + 2*3^4 + 3^5 + 3^6 + 3^7 + 2*3^15 + O(3^16))*T + (2*3^3 + 3^
```

eigenfunctions(*n, F=None, exact_arith=True*)

Calculate approximations to eigenfunctions of self. These are the eigenfunctions of self.hecke_matrix(p, n), which are approximations to the true eigenfunctions. Returns a list of OverconvergentModularFormElement objects, in increasing order of slope.

INPUT:

- `n` - integer. The size of the matrix to use.
- `F` - None, or a field over which to calculate eigenvalues. If the field is None, the current base ring is used. If the base ring is not a p-adic ring, an error will be raised.
- `exact_arith` - True or False (default True). If True, use exact rational arithmetic to calculate the matrix of the U operator and its characteristic power series, even when the base ring is an inexact p-adic ring. This is typically slower, but more numerically stable.

NOTE: Try using `set_verbose(1, 'sage/modular/overconvergent')` to get more feedback on what is going on in this algorithm. For even more feedback, use 2 instead of 1.

EXAMPLES:

```
sage: X = OverconvergentModularForms(2, 2, 1/6).eigenfunctions(8, Qp(2, 100))
sage: X[1]
2-adic overconvergent modular form of weight-character 2 with q-expansion (1 + O(2^36))*q +
sage: [x.slope() for x in X]
[0, 4, 8, 14, 16, 18, 26, 30]
```

gen(*i*)

Return the ith module generator of self.

EXAMPLE:

```
sage: M = OverconvergentModularForms(3, 2, 1/2, prec=4)
sage: M.gen(0)
3-adic overconvergent modular form of weight-character 2 with q-expansion 1 + 12*q + 36*q^2
sage: M.gen(1)
3-adic overconvergent modular form of weight-character 2 with q-expansion 27*q + 648*q^2 + 7
sage: M.gen(30)
3-adic overconvergent modular form of weight-character 2 with q-expansion O(q^4)
```

gens()

Return a generator object that iterates over the (infinite) set of basis vectors of self.

EXAMPLES:

```
sage: o = OverconvergentModularForms(3, 12, 1/2)
sage: t = o.gens()
sage: t.next()
3-adic overconvergent modular form of weight-character 12 with q-expansion 1 - 32760/6120394
sage: t.next()
3-adic overconvergent modular form of weight-character 12 with q-expansion 27*q + 1982919301
```

gens_dict()

Return a dictionary mapping the names of generators of this space to their values. (Required by parent class definition.) As this does not make any sense here, this raises a TypeError.

EXAMPLES:

```
sage: M = OverconvergentModularForms(2, 4, 1/6)
sage: M.gens_dict()
...
TypeError: gens_dict does not make sense as number of generators is infinite
```

hecke_matrix(*m, n, use_recurrence=False, exact_arith=False*)

Calculate the matrix of the T_m operator in the basis of this space, truncated to an $n \times n$ matrix. Conventions are that operators act on the left on column vectors (this is the opposite of the conventions of the sage.modules.matrix_morphism class!) Uses naive q-expansion arguments if use_recurrence=False and uses the Kolberg style recurrences if use_recurrence=True.

The argument "exact_arith" causes the computation to be done with rational arithmetic, even if the base ring is an inexact p-adic ring. This is useful as there can be precision loss issues (particularly with use_recurrence=False).

EXAMPLES:

```
sage: OverconvergentModularForms(2, 0, 1/2).hecke_matrix(2, 4)
[    1     0     0     0]
[    0    24    64     0]
[    0    32  1152  4608]
[    0     0  3072 61440]
sage: OverconvergentModularForms(2, 12, 1/2, base_ring=pAdicField(2)).hecke_matrix(2, 3) * (
[        1 + O(2^2)                   0                   0]
[                 0       2^3 + O(2^5)       2^6 + O(2^8)]
[                 0       2^4 + O(2^6) 2^7 + 2^8 + O(2^9)]
```

```
sage: OverconvergentModularForms(2, 12, 1/2, base_ring=pAdicField(2)).hecke_matrix(2, 3, exa
[                              1                               0
[                              0                33881928/1414477                              6
[                              0 -192898739923312/2000745183529             1626332544/141447
```

hecke_operator(*f*, *m*)

Given an element f and an integer m, calculates the Hecke operator T_m acting on f.

The input may be either a "bare" power series, or an OverconvergentModularFormElement object; the return value will be of the same type.

EXAMPLE:

```
sage: M = OverconvergentModularForms(3, 0, 1/2)
sage: f = M.1
sage: M.hecke_operator(f, 3)
3-adic overconvergent modular form of weight-character 0 with q-expansion 2430*q + 265356*q^
sage: M.hecke_operator(f.qexp(), 3)
2430*q + 265356*q^2 + 10670373*q^3 + 249948828*q^4 + 4113612864*q^5 + 52494114852*q^6 + O(q^
```

is_exact()

True if elements of this space are represented exactly, i.e., there is no precision loss when doing arithmetic. As this is never true for overconvergent modular forms spaces, this returns False.

EXAMPLES:

```
sage: OverconvergentModularForms(13, 12, 0).is_exact()
False
```

ngens()

The number of generators of self (as a module over its base ring), i.e. infinity.

EXAMPLES:

```
sage: M = OverconvergentModularForms(2, 4, 1/6)
sage: M.ngens()
+Infinity
```

normalising_factor()

The normalising factor c such that $g = cf$ is a parameter for the r-overconvergent disc in $X_0(p)$, where f is the standard uniformiser.

EXAMPLE:

```
sage: L.<w> = Qp(7).extension(x^2 - 7)
sage: OverconvergentModularForms(7, 0, 1/4, base_ring=L).normalising_factor()
w + O(w^41)
```

prec()

Return the series precision of self. Note that this is different from the p-adic precision of the base ring.

EXAMPLE:

```
sage: OverconvergentModularForms(3, 0, 1/2).prec()
20
sage: OverconvergentModularForms(3, 0, 1/2,prec=40).prec()
40
```

prime()

Return the residue characteristic of self, i.e. the prime p such that this is a p-adic space.

EXAMPLES:

```
sage: OverconvergentModularForms(5, 12, 1/3).prime()
5
```

radius()

The radius of overconvergence of this space.

EXAMPLE:

```
sage: OverconvergentModularForms(3, 0, 1/3).radius()
1/3
```

recurrence_matrix(*use_smithline=True*)

Return the recurrence matrix satisfied by the coefficients of U, that is a matrix $R = (r_{rs})_{r,s=1...p}$ such that $u_{ij} = \sum_{r,s=1}^{p} r_{rs} u_{i-r,j-s}$. Uses an elegant construction which I believe is due to Smithline. See my paper in IMRN (full citation in reference manual).

EXAMPLES:

```
sage: OverconvergentModularForms(2, 0, 0).recurrence_matrix()
[  48    1]
[4096    0]
sage: OverconvergentModularForms(2, 0, 1/2).recurrence_matrix()
[48 64]
[64  0]
sage: OverconvergentModularForms(3, 0, 0).recurrence_matrix()
[   270     36      1]
[ 26244    729      0]
[531441      0      0]
sage: OverconvergentModularForms(5, 0, 0).recurrence_matrix()
[     1575      1300       315        30         1]
[   162500     39375      3750       125         0]
[  4921875    468750     15625         0         0]
[ 58593750   1953125         0         0         0]
[244140625         0         0         0         0]
sage: OverconvergentModularForms(7, 0, 0).recurrence_matrix()
[       4018        8624        5915        1904         322          28           1]
[     422576      289835       93296       15778        1372          49           0]
[   14201915     4571504      773122       67228        2401           0           0]
[  224003696    37882978     3294172      117649           0           0           0]
[ 1856265922   161414428     5764801           0           0           0           0]
[ 7909306972   282475249           0           0           0           0           0]
[13841287201           0           0           0           0           0           0]
sage: OverconvergentModularForms(13, 0, 0).recurrence_matrix()
[         15145         124852         354536 ...
```

slopes(*n, use_recurrence=False*)

Compute the slopes of the U_p operator acting on self, using an n x n matrix.

EXAMPLES:: sage: OverconvergentModularForms(5,2,1/3,base_ring=Qp(5),prec=100).slopes(5) [0, 2, 5, 6, 9] sage: OverconvergentModularForms(2,1,1/3,char=DirichletGroup(4,QQ).0).slopes(5) [0, 2, 4, 6, 8]

weight()

Return the character of self. For overconvergent forms, the weight and the character are unified into the concept of a weight-character, so this returns exactly the same thing as self.character().

EXAMPLE:

```
sage: OverconvergentModularForms(3, 0, 1/2).weight()
0
sage: type(OverconvergentModularForms(3, 0, 1/2).weight())
<class '...weightspace.AlgebraicWeight'>
sage: OverconvergentModularForms(3, 3, 1/2, char=DirichletGroup(3,QQ).0).weight()
(3, 3, [-1])
```

46.8 Module of Supersingular Points

The module of divisors on the modular curve $X_0(N)$ over F_p supported at supersingular points.

AUTHORS:

- William Stein
- David Kohel
- Iftikhar Burhanuddin

EXAMPLES:

```
sage: x = SupersingularModule(389)
sage: m = x.T(2).matrix()
sage: a = m.change_ring(GF(97))
sage: D = a.decomposition()
sage: D[:3]
[
(Vector space of degree 33 and dimension 1 over Finite Field of size 97
Basis matrix:
[ 0  0  0  1 96 96  1 96 96  0  2 96 96  0  1  0  1  2 95  0  1  1  0  1  0 95  0 96 95  1 96  0  2],
(Vector space of degree 33 and dimension 1 over Finite Field of size 97
Basis matrix:
[ 0  1 96 75 16 81 22 17 17  0  0 80 80  1 16 40 74  0  0 96 81 23 57 74  0  0  0 24  0 23 73  0  0],
(Vector space of degree 33 and dimension 1 over Finite Field of size 97
Basis matrix:
[ 0  1 96 90 90  7  7  6 91  0  0 91  6 13  7  0 91  0  0 84 90  6  0  6  0  0  0 90  0 91  7  0  0],
]
sage: len(D)
9
```

We compute a Hecke operator on a space of huge dimension!:

```
sage: X = SupersingularModule(next_prime(100000))
sage: t = X.T(2).matrix()            # long time (but still less than a minute!)
sage: t.nrows()                      # long time
8334
```

TESTS:

```
sage: X = SupersingularModule(389)
sage: T = X.T(2).matrix().change_ring(QQ)
sage: d = T.decomposition()
sage: len(d)
6
sage: [a[0].dimension() for a in d]
[1, 1, 2, 3, 6, 20]
sage: loads(dumps(X)) == X
True
sage: loads(dumps(d)) == d
True
```

Phi2_quad(*J3, ssJ1, ssJ2*)

This function returns a certain quadratic polynomial over a finite field in indeterminate J3.

The roots of the polynomial along with ssJ1 are the neighboring/2-isogenous supersingular j-invariants of ssJ2.

INPUT:

- J3 – indeterminate of a univariate polynomial ring defined over a finite field with p^2 elements where p is a prime number
- ssJ2, ssJ2 – supersingular j-invariants over the finite field

OUTPUT:

- polynomial – defined over the finite field

EXAMPLES: The following code snippet produces a factor of the modular polynomial $\Phi_2(x, j_{in})$, where j_{in} is a supersingular j-invariant defined over the finite field with 37^2 elements:

```
sage: F = GF(37^2, 'a')
sage: X = PolynomialRing(F, 'x').gen()
sage: j_in = supersingular_j(F)
sage: poly = sage.modular.ssmod.ssmod.Phi_polys(2,X,j_in)
sage: poly.roots()
[(8, 1), (27*a + 23, 1), (10*a + 20, 1)]
sage: sage.modular.ssmod.ssmod.Phi2_quad(X, F(8), j_in)
x^2 + 31*x + 31
```

Note: Given a root (j1,j2) to the polynomial $Phi_2(J1, J2)$, the pairs (j2,j3) not equal to (j2,j1) which solve $Phi_2(j2, j3)$ are roots of the quadratic equation:

J3^2 + (-j2^2 + 1488*j2 + (j1 - 162000))*J3 + (-j1 + 1488)*j2^2 + (1488*j1 + 40773375)*j2 + j1^2 - 162000*j1 + 8748000000

This will be of use to extend the 2-isogeny graph, once the initial three roots are determined for $Phi_2(J1, J2)$.

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

Phi_polys(*L, x, j*)

This function returns a certain polynomial of degree $L + 1$ in the indeterminate x over a finite field.

The roots of the **modular** polynomial $\Phi(L, x, j)$ are the L-isogenous supersingular j-invariants of j.

INPUT:

- L – integer, either 2,3,5,7 or 11
- x – indeterminate of a univariate polynomial ring defined over a finite field with p^2 elements, where p is a prime number
- j – supersingular j-invariant over the finite field

OUTPUT:

- polynomial – defined over the finite field

EXAMPLES: The following code snippet produces the modular polynomial $\Phi_L(x, j_{in})$, where j_{in} is a supersingular j-invariant defined over the finite field with 7^2 elements:

```
sage: F = GF(7^2, 'a')
sage: X = PolynomialRing(F, 'x').gen()
sage: j_in = supersingular_j(F)
sage: sage.modular.ssmod.ssmod.Phi_polys(2,X,j_in)
x^3 + 3*x^2 + 3*x + 1
sage: sage.modular.ssmod.ssmod.Phi_polys(3,X,j_in)
x^4 + 4*x^3 + 6*x^2 + 4*x + 1
```

```
sage: sage.modular.ssmod.ssmod.Phi_polys(5,X,j_in)
x^6 + 6*x^5 + x^4 + 6*x^3 + x^2 + 6*x + 1
sage: sage.modular.ssmod.ssmod.Phi_polys(7,X,j_in)
x^8 + x^7 + x + 1
sage: sage.modular.ssmod.ssmod.Phi_polys(11,X,j_in)
x^12 + 5*x^11 + 3*x^10 + 3*x^9 + 5*x^8 + x^7 + x^5 + 5*x^4 + 3*x^3 + 3*x^2 + 5*x + 1
```

AUTHORS:

- William Stein – wstein@gmail.com
- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

class SupersingularModule(*prime=2, level=1, base_ring=Integer Ring*)

The module of supersingular points in a given characteristic, with given level structure.

The characteristic must not divide the level.

NOTE: Currently, only level 1 is implemented.

EXAMPLES:

```
sage: S = SupersingularModule(17)
sage: S
Module of supersingular points on X_0(1)/F_17 over Integer Ring
sage: S = SupersingularModule(16)
...
ValueError: the argument prime must be a prime number
sage: S = SupersingularModule(prime=17, level=34)
...
ValueError: the argument level must be coprime to the argument prime
sage: S = SupersingularModule(prime=17, level=5)
...
NotImplementedError: supersingular modules of level > 1 not yet implemented
```

dimension()

Return the dimension of the space of modular forms of weight 2 and level equal to the level associated to self.

INPUT:

- `self` – SupersingularModule object

OUTPUT: integer – dimension, nonnegative

EXAMPLES:

```
sage: S = SupersingularModule(7)
sage: S.dimension()
1

sage: S = SupersingularModule(15073)
sage: S.dimension()
1256

sage: S = SupersingularModule(83401)
sage: S.dimension()
6950
```

NOTES: The case of level > 1 has not yet been implemented.

AUTHORS:

•David Kohel – kohel@maths.usyd.edu.au

•Iftikhar Burhanuddin – burhanud@usc.edu

hecke_matrix(*L*)

This function returns the L^{th} Hecke matrix.

INPUT:

•`self` – SupersingularModule object

•`L` – integer, positive

OUTPUT: matrix – sparse integer matrix

EXAMPLES: This example computes the action of the Hecke operator T_2 on the module of supersingular points on $X_0(1)/F_{37}$:

```
sage: S = SupersingularModule(37)
sage: M = S.hecke_matrix(2)
sage: M
[1 1 1]
[1 0 2]
[1 2 0]
```

This example computes the action of the Hecke operator T_3 on the module of supersingular points on $X_0(1)/F_{67}$:

```
sage: S = SupersingularModule(67)
sage: M = S.hecke_matrix(3)
sage: M
[0 0 0 0 2 2]
[0 0 1 1 1 1]
[0 1 0 2 0 1]
[0 1 2 0 1 0]
[1 1 0 1 0 1]
[1 1 1 0 1 0]
```

Note: The first list — list_j — returned by the supersingular_points function are the rows *and* column indexes of the above hecke matrices and its ordering should be kept in mind when interpreting these matrices.

AUTHORS:

•David Kohel – kohel@maths.usyd.edu.au

•Iftikhar Burhanuddin – burhanud@usc.edu

level()

This function returns the level associated to self.

INPUT:

•`self` – SupersingularModule object

OUTPUT: integer – the level, positive

EXAMPLES:

```
sage: S = SupersingularModule(15073)
sage: S.level()
1
```

AUTHORS:

•David Kohel – kohel@maths.usyd.edu.au

•Iftikhar Burhanuddin – burhanud@usc.edu

prime()

This function returns the characteristic of the finite field associated to self.

INPUT:

•self – SupersingularModule object

OUTPUT:

•integer – charateristic, positive

EXAMPLES:

```
sage: S = SupersingularModule(19)
sage: S.prime()
19
```

AUTHORS:

•David Kohel – kohel@maths.usyd.edu.au

•Iftikhar Burhanuddin – burhanud@usc.edu

rank()

Return the dimension of the space of modular forms of weight 2 and level equal to the level associated to self.

INPUT:

•self – SupersingularModule object

OUTPUT: integer – dimension, nonnegative

EXAMPLES:

```
sage: S = SupersingularModule(7)
sage: S.dimension()
1

sage: S = SupersingularModule(15073)
sage: S.dimension()
1256

sage: S = SupersingularModule(83401)
sage: S.dimension()
6950
```

NOTES: The case of level > 1 has not yet been implemented.

AUTHORS:

•David Kohel – kohel@maths.usyd.edu.au

•Iftikhar Burhanuddin – burhanud@usc.edu

supersingular_points()

This function computes the supersingular j-invariants over the finite field associated to self.

INPUT:

•self – SupersingularModule object

OUTPUT: list_j, dict_j – list_j is the list of supersingular j-invariants, dict_j is a dictionary with these j-invariants as keys and their indexes as values. The latter is used to speed up j-invariant look-up. The indexes are based on the order of their *discovery*.

EXAMPLES:

The following examples calculate supersingular j-invariants over finite fields with characteristic 7, 11 and 37:

```
sage: S = SupersingularModule(7)
sage: S.supersingular_points()
([6], {6: 0})

sage: S = SupersingularModule(11)
```

```
sage: S.supersingular_points()
([1, 0], {0: 1, 1: 0})

sage: S = SupersingularModule(37)
sage: S.supersingular_points()
([8, 27*a + 23, 10*a + 20], {8: 0, 10*a + 20: 2, 27*a + 23: 1})
```

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

upper_bound_on_elliptic_factors(*p=None, ellmax=2*)

Return an upper bound (provably correct) on the number of elliptic curves of conductor equal to the level of this supersingular module.

INPUT:

- p - (default: 997) prime to work modulo

ALGORITHM: Currently we only use T_2. Function will be extended to use more Hecke operators later.

The prime p is replaced by the smallest prime that doesn't divide the level.

EXAMPLE:

```
sage: SupersingularModule(37).upper_bound_on_elliptic_factors()
2
```

(There are 4 elliptic curves of conductor 37, but only 2 isogeny classes.)

weight()

This function returns the weight associated to self.

INPUT:

- self – SupersingularModule object

OUTPUT: integer – weight, positive

EXAMPLES:

```
sage: S = SupersingularModule(19)
sage: S.weight()
2
```

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin – burhanud@usc.edu

dimension_supersingular_module(*prime, level=1*)

This function returns the dimension of the Supersingular module, which is equal to the dimension of the space of modular forms of weight 2 and conductor equal to prime times level.

INPUT:

- prime – integer, prime
- level – integer, positive

OUTPUT: dimension – integer, nonnegative

EXAMPLES: The code below computes the dimensions of Supersingular modules with level=1 and prime = 7, 15073 and 83401:

```
sage: dimension_supersingular_module(7)
1

sage: dimension_supersingular_module(15073)
1256

sage: dimension_supersingular_module(83401)
6950
```

NOTES: The case of level > 1 has not been implemented yet.

AUTHORS:

- David Kohel – kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin - burhanud@usc.edu

supersingular_D(*prime*)

This function returns a fundamental discriminant D of an imaginary quadratic field, where the given prime does not split (see Silverman's Advanced Topics in the Arithmetic of Elliptic Curves, page 184, exercie 2.30(d).)

INPUT:

- prime – integer, prime

OUTPUT: D – integer, negative

EXAMPLES:

These examples return *supersingular discriminants* for 7, 15073 and 83401:

```
sage: supersingular_D(7)
-4

sage: supersingular_D(15073)
-15

sage: supersingular_D(83401)
-7
```

AUTHORS:

- David Kohel - kohel@maths.usyd.edu.au
- Iftikhar Burhanuddin - burhanud@usc.edu

supersingular_j(*FF*)

This function returns a supersingular j-invariant over the finite field FF.

INPUT:

- `FF` – finite field with p^2 elements, where p is a prime number

OUTPUT: finite field element – a supersingular j-invariant defined over the finite field FF

EXAMPLES:

The following examples calculate supersingular j-invariants for a few finite fields:

```
sage: supersingular_j(GF(7^2, 'a'))
6
```

Observe that in this example the j-invariant is not defined over the prime field:

```
sage: supersingular_j(GF(15073^2,'a'))  # optional -- requires database
10630*a + 6033

sage: supersingular_j(GF(83401^2, 'a'))
67977
```

AUTHORS:

•David Kohel – kohel@maths.usyd.edu.au

•Iftikhar Burhanuddin – burhanud@usc.edu

46.9 Brandt Modules

AUTHORS:

- Jon Bober
- Alia Hamieh
- Victoria de Quehen
- William Stein
- Gonzalo Tornaria

46.9.1 Introduction

This tutorial outlines the construction of Brandt modules in SAGE. The importance of this construction is that it provides us with a method to compute modular forms on $\Gamma_0(N)$ as outlined in Pizer's paper [Pi]. In fact there exists a non-canonical Hecke algebra isomorphism between the Brandt modules and a certain subspace of $S_2(\Gamma_0(pM))$ which contains all the newforms.

The Brandt module is the free abelian group on right ideal classes of a quaternion order together with a natural Hecke action determined by Brandt matrices.

Quaternion Algebras

A quaternion algebra over $\mathbf{Q}$ is a central simple algebra of dimension 4 over $\mathbf{Q}$. Such an algebra A is said to be ramified at a place v of $\mathbf{Q}$ if and only if $A_v = A \otimes \mathbf{Q}_v$ is a division algebra. Otherwise A is said to be split at v.

`A = QuaternionAlgebra(p)` returns the quaternion algebra A over $\mathbf{Q}$ ramified precisely at the places p and ∞.

`A = QuaternionAlgebra(k,a,b)` returns a quaternion algebra with basis $\{1, i, j, j\}$ over $\mathbb{K}$ such that $i^2 = a$, $j^2 = b$ and $ij = k$.

An order R in a quaternion algebra is a 4-dimensional lattice on A which is also a subring containing the identity.

`R = A.maximal_order()` returns a maximal order R in the quaternion algebra A.

An Eichler order $\mathcal{O}$ in a quaternion algebra is the intersection of two maximal orders. The level of $\mathcal{O}$ is its index in any maximal order containing it.

`O = A.order_of_level_N` returns an Eichler order $\mathcal{O}$ in A of level N where p does not divide N.

A right $\mathcal{O}$-ideal I is a lattice on A such that $I_p = a_p\mathcal{O}$ (for some $a_p \in A_p^*$) for all $p < \infty$. Two right $\mathcal{O}$-ideals I and J are said to belong to the same class if $I = aJ$ for some $a \in A^*$. (Left $\mathcal{O}$-ideals are defined in a similar fashion.)

The right order of I is defined to be the set of elements in A which fix I under right multiplication.

right_order (R, basis) returns the right ideal of I in R given a basis for the right ideal I contained in the maximal order R.

ideal_classes(self) returns a tuple of all right ideal classes in self which, for the purpose of constructing the Brandt module B(p,M), is taken to be an Eichler order of level M.

The implementation of this method is especially interesting. It depends on the construction of a Hecke module defined as a free abelian group on right ideal classes of a quaternion algebra with the following action

where $(n, pM) = 1$ and the sum is over cyclic $\mathcal{O}$-module homomorphisms $\phi : I \to J `of degree` n$ up to isomorphism of J. Equivalently one can sum over the inclusions of the submodules $J \to n^{-1}I$. The rough idea is to start with the trivial ideal class containing the order $\mathcal{O}$ itself. Using the method cyclic_submodules(self, I, p) one computes $T_p([\mathcal{O}])$ for some prime integer \$p\$ not dividing the level of the order $\mathcal{O}$. Apply this method repeatedly and test for equivalence among resulting ideals. A theorem of Serre asserts that one gets a complete set of ideal class representatives after a finite number of repetitions.

One can prove that two ideals I and J are equivalent if and only if there exists an element $\alpha \in I\overline{J}$ such $N(\alpha) = N(I)N(J)$.

is_equivalent(I,J) returns true if I and J are equivalent. This method first compares the theta series of I and J. If they are the same, it computes the theta series of the lattice $I\overline{(J)}$. It returns true if the n^{th} coefficient of this series is nonzero where $n = N(J)N(I)$.

The theta series of a lattice L over the quaternion algebra A is defined as

L.theta_series(T,q) returns a power series representing $\theta_L(q)$ up to a precision of $\mathcal{O}(q^{T+1})$.

Hecke Structure

The Hecke structure defined on the Brandt module is given by the Brandt matrices which can be computed using the definition of the Hecke operators given earlier.

hecke_matrix_from_defn (self,n) returns the matrix of the nth Hecke operator $B_0(n)$ acting on self, computed directly from the definition.

However, one can efficiently compute Brandt matrices using theta series. In fact, let $\{I_1, ..., I_h\}$ be a set of right $\mathcal{O}$-ideal class representatives. The (i,j) entry in the Brandt matrix $B_0(n)$ is the product of the n^{th} coefficient in the theta series of the lattice $I_i\overline{I_j}$ and the first coefficient in the theta series of the lattice $I_i\overline{I_i}$.

compute_hecke_matrix_brandt(self,n) returns the nth Hecke matrix, computed using theta series.

Example

```
sage: B = BrandtModule(23)

sage: B.maximal_order()
Order of Quaternion Algebra (-1, -23) with base ring Rational Field with basis (1/2 + 1/2*j, 1/2*i +

sage: B.right_ideals()
(Fractional ideal (2 + 2*j, 2*i + 2*k, 4*j, 4*k), Fractional ideal (2 + 2*j, 2*i + 6*k, 8*j, 8*k), Fr

sage: B.hecke_matrix(2)
[1 2 0]
[1 1 1]
[0 3 0]

sage: B.brandt_series(3)
```

```
[1/4 + q + q^2 + O(q^3)      1/4 + q^2 + O(q^3)            1/4 + O(q^3)]
[  1/2 + 2*q^2 + O(q^3) 1/2 + q + q^2 + O(q^3)   1/2 + 3*q^2 + O(q^3)]
[           1/6 + O(q^3)      1/6 + q^2 + O(q^3)        1/6 + q + O(q^3)]
```

References

- [Pi] Arnold Pizer, *An Algorithm for Computing Modular Forms on* $\Gamma_0(N)$
- [Ko] David Kohel, *Hecke Module Structure of Quaternions*

Further Examples

We decompose a Brandt module over both **Z** and **Q**.:

```
sage: B = BrandtModule(43, base_ring=ZZ); B
Brandt module of dimension 4 of level 43 of weight 2 over Integer Ring
sage: D = B.decomposition()
sage: D
[
Subspace of dimension 1 of Brandt module of dimension 4 of level 43 of weight 2 over Integer Ring,
Subspace of dimension 1 of Brandt module of dimension 4 of level 43 of weight 2 over Integer Ring,
Subspace of dimension 2 of Brandt module of dimension 4 of level 43 of weight 2 over Integer Ring
]
sage: D[0].basis()
((0, 0, 1, -1),)
sage: D[1].basis()
((1, 2, 2, 2),)
sage: D[2].basis()
((1, 1, -1, -1), (0, 2, -1, -1))
sage: B = BrandtModule(43, base_ring=QQ); B
Brandt module of dimension 4 of level 43 of weight 2 over Rational Field
sage: B.decomposition()[2].basis()
((1, 0, -1/2, -1/2), (0, 1, -1/2, -1/2))
```

BrandtModule(*N, M=1, weight=2, base_ring=Rational Field, use_cache=True*)

Return the Brandt module of given weight associated to the prime power p^r and integer M, where p and M are coprime.

INPUT:
- N – a product of primes with odd exponents
- M – an integer coprime to q (default: 1)
- weight – an integer that is at least 2 (default: 2)
- base_ring – the base ring (default: QQ)
- use_cache – whether to use the cache (default: True)

OUTPUT:
- a Brandt module

EXAMPLES:

```
sage: BrandtModule(17)
Brandt module of dimension 2 of level 17 of weight 2 over Rational Field
sage: BrandtModule(17,15)
Brandt module of dimension 32 of level 17*15 of weight 2 over Rational Field
sage: BrandtModule(3,7)
Brandt module of dimension 2 of level 3*7 of weight 2 over Rational Field
sage: BrandtModule(3,weight=2)
```

```
Brandt module of dimension 1 of level 3 of weight 2 over Rational Field
sage: BrandtModule(11, base_ring=ZZ)
Brandt module of dimension 2 of level 11 of weight 2 over Integer Ring
sage: BrandtModule(11, base_ring=QQbar)
Brandt module of dimension 2 of level 11 of weight 2 over Algebraic Field
```

The use_cache option determines whether the Brandt module returned by this function is cached.:

```
sage: BrandtModule(37) is BrandtModule(37)
True
sage: BrandtModule(37,use_cache=False) is BrandtModule(37,use_cache=False)
False
```

TESTS:

Note that N and M must be coprime:

```
sage: BrandtModule(3,15)
...
ValueError: M must be coprime to N
```

Only weight 2 is currently implemented:

```
sage: BrandtModule(3,weight=4)
...
NotImplementedError: weight != 2 not yet implemented
```

Brandt modules are cached:

```
sage: B = BrandtModule(3,5,2,ZZ)
sage: B is BrandtModule(3,5,2,ZZ)
True
```

class **BrandtModuleElement** (*parent, x*)

monodromy_pairing (*x*)

Return the monodromy pairing of self and x.

EXAMPLES:

```
sage: B = BrandtModule(5,13)
sage: B.monodromy_weights()
(1, 3, 1, 1, 1, 3)
sage: (B.0 + B.1).monodromy_pairing(B.0 + B.1)
4
```

class **BrandtModule_class** (*N, M, weight, base_ring*)

A Brandt module.

EXAMPLES:

```
sage: BrandtModule(3, 10)
Brandt module of dimension 4 of level 3*10 of weight 2 over Rational Field
```

M ()

Return the auxiliary level (prime to p part) of the quaternion order used to compute this Brandt module.

EXAMPLES:

```
sage: BrandtModule(7,5,2,ZZ).M()
5
```

N()

Return ramification level N.

EXAMPLES:

```
sage: BrandtModule(7,5,2,ZZ).N()
7
```

brandt_series(*prec, var='q'*)

Return matrix of power series $\sum T_n q^n$ to the given precision. Note that the Hecke operators in this series are always over $\mathbf{Q}$, even if the base ring of this Brandt module is not $\mathbf{Q}$.

INPUT:
- prec – positive intege
- var – string (default: q)

OUTPUT: matrix of power series with coefficients in $\mathbf{Q}$

EXAMPLES:

```
sage: B = BrandtModule(11)
sage: B.brandt_series(2)
[1/4 + q + O(q^2)     1/4 + O(q^2)]
[    1/6 + O(q^2) 1/6 + q + O(q^2)]
sage: B.brandt_series(5)
[1/4 + q + q^2 + 2*q^3 + 5*q^4 + O(q^5)   1/4 + 3*q^2 + 3*q^3 + 3*q^4 + O(q^5)]
[  1/6 + 2*q^2 + 2*q^3 + 2*q^4 + O(q^5)         1/6 + q + q^3 + 4*q^4 + O(q^5)]
```

Asking for a smaller precision works.:

```
sage: B.brandt_series(3)
[1/4 + q + q^2 + O(q^3)   1/4 + 3*q^2 + O(q^3)]
[  1/6 + 2*q^2 + O(q^3)       1/6 + q + O(q^3)]
sage: B.brandt_series(3,var='t')
[1/4 + t + t^2 + O(t^3)   1/4 + 3*t^2 + O(t^3)]
[  1/6 + 2*t^2 + O(t^3)       1/6 + t + O(t^3)]
```

cyclic_supermodules(*I, p*)

Return a list of rescaled versions of the fractional right ideals J such that J contains I and the quotient has group structure the product of two cyclic groups of order p.

We emphasize again that J is rescaled to be integral.

INPUT: I – ideal I in R = self.order_of_level_N() p – prime p coprime to self.level()

OUTPUT: list of the p+1 fractional right R-ideals that contain I such that J/I is GF(p) x GF(p).

EXAMPLES:

```
sage: B = BrandtModule(11)
sage: I = B.order_of_level_N().unit_ideal()
sage: B.cyclic_supermodules(I, 2)
[Fractional ideal (1/2 + 3/2*j + k, 1/2*i + j + 1/2*k, 2*j, 2*k),
 Fractional ideal (1/2 + 1/2*i + 1/2*j + 1/2*k, i + k, j + k, 2*k),
 Fractional ideal (1/2 + 1/2*j + k, 1/2*i + j + 3/2*k, 2*j, 2*k)]
sage: B.cyclic_supermodules(I, 3)
[Fractional ideal (1/2 + 1/2*j, 1/2*i + 5/2*k, 3*j, 3*k),
 Fractional ideal (1/2 + 3/2*j + 2*k, 1/2*i + 2*j + 3/2*k, 3*j, 3*k),
 Fractional ideal (1/2 + 3/2*j + k, 1/2*i + j + 3/2*k, 3*j, 3*k),
 Fractional ideal (1/2 + 5/2*j, 1/2*i + 1/2*k, 3*j, 3*k)]
sage: B.cyclic_supermodules(I, 11)
...
ValueError: p must be coprime to the level
```

eisenstein_subspace()

Return the 1-dimensional subspace of self on which the Hecke operators T_p act as $p+1$ for p coprime to the level.

NOTE: This function assumes that the base field has characteristic 0.

EXAMPLES:

```
sage: B = BrandtModule(11); B.eisenstein_subspace()
Subspace of dimension 1 of Brandt module of dimension 2 of level 11 of weight 2 over Rationa
sage: B.eisenstein_subspace() is B.eisenstein_subspace()
True
sage: BrandtModule(3,11).eisenstein_subspace().basis()
((1, 1),)
sage: BrandtModule(7,10).eisenstein_subspace().basis()
((1, 1, 1, 1/2, 1, 1, 1/2, 1, 1, 1),)
sage: BrandtModule(7,10,base_ring=ZZ).eisenstein_subspace().basis()
((2, 2, 2, 1, 2, 2, 1, 2, 2, 2),)
```

free_module()

Return the underlying free module of the Brandt module.

EXAMPLES:

```
sage: B = BrandtModule(10007,389)
sage: B.free_module()
Vector space of dimension 325196 over Rational Field
```

maximal_order()

Return a maximal order in the quaternion algebra associated to this Brandt module.

EXAMPLES:

```
sage: BrandtModule(17).maximal_order()
Order of Quaternion Algebra (-17, -3) with base ring Rational Field with basis (1/2 + 1/2*j,
sage: BrandtModule(17).maximal_order() is BrandtModule(17).maximal_order()
True
```

monodromy_weights()

Return the weights for the monodromy pairing on this Brandt module. The weights are associated to each ideal class in our fixed choice of basis. The weight of an ideal class $[I]$ is half the number of units of the right order I.

NOTE: The base ring must be $\mathbf{Q}$ or $\mathbf{Z}$.

EXAMPLES:

```
sage: BrandtModule(11).monodromy_weights()
(2, 3)
sage: BrandtModule(37).monodromy_weights()
(1, 1, 1)
sage: BrandtModule(43).monodromy_weights()
(2, 1, 1, 1)
sage: BrandtModule(7,10).monodromy_weights()
(1, 1, 1, 2, 1, 1, 2, 1, 1, 1)
sage: BrandtModule(5,13).monodromy_weights()
(1, 3, 1, 1, 1, 3)
```

order_of_level_N()

Return Eichler order of level N = p^(2*r+1)*M in the quaternion algebra.

EXAMPLES:

```
sage: BrandtModule(7).order_of_level_N()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis (1/2 + 1/2*j,
sage: BrandtModule(7,13).order_of_level_N()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis (1/2 + 1/2*j +
sage: BrandtModule(7,3*17).order_of_level_N()
Order of Quaternion Algebra (-1, -7) with base ring Rational Field with basis (1/2 + 1/2*j +
```

quaternion_algebra()

Return the quaternion algebra A over QQ ramified precisely at p and infinity used to compute this Brandt module.

EXAMPLES:

```
sage: BrandtModule(997).quaternion_algebra()
Quaternion Algebra (-2, -997) with base ring Rational Field
sage: BrandtModule(2).quaternion_algebra()
Quaternion Algebra (-1, -1) with base ring Rational Field
sage: BrandtModule(3).quaternion_algebra()
Quaternion Algebra (-1, -3) with base ring Rational Field
sage: BrandtModule(5).quaternion_algebra()
Quaternion Algebra (-2, -5) with base ring Rational Field
sage: BrandtModule(17).quaternion_algebra()
Quaternion Algebra (-17, -3) with base ring Rational Field
```

right_ideals(*B=None*)

Return sorted tuple of representatives for the equivalence classes of right ideals in self.

OUTPUT: • sorted tuple of fractional ideals

EXAMPLES:

```
sage: B = BrandtModule(23)
sage: B.right_ideals()
(Fractional ideal (2 + 2*j, 2*i + 2*k, 4*j, 4*k),
 Fractional ideal (2 + 2*j, 2*i + 6*k, 8*j, 8*k),
 Fractional ideal (2 + 10*j + 8*k, 2*i + 8*j + 6*k, 16*j, 16*k))
```

TEST:

```
sage: B = BrandtModule(1009)
sage: Is = B.right_ideals()
sage: n = len(Is)
sage: prod(not Is[i].is_equivalent(Is[j]) for i in range(n) for j in range(i))
1
```

class **BrandtSubmodule**(*ambient, submodule, dual_free_module=None, check=True*)

basis_for_left_ideal(*R, gens*)

Return a basis for the left ideal of R with given generators.

INPUT: • R – quaternion order
• gens – list of elements of R

OUTPUT: • list of four elements of R

EXAMPLES:

```
sage: B = BrandtModule(17); A = B.quaternion_algebra(); i,j,k = A.gens()
sage: sage.modular.quatalg.brandt.basis_for_left_ideal(B.maximal_order(), [i+j,i-j,2*k,A(3)])
[1/2 + 1/6*j + 2/3*k, 1/2*i + 1/2*k, 1/3*j + 1/3*k, k]
sage: sage.modular.quatalg.brandt.basis_for_left_ideal(B.maximal_order(), [3*(i+j),3*(i-j),6*k,A
[3/2 + 1/2*j + 2*k, 3/2*i + 3/2*k, j + k, 3*k]
```

benchmark_magma(*levels, silent=False*)

INPUT: • levels – list of pairs (p,M) where p is a prime not dividing M
• silent – bool, default False; if True surpress printing during computation

OUTPUT: • list of 4-tuples ('magma', p, M, tm), where tm is the CPU time in seconds to compute T2 using Magma

EXAMPLES:

```
sage: a = sage.modular.quatalg.brandt.benchmark_magma([(11,1), (37,1), (43,1), (97,1)])  # optic
('magma', 11, 1, ...)
('magma', 37, 1, ...)
('magma', 43, 1, ...)
('magma', 97, 1, ...)
sage: a = sage.modular.quatalg.brandt.benchmark_magma([(11,2), (37,2), (43,2), (97,2)])  # optic
('magma', 11, 2, ...)
('magma', 37, 2, ...)
('magma', 43, 2, ...)
('magma', 97, 2, ...)
```

benchmark_sage(*levels, silent=False*)

INPUT: • levels – list of pairs (p,M) where p is a prime not dividing M
• silent – bool, default False; if True surpress printing during computation

OUTPUT: • list of 4-tuples ('sage', p, M, tm), where tm is the CPU time in seconds to compute T2 using Sage

EXAMPLES:

```
sage: a = sage.modular.quatalg.brandt.benchmark_sage([(11,1), (37,1), (43,1), (97,1)])
('sage', 11, 1, ...)
('sage', 37, 1, ...)
('sage', 43, 1, ...)
('sage', 97, 1, ...)
sage: a = sage.modular.quatalg.brandt.benchmark_sage([(11,2), (37,2), (43,2), (97,2)])
('sage', 11, 2, ...)
('sage', 37, 2, ...)
('sage', 43, 2, ...)
('sage', 97, 2, ...)
```

class_number(*p, r, M*)

Return the class number of an order of level N = p^r*M in the quaternion algebra over QQ ramified precisely at p and infinity.

This is an implementation of Theorem 1.12 of [Pizer, 1980].

INPUT: • p – a prime
• r – an odd positive integer (default: 1)
• M – an integer coprime to q (default: 1)

OUTPUT: Integer

EXAMPLES:

```
sage: sage.modular.quatalg.brandt.class_number(389,1,1)
33
sage: sage.modular.quatalg.brandt.class_number(389,1,2)  # TODO -- right?
97
sage: sage.modular.quatalg.brandt.class_number(389,3,1)  # TODO -- right?
4892713
```

maximal_order(*A*)

Return a maximal order in the quaternion algebra ramified at p and infinity.

This is an implementation of Proposition 5.2 of [Pizer, 1980].

INPUT: • A – quaternion algebra ramified precisely at p and infinity

OUTPUT: • a maximal order in A

EXAMPLES:

```
sage: A = BrandtModule(17).quaternion_algebra()
sage: sage.modular.quatalg.brandt.maximal_order(A)
Order of Quaternion Algebra (-17, -3) with base ring Rational Field with basis (1/2 + 1/2*j, 1/2
```

quaternion_order_with_given_level (*A, level*)

Return an order in the quaternion algebra A with given level. (Implemented only when the base field is the rational numbers.)

INPUT: **level – The level of the order to be returned. Currently this is only implemented** when the level is divisible by at most one power of a prime that ramifies in this quaternion algebra.

EXAMPLES:

```
sage: from sage.modular.quatalg.brandt import quaternion_order_with_given_level, maximal_order
sage: A.<i,j,k> = QuaternionAlgebra(5)
sage: level = 2 * 5 * 17
sage: O = quaternion_order_with_given_level(A, level)
sage: M = maximal_order(A)
sage: L = O.free_module()
sage: N = M.free_module()
sage: print L.index_in(N) == level/5  #check that the order has the right index in the maximal o
True
```

right_order (*R, basis*)

Given a basis for a left ideal I, return the right order in the quaternion order R of elements such that I*x is contained in I.

INPUT: • R – order in quaternion algebra

• basis – basis for an ideal I

OUTPUT: • order in quaternion algebra

EXAMPLES:

We do a consistency check with the ideal equal to a maximal order.:

```
sage: B = BrandtModule(17); basis = sage.modular.quatalg.brandt.basis_for_left_ideal(B.maximal_o
sage: sage.modular.quatalg.brandt.right_order(B.maximal_order(), basis)
Order of Quaternion Algebra (-17, -3) with base ring Rational Field with basis (1/2 + 1/6*j + 2/
sage: basis
[1/2 + 1/6*j + 2/3*k, 1/2*i + 1/2*k, 1/3*j + 1/3*k, k]

sage: B = BrandtModule(17); A = B.quaternion_algebra(); i,j,k = A.gens()
sage: basis = sage.modular.quatalg.brandt.basis_for_left_ideal(B.maximal_order(), [i*j-j])
sage: sage.modular.quatalg.brandt.right_order(B.maximal_order(), basis)
Order of Quaternion Algebra (-17, -3) with base ring Rational Field with basis (1/2 + 1/2*i + 1/
```

CHAPTER

FORTYSEVEN

HISTORY AND LICENSE

Sage was initially created by William Stein in 2004–2005, using Python, IPython, PARI, SWIG, Pyrex, NTL, and GMP. These programs are all open source and released under the GPL or a GPL-compatible license.

All Sage releases are released under the GPL. All documentation is released under the GNU Free Documentation License.

47.1 The GNU General Public License

Version 2, June 1991

```
Copyright (C) 1989, 1991 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA  02110-1301, USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.
```

47.1.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

47.1.2 Terms and Conditions For Copying, Distribution and Modification

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

   You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   (a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   (b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   (c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

3. These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   (a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   (c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

10. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

1. Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. Except when otherwise stated in writing the copyright holders and/or other parties provide the program "as is" without warranty of any kind, either expressed or implied, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

2. In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

End of Terms and Conditions

47.1.3 Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and a brief idea of what it
does.
Copyright (C) yyyy name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License as
published by the Free Software Foundation; either version 2 of the
License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but
WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
02110-1301, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) yyyy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type
'show w'.
This is free software, and you are welcome to redistribute it under
certain conditions; type 'show c' for details.
```

The hypothetical commands show w and show c should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than show w and show c; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
program
'Gnomovision' (which makes passes at compilers) written by James
Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

CHAPTER

FORTYEIGHT

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

BIBLIOGRAPHY

[CMR05] C. Cid, S. Murphy, M. Robshaw *Small Scale Variants of the AES*; in Proceedings of Fast Software Encryption 2005; LNCS 3557; Springer Verlag 2005; available at http://www.isg.rhul.ac.uk/~sean/smallAES-fse05.pdf

[CMR06] C. Cid, S. Murphy, and M. Robshaw *Algebraic Aspects of the Advanced Encryption Standard*; Springer Verlag 2006

[MR02] S. Murphy, M. Robshaw *Essential Algebraic Structure Within the AES*; in Advances in Cryptology - CRYPTO 2002; LNCS 2442; Springer Verlag 2002

[BPW06] J. Buchmann, A. Pychkine, R.-P. Weinmann *Block Ciphers Sensitive to Groebner Basis Attacks* in Topics in Cryptology – CT RSA'06; LNCS 3860; pp. 313–331; Springer Verlag 2006; pre-print available at http://eprint.iacr.org/2005/200

[CB07] Nicolas T. Courtois, Gregory V. Bard *Algebraic Cryptanalysis of the Data Encryption Standard*; Cryptography and Coding – 11th IMA International Conference; 2007; available at http://eprint.iacr.org/2006/402

[Heys02] H. Heys *A Tutorial on Linear and Differential Cryptanalysis* ; 2002' available at http://www.engr.mun.ca/~howard/PAPERS/ldc_tutorial.pdf

[PRESENT07] A. Bogdanov, L. Knudsen, G. Leander, C. Paar, A. Poschmann, M. Robshaw, Y. Seurin, C. Vikkelsoe *PRESENT: An Ultra-Lightweight Block Cipher*; in Proceedings of CHES 2007; LNCS 7427; pp. 450-466; Springer Verlag 2007; available at http://www.crypto.rub.de/imperia/md/content/texte/publications/conferences/present_ches2007.pdf

[BC03] A. Biryukov and C. D. Canniere *Block Ciphers and Systems of Quadratic Equations*; in Proceedings of Fast Software Encryption 2003; LNCS 2887; pp. 274-289, Springer-Verlag 2003.

[HPS08] J. Hoffstein, J. Pipher, and J.H. Silverman. *An Introduction to Mathematical Cryptography*. Springer, 2008.

[ZBN97] C. Zhu, R. H. Byrd and J. Nocedal. L-BFGS-B: Algorithm 778: L-BFGS-B, FORTRAN routines for large scale bound constrained optimization. ACM Transactions on Mathematical Software, Vol 23, Num. 4, pp.550–560, 1997.

[MF99] J.H. Mathews and K.D. Fink. *Numerical Methods Using MATLAB*. 3rd edition, Prentice-Hall, 1999.

[BF05] R.L. Burden and J.D. Faires. *Numerical Analysis*. Thomson Brooks/Cole, 8th edition, 2005.

[AB2007] M. Aschenbrenner, C. Hillar, Finite generation of symmetric ideals. Trans. Amer. Math. Soc. 359 (2007), no. 11, 5171–5192.

[AB2008] M. Aschenbrenner, C. Hillar, An Algorithm for Finding Symmetric Groebner Bases in Infinite Dimensional Rings.

[AB2007] M. Aschenbrenner, C. Hillar, Finite generation of symmetric ideals. Trans. Amer. Math. Soc. 359 (2007), no. 11, 5171–5192.

[Laz92] Daniel Lazard, *Solving Zero-dimensional Algebraic Systems*, in Journal of Symbolic Computation (1992) vol. 13, pp. 117-131

[BD07] Michael Brickenstein, Alexander Dreyer; *PolyBoRi: A Groebner basis framework for Boolean polynomials*; pre-print available at http://www.itwm.fraunhofer.de/zentral/download/berichte/bericht122.pdf

MODULE INDEX

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z