THE IMPLEMENTATION OF THE POLYOMINO CLASS INTO SAGE

Neil Johnson
Nasim Shomali

Polyominoes

Introduction

A polyomino is a plane geometric figure formed by joining one or more square polyforms edge to edge. Polyominoes are characterized by the number of square cells per figure. Since 1907, polyominoes have been incorporated into popular puzzles. With respect to mathematics, polyominoes have raised numerous combinatorial problems such as enumerating polyominoes of a given size. Although there has been extensive algorithms devised for the enumeration of polyominoes there currently is no standard formula known to calculate the exact number of polyominoes that could be produced for a given number of square cells.

As of today, the largest enumeration of polyominoes is for n =56, giving a total of 6915 X 10^{28} different combinations that could be produced. This is a huge number compared to the number of polyominoes that could be produced for n =2, which is 2. This is because it has been proven that $\lim n \to \infty An\ 1n = \lambda$ where A_n is approximated by: $A_n \sim .3169 * (4.0626^n) / n$ such that the number of polyominoes grows exponentially. It a separate study, it has been written that the bounds for the number of n-polyominoes is $3.72^n < A(n) < 4.65^n$.

Polyominoes can also be classified as fixed and free, with and without holes, and one-sided. Free polyominoes are defined such that the mirror image pieces are identical, fixed polyominoes are defined such that if they have different chirality or orientation they are considered distinct, and one-sided polyominoes are defined such that they may not be flipped but can be rotated so different rotational orientations are the same but ones with different chirality are distinct. Polyominoes with holes indicate that there is a region within the interior of the polyomino that is not covered fully by a square piece.

Polyominoes have influenced popularized games such as Tetris, Sudoku, and the board game Blokus. Although there is little research left in the field of polyominoes today, this is still an area of fascination to some combinatorial mathematicians such as Professor Michael Reid at the University of Central Florida where his research is focused on the titling rectangles of half strips with congruent polyominoes.

Motivation

The inspiration for our final project was derived from the complexity of polyominoes. We saw that there have not been any algorithms, graphical interfaces, or classes of polyominoes incorporated into Sage yet. Thus, our final project was to devise a class titled Polyominoes that would allow for the creation, manipulation, and enumeration of polyominoes in Sage. Included with the code, is extensive documentation describing the necessity of each function we included and how each function could be called to. Our goal was to implement a fast enough (via implementation in Cython) program that it could eventually be incorporated into the Sage library.

The following table lists the name, parameters, and a short description of what each function is able to do. Detailed examples of each function are included in the attached .sws file called PolyomonioExamples.Sws. Due to the excessive nature of our code, this table along with the detailed examples and testing shown in the attached Sage worksheet, are in lieu of Sage documentation.

Table 1: Reference table for all functions in the Polyomino and PolyominoSet classes

Name of function	Description	Inputs and Outputs
CLASS POLYOMINO		
def__init__(self, piece = [], color =1)	This is the "constructor" that creates a polyomino based on the matrix you pass. The user can also specify the color of the polyomino based on any positive integers greater than zero	Input: An (mxn) matrix of specificying the where the squares are located represented by 1's and where the holes are represented by 0's. The second parameter can be any positive integer which is represents the color of the polyomino Output: A polyomino based on the specification above
def __len__ (self)	Determines the length of the polyomino	Input: 1 polyomino Output: Returns the number of squares in the polyomino
def __eq__ (self, other)	Checks if two polyominoes are equivalent	Input: 2 polyominoes Output: Returns true if the 2 polyominoes are the same height and width regardless of color. Returns false otherwise
def __lt__(self, other)	Checks if the second polyomino is less than the first polyomino	Input: 2 polyominoes Output: If the widths of the 2 polyominoes are the same, then it returns true if and only if the height of the first polyomino is greater than the height of the second polyomino. Otherwise returns false
def __gt__(self, other)	Checks if the second polyomino is greater than the first polyomonio	Input: 2 polyominoes Output: If the widths of the 2 polyominoes are the same, then it returns true if and only if the height of the first polyomino is less than the height of the second polyomino. Otherwise returns false
def __le__(self, other)	Checks if the second polyomino is less than or equal to the first polyomino	Input: 2 polyominoes Output: If the widths of the 2 polyominoes are the same, then it returns true if and only if the height of the first polyomino is greater than or equal to the height of the second polyomino. Otherwise returns false
def __ge__(self, other)	Checks if the second polyomino is greater than or equal to the first polyomino	Input: 2 polyominoes Output: If the widths of the 2 polyominoes are the same, then it returns true if and only if the height of the first polyomino is less than or equal to the height of the second polyomino. Otherwise returns false
def __ne__(self, other)	Checks if the two polyominoes are not equal	Input: 2 polyominoes Output: If the widths of the 2 polyominoes are not equal (regardless of the heights) then it will return true. Otherwise it will return false

def __add__(self, other)	Adds the two polyominoes together by creating a new polyomino set	Input: 2 polyominoes Output: Creates a new polyomino set that adds the first polyomino then adds the second polyomino to the set
def __str__(self)	Prints out the string representation of the polyomino	Input: 1 polyomino Output: Returns the string representation of the polyomino.
def __repr__(self)	Prints out the polyomino as a nested list which is the form that the __init__() of the polyomino requires	Input: 1 polyomino Output: Prints out the polyomino as a nested list which is the form that the __int__() of the polyomino requires
def __copy__(self)	Creates a copy of the given polyomino that can be altered	Input: 1 polyomino Output: Returns a copy of the polyomino that can be altered
def __getitem__(self, n)	Returns the row element of the polyomino at row n	Input: 1 polyomino and integer referring to the row Output: Returns the row element of the polyomino at row n as a list while maintaining the color. If the height and width position is too big then it returns an index error.
def _isConnected(self,h,w)	Returns true if there is an adjacent square that is not blank.	Input: Polyomino, height position, width position Output: Returns true if there is an adjacent square that is not blank. Otherwise returns false. If the height and width position is too big then it returns an index error. If it is 1x1 polyomino then it will return false.
def isCompatibleWith(self, other)	Returns two polyomino grids that produce the same resulting shape after tiling with themselves	Input: Two Polyominos Output: Two polyomino grids that are each composed of the first and second polyominoes respectively that produce the same resulting shape after tiling with themselves.
def isCompatibleWith2(self, obj, gridList, newGridlist2 = None)	This is a helper method that operates with the isCompatibleWidth() method	Input: A polyomino (self), a polyomino, a list of polyomino grids, and an optional second list of polyomino grids Output: Returns a list of lists of grids that are all the possible combinations of the polyomino and the grids in the required list.
def filledSpaces(self)	Returns the filled positions (height, width) of the polyomino	Input: A polyomino Output: Returns a list of height and width pairs that filled in the given polyomino
def openSpaces(self,	Returns the not filled	Input: A polyomino, a boolean

expand = true)	positions (height, width) of the polyomino and the adjacent pieces.	Output: Returns a list of height and width pairs that are not filled and adjacent to an existing square in the given polyomino. If expand is true it also returns the pairs that are surrounding the piece otherwise it does not.
def insertColumn(self,index=None,column=None, newCopy = false)	Inserts a column to the given polyominio	Input: Polyomino, (index where to insert column, the column, whether to add the column to a copy or the original polyomino) Output: Adds the given column to the specified index. If no index is given it adds the column to the far right. If no column is given, it inserts a column of zeros.
def insertRow(self,index=None,row=None, newCopy = false)	Inserts a row to the given polyomino	Input: Polyomino, (index where to insert row, the row, whether to add the row to a copy or the original polyomino) Output: Adds the given row to the specified index. If no index is given it adds the row to the far right. If no row is given, it inserts a row of zeros.
def rotate(self,direction=1)	Rotates a polyomino by the given number	Input: A polyomino and an integer Output: A copy of the polyomino rotated by the integer specified. It no integer is given it will rotate the copy of the polyomino once to the right. Positive integers rotate the polyomino to the right. Negative integers rotate the polyomino to the left.
def setColor(self,color=1, RGBtuple=None)	Sets the numerical color and the rgb color of the given polyomino.	Input: A polyomino, integer representation of the color Output: Sets the color of the polyomino to the integer specified. If no color is specified, then it sets the color to "1".
def getColorNum(self)	Returns the numerical value for the color of the given polyomino	Input: A polyomino Output: Returns an integer which indicates what color the polyomino is
def getColorRGB(self)	Returns the RGB tuple of the given Polyomino	Input: A polyomino Output: Tuple representing the RGB values of the polyomino's color
def getColorStr(self)	Returns the symbol used when printing the polyomino	Input: A polyomino Output: symbol pertaining to the color of the polyomino.
def getWidth(self)	Returns the number of rows	Input: A polyomino Output: Returns the number of rows in

		the polyomino
def getHeight(self)	Returns the number of columns	Input: A polyomino Output: Returns the number of columns in the polyomino
def getXY(self,x,y)	Returns the number of rows and columns	Input: A polyomino, row and column position Output: Returns the number of rows and columns in the polyomino.
def addXY(self,x,y)	Adds a square of the same color at the position given by x and y	Input: A polyomino, row, and column position Output: Makes a copy of the given polyomino and adds a square of the same color to the given position
def flipV(self)	Vertically flips a copy of the polyomino	Input: A polyomino Output: Makes a copy of the given polyomino and vertically flips the copy version
def flipH(self)	Horizontally flips a copy of the polyomino	Input: A polyomino Output: Makes a copy of the given polyomino and horizontally flips the copy version
def isSymmetricV(self)	Returns true if the polyomino is vertically symmetric	Input: A polyomino Output: Returns true if the polyomino is vertically symmetric
def isSymmetricH(self)	Returns true if the polyomino is horizontally symmetric	Input: A polyomino Output: Returns true if the polyomino is horizontally symmetric
def isSymmetricR(self, n =1)	Returns true if the polyomino is rotationally symmetric	Input: A polyomino, an integer representing the number of rotations required Output: Returns true if the polyomino is rotationally symmetric after n rotations or (1 if no integer is specified)
def isSymmetric(self)	Returns true if the polyomino is both horizontally and vertically symmetric	Input: A polyomino Output: Returns true if the polyomino is horizontally and vertically symmetric
def hasHole(self)	Determines if there is a hole in the polyomino	Input: A polyomino Output: Returns true if there a tile that has no other tiles adjacent to it (a hole) else it returns false
def createPolyomino(self, grid = None, color =1)	Creates a polyomino of a specific color. If no grid is specified, it assumes the grid of "self"	Input: A polyomino, a grid for the new polyomino , an integer representing the color Output: Creates a polyomino piece of color 1 (if no color is specified).

def generateDrawingJS(self, cellID,size, border_width, xPos, yPos)	Works with the draw() method by receiving a canvas ID and returning a script which will draw that piece on that canvas with the specified parameters passed.	Input: A polyomino, an integer representing a cell whose outputs houses the canvas on which it is to draw, integers representing the size, border width, x-position, and y-position respectively. Output: A string of JavaScript that is intended to draw the shape on the canvas specified
def draw(self,size=35,borderWidth=None,padding=20, innerBorder = false, outerBorder = true, CellID= None, boxBorder = 2)	Draws the polyomino using JavaScript in an HTML canvas	Input: A polyomino, three integers representing size in pixels of the height and width of each square, border width, and padding (white space surrounding piece), boolean representing innerBorder and outerBorder, a cellID, and box border thickness. Output: Draws the polyomino using JavaScript in a HTML canvas
CLASS POLYOMINOSET		
def __init__ (self , set=None)	This is the "constructor" that creates a polyomino set. It no set is passed, it adds the polyomino to a new set. If an integer is passed is given, it creates a set of polyominoes with n pieces. If a list of lists is passed, it adds the polyomino to the list.	Input: A polyomino, an integer, a list of lists, list of integers, a polyomino set that has already been created, a list of lists of lists of integers, or a polyomino and no second input. Output: Adds the polyomino to a set regardless of polyomino size or color.
def __getitem__(self,n)	Returns the polyomino that is in position n of the et	Input: 1 polyomino and a non-negative integer referring to the position of the polyomino. Output: Returns the polyomino at position n.
def __add__(self,other)	Returns a new polyomino set that includes both polyomino sets passed	Input: 2 PolyominoSets Output: Returns a new polyomino set that includes both polyomino sets passed
def __sub__(self,other)	Subtracts the pieces of the 2nd polyomino set from the 1st polyomino set regardless of color.	Input: 2 PolyominoSets Output: Returns a new polyomino set that consists of the 1st polyomino set pieces excluding the pieces of the 2nd polyomino set regardless of color.
def __mul__(self,n)	Returns a new polyomino set that contains n-copies	Input: A PolyominoSet , an integer Output: Returns a new polyomino set that

	of the passed polyomino set.	contains n-copies of the passed polyomino set.
def __len__(self)	Returns the number of polyominoes in the polyomino set	Input: A PolyominoSet Output: Returns the number of polyominoes the polyomino set
def __repr__(self)	Prints out the polyomino set as a nested list which is the form that the __init__() of the polyomino set requires	Input: A PolyominoSet Output: Prints out the the polyomino set as a nested list which is the form that the __int__() of the polyomino set requires
def __str__(self)	Prints out the string representation of the polyomino set	Input: A PolyominoSet Output: Returns the string representation of the polyomino set using the different shaded box characters
def Print(self)	Prints the polyomino set ensuring that the output is not truncated and with no line wrapping	Input: A PolyominoSet Output: Prints out the string of the polyomino as HTML
def strSettings(self, *args , **kwds)	Allows the user to change the settings for the printing of the polyominoes	Input: A PolyominoSet, any setting for the strings. The choices of the settings you can set include: peiceSeperator which is the breaker between the pieces when printing out, printNumbs which is whether or not print the polyominoes as numbers, printSquares which is whether or not to print the polyominoes as squares. Output: None
def __copy__(self)	Returns a copy of the polyomino set	Input: A PolyominoSet Output: Returns a copy of the polyomino set
def maxHeight(self)	Returns the number indicating the height of the tallest polyomino in the polyomino set	Input: A PolyominoSet Output: Returns the number indicating the height of the tallest polyomino in the polyomino set
def maxWidth(self)	Returns the number indicating the width of the widest polyomino in the polyomino set	Input: A PolyominoSet Output: Returns the number indicating the width of the widest polyomino in the polyomino set
def genSet(self,n,filters=[],fixed=false,p=[[1]])	Clears the initial set and generates a new set based on the filter settings passed as a list of strings.	Input: A PolyominoSet, an integer representing the number of pieces, filter settings, boolean representing whether or not polyominoes that are rotationally different should be included, and the initial polyomino that the set is build off of. Output: Clears the initial set and generates

		a new set based on the parameters passed.
def addSet(self,n,filters=[],fixed=false,p=[[1]])	Adds the polyomino set passed to the given set.	Input: A PolyominoSet, an integer representing the number of pieces, filter settings, and a Boolean representing whether or not polyominoes that are rotationally different should be included, and the initial polyomino that the set is build off of. Output: Adds the generated polyomino set to the existing set.
def genSet2(self,n,filters,fixed,p)	Works as a helper function to genSet() and addSet().	Input: A PolyominoSet, an integer representing the number of squares in each piece, filter settings, and a boolean representing whether or not polyominoes that are rotationally different should be included, and the initial polyomino that the set is build off of. Output: Generates the polyomino set with the given parameters.
def addPiece(self,p,fixed=false,duplicates=true)	Adds the passed polyomino piece to the passed polyomino set.	Input: A PolyominoSet, initial polyomino that the set is build off of, a boolean representing whether or not polyominoes that are rotationally different should be included, and a boolean representing whether or not duplicates should be included. Output:
def makeTall(self)	Makes all the polyominoes in the set as thin and tall as they could be via rotation	Input: A PolyominoSet Output: Returns the polyomino set with all the pieces that are wider than they are tall and rotates them
def makeWide(self)	Makes all the polyominoes in the set as wide and short as they could be via rotation	Input: Output: Returns the PolyominoSet with all the pieces that are taller than they are wide and rotates them
def setColor(self,n)	Returns the polyomino set with all the polyomino pieces set to the color represented by n	Input: A PolyominoSet, a non-negative integer Output: Returns the PolyominoSet with all the polyomino pieces set to the color represented by n
def clearSet(self)	Returns an empty polyomino set	Input: A PolyominoSet Output: Clears the polyomino set
def movePiece(self,index,newIndex)	Given the polyomino set, it moves a polyomino piece from position	Input: A PolyominoSet, an index integer , and a new index integer Output: Given the polyomino set, it

	"index" to "newindex" and shifts the rest of the polyomino set accordingly.	moves a polyomino piece from position "index" to "newindex" and shifts the rest of the polyomino set accordingly.
def sort(self,criteria="width", ascending=true, descending=false)	Returns the sorted polyomino set by the criteria given and either in ascending or descending order.	Input: A PolyominoSet, sorting criteria, with an option of setting ascending and descending options. Output: Returns the sorted PolyominoSet by the criteria given and either in ascending or descending order. If no criteria are given, then it sorts by width. If order is not given then it sorts in ascending manner.
def draw(self, size = 35, borderWidth = None, padding = 20, innerBorder = false, outerBorder = true, cellID = None, boxBorder = 2, boxMaxWidth = 900)	Draws the PolyominoSet in JavaScript with and HTML canvas	Input: A PolyominoSet, eight integers representing size in pixels of the height and width of each square, border width, and padding (white space surrounding piece), boolean representing innerBorder and outerBorder, box border thickness, cell ID, and box max width. If no borderWidth is given it is automatically set to size * .09. Output: Draws the polyomino set in JavaScript with an HTML canvas
PolyominoGrid Class		
def__init__(self, grid=None, h = 0, w= 0)	Creates a PolyominoGrid	Input: an existing grid, a polyomino, the height and width as integers. Output: A PolyominoGrid
def __len__ (self)	The number of polyominoes currently on the grid	Input: A PolyominoGrid Output: Returns the number of polyominoes on the grid
def __eq__ (self, other)	Checks if two PolyominoGrids are equivalent ignoring color	Input: Two PolyominoGrids Output: Returns true if the 2 PolyominoGrids are the same height and width regardless of color. Returns false otherwise
def __ne__(self, other)	Checks if the two PolyominoGrids are not equal ignoring color	Input: Two PolyominoGrids Output: If the widths of the 2 PolyominoGrids are not equal then it will return true. Otherwise it will return false
def__hash__(self)	Sets all the numbers in the grid that are not equal to zero to one and hashes as 2D list	Input: A PolyominoGrid Output: Integer representing the hash of the polyomino grid
def __str__(self)	Prints out the string representation of the	Input: A PolyominoGrid Output: Returns the string representation

	PolyominoGrid	of the PolyominoGrid using various squares as colors.
def __repr__(self)	Prints out the polyomino grid as a nested list which is the form that the __init__() of the PolyominoGrid requires	Input: A PolyominoGrid Output: Prints out the PolyominoGrid as a nested list which is the form that the __int__() of the PolyominoGrid requires
def __copy__(self)	Creates a copy of the given polyomino grid that can be altered without affecting the original polyomino grid	Input: A PolyominoGrid Output: Returns a copy of the polyomino that can be altered without affecting the original PolyominoGrid
def __getitem__(self, n)	Returns the row element of the polyomino grid at row n	Input: A PolyominoGrid and integer referring to the row Output: Returns the row element of the polyomino grid at row n as a list while maintaining the color. If the height and width position is too big then it returns an index error.
def binaryStr(self)	Replaces all the non-empty places in the grid with ones	Input: A PolyominoGrid Output: The current PolyominoGrid with all the non-empty places replaced with one
def _isConnected(self,h,w)	Returns true if there is an adjacent square that is not blank.	Input: A PolyominoGrid, height position, width position Output: Returns true if there is an adjacent square that is not blank. Otherwise returns false. If the height and width position is too big then it returns an index error. If it is 1x1 polyomino then it will return false.
def openSpaces(self, expand = true)	Returns the not filled positions (height, width) of the PolyominoGrid and the adjacent pieces.	Input: A PolyominoGrid, a boolean Output: Returns a list of height and width pairs that are not filled and adjacent to an existing square in the given PolyominoGrid. If expand is true it also returns the pairs that are surrounding the piece otherwise it does not.
def insertColumns(self, numOfColumns =1, index=None)	Inserts columns to the given PolyominoGrid	Input: A PolyominoGrid, integer for number of columns and an index where to insert the columns. Output: Adds the given column to the specified index. If no index is given it adds the column to the far right. The columns inserted are always of zeroes.
def insertRows(self, numOfRows =1,	Inserts rows to the given PolyominoGrid	Input: A PolyominoGrid, integer for number of rows and an index where to

index=None)		insert the rows. Output: Adds the given row to the specified index. If no index is given it adds the row to the far right. The rows inserted are always of zeroes.
def removeColumns(self, numOfColumns=1, index=None)	Removes columns from the given PolyominoGrid	Input: A PolyominoGrid, integer for number of rows and an index where to remove the columns. Output: Removes the column at the specified index and moves the rest of the columns to the left.
def removeRows(self, numOfRows =1, index=None)	Removes rows from the given PolyominoGrid	Input: A PolyominoGrid, integer for number of rows and an index where to remove the rows. Output: Removes the row at the specified index and moves the rest of the rows up.
def isSymmetricR(self)	Returns true if the PolyominoGrid is rotationally symmetric	Input: A PolyominoGrid Output: Returns true if the PolyominoGrid is rotationally symmetric after 2 rotations
def draw(self, other=None, size=35, borderWidth=None, padding=20, innerBorder=false, outerBorder=true, cellID=None, boxBorder=2)	Draws the grid as an HTML canvas using JavaScript. If the 'other' parameter is specified,	Input: A PolyominoGrid, a second PolyominoGrid, seven integers representing size in pixels of the height and width of each square, border width, and padding (white space surrounding piece), boolean representing innerBorder and outerBorder, cellID, and box border thickness. If no borderWidth is given it is automatically set to size * .09. Output: Draws the grid in JavaScript with an HTML canvas