Polygon – a Python package for handling 2D-polygons

Introduction

Polygon is a python package that handles polygonal shapes in 2D. It contains Python bindings for gpc, the excellent General Polygon Clipping Library by Alan Murta and some extensions written in C and pure Python. With Polygon you may handle complex polygonal shapes in Python in a very intuitive way. Polygons are simple Python objects, clipping operations are bound to standard operators like +, -, |, & and ^. TriStrips can be constructed from Polygons with a single statement. Functions to compute the area, center point, convex hull, point containment and much more are included. This package was already used to process shapes with more than one million points!

Author:	Joerg Raedler <joerg@j-raedler.de>
License:	LGPL (not for gpc itself, see below!)
Status:	Should be almost stable now, but there may be memory leaks. Malformed files and illegal contours may crash the library and your running Python interpreter! Use Polygon at your own risk or don't use it at all!
Homepage:	• Polygon: http://www.j-raedler.de/projects/polygon • gpc: http://www.cs.man.ac.uk/~toby/alan/software/

gpc is included in Polygon, you don't need to download it separately. I made two small changes to gpc:

1. fixed warnings regarding a printf format string and
2. made GPC_EPSILON adjustable, this may slow down the clipping a little bit.

The author of gpc (Alan Murta) is not responsible for this distribution! The wrapping and extension code is free software, but the core gpc library is free for non-commercial usage only. The author says:

> GPC is free for non-commercial use only. We invite non-commercial users to make a voluntary donation towards the upkeep of GPC. If you wish to use GPC in support of a commercial product, you must obtain an official GPC Commercial Use Licence from The University of Manchester.

Please respect this statement and contact the author (see gpc homepage) if you wish to use this software in commercial projects!

Platforms and Dependencies

There are two branches of the package. Versions 2.0.x works with python versions 2.x starting with python 2.5. Versions 3.0.x needs python 3.x. Both versions have almost identical features and APIs. You need a python interpreter from python.org, jython or ironpython will not work.

Polygon should work with recent versions of Linux, MacOS X and Windows systems. Not all combinations of operating system, python version and compiler could be tested.

Installation

Binaries

Please choose a binary distribution from the homepage that fits your operating system and python version. For a Linux system you should compile the package by yourself (see below).

From Source

Polygon uses the python distutils package. Please edit setup.py and adjust the values on top of the file. Run the command `python setup.py install` to compile and install the module. Make sure you have write permissions in pythons site-packages directory or use the --prefix or --home options!

You need a C-compiler to install Polygon from source. While this is usually no problem on Linux, it may be difficult to compile the module on Windows. I strongly recommend using the python-xy distribution which includes the free C-compiler MinGW32.

Basic Example

```
>>> import Polygon, Polygon.IO
>>> q = Polygon.Polygon(((0.0, 0.0), (10.0, 0.0), (10.0, 5.0), (0.0, 5.0)))
>>> t = Polygon.Polygon(((1.0, 1.0), (3.0, 1.0), (2.0, 3.0)))
>>> a = q - t  # gives a rectangle with a triangular hole
>>> Polygon.IO.writeSVG('test.svg', (a,))
```

Contents and Usage

The package consists of the main module (imported from cPolygon) and additional modules IO, Shapes and Utils.

Main module: the Polygon Class

The main module contains the following symbols:

Polygon(|arg):

the class for Polygon objects, see below

setDataStyle(style):

set data style, one of STYLE_TUPLE, STYLE_LIST or STYLE_NUMPY

setTolerance(tol):

set tolerance for detection of coincident nodes

getTolerance():

get tolerance for detection of coincident nodes

Error:

the exception raised when methods or operations fail

withNumPy:

flag showing if the support for NumPy is enabled

STYLE_TUPLE, STYLE_LIST, STYLE_ARRAY:

data style constants to be used with setDataStyle

__version__, __author__, __license__:

variables to hold meta information on the package

In this library a polygon object is a sequence of contours. A contour (a.k.a point list) is an ordered sequence of nodes (points) while a point is a 2-sequence of floats (x and y coordinates). If support for NumPy is enabled, a point list may be an array with the shape (i, 2) with i being the number of points.

A Polygon object may contain any number of normal contours (outline) and contours which describe a hole. Every contour has an associated 0/1 flag to specify a hole. The length of a Polygon object is the number of contours. You may access single contours with indexing ([]), slicing is not (yet) supported. You can't delete or change existing contours or single points of a polygon by assigning values. Please change the point lists before you apply them to a Polygon object instead.

Operations on Polygon Objects

(p and q are polygons)

p & q:	intersection: a polygon with the area that is covered by both p and q
p | q:	union: a polygon containing the area that is covered by p or q or both
p - q:	difference: a polygon with the area of p that is not covered by q
p + q:	sum: same as union
p ^ q:	xor: a polygon with the area that is covered by exactly one of p and q
len(p):	number of contours
p[i]:	contour with index i, the same as p.contour(i), slicing is not yet supported
bool(p):	logical value is true, if there are any contours in p (contours may be empty!)

Polygon(|arg)

Constructor: Create a new Polygon object. Files must contain data in gpc polygon format. If arg is another polygon, a copy is returned.

Arguments:

- a filename string and an optional holeflag (see read() and write()),
- or a readable file object (version 2.0.x only)
- or a pointlist (sequence of 2-sequences)
- or another polygon object

Returns: Polygon object

p.addContour(c |, hole=0)

Add a contour (outline or hole).

Arguments:

- c: pointlist (sequence of 2-tuples)
- optional hole: bool

Returns: None

p.contour(i)

gives the contour with index i (the same as p[i])

Arguments:

- i: integer

Returns: a contour

p.isHole(|i)

Returns the hole flag of a single contour (when called with index argument) or a list of all flags when called without arguments.

Arguments:

- optional i: integer

Returns: bool or list of bools

p.isSolid(|i)

Returns the inverted hole flag of a single contour (when called with index argument) or a list of all inverted flags when called without arguments.

Arguments:

- optional i: integer

Returns: bool or list of bools

p.nPoints(|i)

Returns the number of points of one contour or of the whole polygon. Is much faster than len(p[i]) or reduce(add, map(len, p))!

Arguments:
- optional i: integer

Returns: an integer

p.read(file)

Reads Polygon data from a file in gpc format. The file format has changed a while ago, hole flags can now be read and written (this is the default). Make sure you set the optional argument to False when reading files without this flag, otherwise the coordinates may not be correct!

Arguments:
- file: readable file object (version 2.0.x only) or filename string
- optional holeflag: bool

Returns: None

p.write(file)

Writes Polygon data to a file in gpc format. For the optional argument see above.

Arguments:
- file: writable file object (version 2.0.x only) or filename string
- optional holeflag: bool

Returns: None

p.simplify()

Try to simplify Polygon. It's possible to add overlapping contours or holes which are outside of other contours. This may result in wrong calculations of the area, center point, bounding box or other values. Call this method to make sure the Polygon is in a good shape. The method first adds all contours with a hole flag of 0, then substracts all holes and replaces the original Polygon with the result.

Arguments: None

Returns: None

p.area(|i)

Calculates the area of one contour (when called with index) or of the whole polygon. All values are positive! The polygon area is the sum of areas of all solid contours minus the sum of all areas of holes.

Arguments:
- optional i: integer

Returns: a float

p.center(|i)

Calculates the center of gravity of one contour (when called with index) or of the whole Polygon. The center may be outside the contours or inside holes. This is not the center of the bounding box!

Arguments:
- optional i: integer

Returns: a 2-tuple containing x and y float values

p.orientation(|i)

Calculates the orientation of one contour (when called with index) or of all contours. There's no default orientation, holes are defined by the hole flag, not by the orientation!

Arguments:
- optional i: integer

Returns: single integer or list of integers: 1 for ccw, -1 for cw, 0 for invalid contour.

p.isInside(x, y |, i)

Point containment test: returns logical containment value for a single contour (when called with index) or of the whole Polygon. If point is exactly on the border, the value may be True or False, sorry!

Arguments:
- x: float
- y: float
- optional i: integer

Returns: bool

p.covers(q)

Tests if the polygon completely covers the other polygon q. At first the bounding boxes are tested for obvious cases and then an optional clipping is performed.

Arguments:
- p: Polygon

Returns: bool

p.overlaps(q)

Tests if the polygon overlaps the other polygon q. At first the bounding boxes are tested for obvious cases and then an optional clipping is performed.

Arguments:
- p: Polygon

Returns: bool

p.boundingBox(|i)

Calculates the bounding box of one contour (when called with index) or of the whole polygon. In the latter case the data is cached and used for following calls and internal calculations. The data will be recalculated automatically when this method is called after the polygon has changed.

Arguments:
- optional i: integer

Returns: tuple of four floats: xmin, xmax, ymin and ymax

p.aspectRatio(|i)

Returns the aspect ratio (ymax-ymin) / (xmax-xmin) of the bounding box of one contour (when called with index) or of the whole polygon.

Arguments:
- optional i: integer

Returns: float

p.scale(xs, ys |, xc, yc)

Scales the polygon by multiplying with xs and ys around the center point. If no center is given the center point of the bounding box is used, which will not be changed by this operation.

Arguments:

- xs: float
- ys: float
- optional xc: float
- optional yc: float

Returns: None

p.shift(xs, ys)

Shifts the polygon by adding xs and ys.

Arguments:

- xs: float
- ys float

Returns: None

p.rotate(a |, xc, yc)

Rotates the polygon by angle a around center point in ccw direction. If no center is given the center point of the bounding box is used.

Arguments:

- a: float
- optional xc: float
- optional yc: float

Returns: None

p.warpToBox(x0, x1, y0, y1)

Scales and shifts the polygon to fit into the bounding box specified by x0, x1, y0 and y1. Make sure: x0 < x1 and y0 < y1!

Arguments:

- x0: float
- x1: float
- y0: float
- y1: float

Returns: None

p.flip(|x)

Flips polygon in x direction. If a value for x is not given, the center of the bounding box is used.

Arguments:

- optional x: float

Returns: None

p.flop(|y)

Flips polygon in y direction. If a value for y is not given, the center of the bounding box is used.

Arguments:

- optional y: float

Returns: None

p.cloneContour(i|, xs, ys)

Clones the contour i, returns index of clone, optionally shifts clone by xs and ys.

Arguments:

- i: integer
- optional xs: float
- optional ys: float

Returns: integer

p.triStrip()

Returns a list of tristrips describing the Polygon area. A tristrip is a list of triangles. The sum of all triangles fill the tristrip area. The triangles are usually not in a good shape for FEM methods!

Each strip stores triangle data by an memory-efficient method. A strip is a tuple containing points (2-tuples). The first three items of the tuple belong to the first triangle. The second, third and fourth item are the corners of the second triangle. Item number three, four and five are the corners of the third triangle, (...you may guess the rest!). The number of triangles in a strip is the number of points minus 2.

Arguments: None

Returns: list of tuples of 2-tuples

p.sample(rng)

Returns a random sample somewhere within the polygon.

Arguments:

- rng: Random number generator, a function or method taking 0 arguments, that returns a float [0.0..1.0] (e.g., python's random.random).

Returns: random point in the polygon as a 2-tuple

Module Polygon.Shapes

This module contains functions that create polygons in different shapes.

Circle(|radius=1.0, center=(0.0,0.0), points=32)

Create a polygonal approximation of a circle.

Arguments:

- optional radius: float
- optional center: point
- optional points: integer

Returns: new Polygon

Star(|radius=1.0, center=(0.0,0.0), beams=16, iradius=0.5)

Create a star shape, iradius is the inner and radius the outer radius.

Arguments:

- optional radius: float
- optional center: point
- optional beams: integer
- optional iradius: float

Returns: new Polygon

Rectangle(|xl= 1.0, yl=None)

Create a rectangular shape. If yl is not set, a square is created.

Arguments:

- optional xl: float
- optional yl: float

Returns: new Polygon

SierpinksiCarpet(|width= 1.0, level=5)

Create a sierpinski carpet.

DO NOT USE LEVELS > 6 UNLESS YOU KNOW WHAT YOU DO!

Arguments:

- optional width: float (1.0)
- optional level: int (5)

Returns: new Polygon

Module Polygon.IO

This module provides functions for reading and writing Polygons in different formats.

encodeBinary(p)

Encode Polygon p to a binary string. The binary string will be in a standard format with network byte order and should be rather machine independant. There's no redundancy in the string, any damage will make the hole polygon information unusable.

Arguments:

- p: Polygon

Returns: string

decodeBinary(s)

Create Polygon from a binary string created with encodeBinary(). If the string is not valid, the whole thing may break!

Arguments:

- s: string

Returns: new Polygon

The following write-methods will accept different argument types for the output. If ofile is None, the method will create and return a StringIO-object. If ofile is a string, a file with that name will be created. If ofile is a file, it will be used for writing.

The following read-methods will accept different argument types for the output. An file or StringIO object will be used directly. If the argument is a string, the function tries to read a file with that name. If it fails, it will evaluate the string directly.

writeGnuplot(ofile, polylist)

Write a list of Polygons to a gnuplot file, which may be plotted using the command `plot "ofile" with lines` from gnuplot.

Arguments:

- ofile: see above
- polylist: sequence of Polygons

Returns: ofile object

writeGnuplotTriangles(ofile, polylist)

Converts a list of Polygons to triangles and write the tringle data to a gnuplot file, which may be plotted using the command `plot "ofile" with lines` from gnuplot.

Arguments:
- ofile: see above
- polylist: sequence of Polygons

Returns: ofile object

writeSVG(ofile, polylist, ...)

Write a SVG representation of the Polygons in polylist, width and/or height will be adapted if not given. fill_color, fill_opacity, stroke_color and stroke_width can be sequences of the corresponding SVG style attributes to use. Optional labels can be placed at the polygons.

Arguments:
- ofile: see above
- polylist: sequence of Polygons
- optional width: float
- optional height: height
- optional fill_color: sequence of colors (3-tuples of integers [0,255]: RGB)
- optional fill_opacity: sequence of colors
- optional stroke_color: sequence of colors
- optional stroke_width: sequence of floats
- optional labels: sequence of strings (with same length as polylist)
- optional labels_coords: sequence of x,y coordinates
- optional labels_centered: if true, then the x,y coordinates specify the middle of the text's bounding box

Returns: ofile object

writeXML(ofile, polylist, withHeader=False)

Write a readable representation of the Polygons in polylist to a XML file. A simple header can be added to make the file parsable.

Arguments:
- ofile: see above
- polylist: sequence of Polygons
- optional withHeader: bool

Returns: ofile object

readXML(ifile)

Read a list of Polygons from a XML file which was written with writeXML().

Arguments:
- ofile: see above

Returns: list of Polygon objects

writePDF(ofile, polylist, pagesize=None, linewidth=0, fill_color=None)

This function is only available if the reportlab package is installed! Write a the Polygons in polylist to a PDF file.

Arguments:

- ofile: see above
- polylist: sequence of Polygons
- optional pagesize: 2-tuple of floats
- optional linewidth: float
- optional fill_color: color

Returns: ofile object

Module Polygon.Utils

This Module contains several utility functions.

fillHoles(p)

Returns the polygon p without any holes.

Arguments:

- p: Polygon

Returns: new Polygon

pointList(p)

Returns a list of all points of p.

Arguments:

- p: Polygon

Returns: list of points

convexHull(p)

Returns a polygon which is the convex hull of p.

Arguments:

- p: Polygon

Returns: new Polygon

tile(p, x=[], y=[] |, bb=None)

Returns a list of polygons which are tiles of p splitted at the border values specified in x and y. If you already know the bounding box of p, you may give it as argument bb (4-tuple) to speed up the calculation.

Arguments:

- p: Polygon
- x: list of floats
- y: list of floats
- optional bb: tuple of 4 floats

Returns: list of new Polygons

tileEqual(p, nx=1, ny=1 |, bb=None)

works like tile(), but splits into nx and ny parts.

Arguments:

- p: Polygon
- nx: integer
- ny: integer
- optional bb: tuple of 4 floats

Returns: list of new Polygons

warpToOrigin(p)

Shifts lower left corner of the bounding box to origin.

Arguments:

- p: Polygon

Returns: None

centerAroundOrigin(p)

Shifts the center of the bounding box to origin.

Arguments:

- p: Polygon

Returns: None

prunePoints(p)

Returns a new Polygon which has exactly the same shape as p, but unneeded points are removed. The new Polygon has no double points or points that are exactly on a straight line.

Arguments:

- p: Polygon

Returns: new Polygon

reducePoints(cont, n)

Remove points of the contour 'cont', return a new contour with 'n' points. *Very simple* approach to reduce the number of points of a contour. Each point gets an associated 'value of importance' which is the product of the lengths and absolute angle of the left and right vertex. The points are sorted by this value and the n most important points are returned. This method may give *very* bad results for some contours like symmetric figures. It may even produce self-intersecting contours which are not valid to process with this module.

Arguments:

- cont: list of points
- n: number of points to keep

Returns: new list of points

reducePointsDP(cont, tol)

Remove points of the contour 'cont' using the Douglas-Peucker algorithm. The value of tol sets the maximum allowed difference between the contours. This (slightly changed) code was written by Schuyler Erle and put into public domain. It uses an iterative approach that may need some time to complete, but will give better results than reducePoints().

Arguments:

- cont: list of points
- tol: allowed difference between original and new contour

Returns: new list of points

cloneGrid(poly, con, xl, yl, xstep, ystep)

Create a single new polygon with contours that are made from contour con from polygon poly arranged in a xl-yl-grid with spacing xstep and ystep.

Arguments:

- poly: Polygon
- con: integer
- xl: integer
- yl: integer
- xstep: float
- ystep: float

Returns: new Polygon

tileBSP(p)

This generator function returns tiles of a polygon. It will be much more efficient for larger polygons and a large number of tiles than the original tile() function. For a discussion see: http://dr-josiah.blogspot.com/2010/08/binary-space-partitions-and-you.html

Arguments:

- p: Polygon

Returns: tiles of the Polygon p on the integer grid

gpfInfo(fileName)

Get information on a gpc/gpf file.

Arguments:

- fileName: name of the file to read

Returns:

- contours: number of contours
- holes: number of holes (if contained)
- points: total number of points
- withHoles: file contains hole-flags