Polygon – a Python package for handling polygons (2D)

Documentation for Version 1.x

Contents

WARNING: The examples on the gpc homepage use an old file format and can't be read by the current version of gpc and Polygon without modifications!

Introduction

Polygon is a module to handle polygons in 2D space. The polygon storage and clipping functions as well as the tristrip conversion is based on gpc, the *General Polygon Clipping Library* by Alan Murta. Additional functionality is partially coded in C and in pure Python.

Author:	Joerg Raedler <joerg@dezentral.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:	http://www.dezentral.de/soft/Polygon/ The gpc package is located at: http://www.cs.man.ac.uk/aig/staff/alan/software/gpc.html

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:

"This library is provided free of charge for non-commercial users. Reaching its current state took up almost two years of personal development effort. If you have found this software to be useful for your purposes please make a donation and help support the continuation of this project. Thanks!"

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

Installation

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. Be sure to have write permissions in pythons site-packages directory!

This will install two modules on your system: cPolygon and Polygon.

Important Changes from 0.x to 1.x versions

- The style of point lists can be configured. Point lists can be returned as Lists, Tuples and Nuemeric.array.
- The object type TriStrip is dropped completely. Polygons now have a method called triStrip() instead. This method returns a list (or tuple) of triangle strips.
- Numeric is supported in several ways. If enabled during compilation there are two new features:
    - Point lists and Triangles can be returned as Numeric arrays if data style is set to STYLE_ARRAY
    - Polygon.addContour() will work faster with Numeric.arrays and other data types

Contents and Usage

The package currently contains two modules, a C extension module called cPolygon and a native Python module called Polygon.py.

Polygon.py imports all of cPolygon's functionality. The preferred usage is to import Polygon.py and use the functionality of both modules. But you may import cPolygon directly if you don't need the extensions in Polygon.py.

Most internal errors raise the `Error` exception, but in standard cases the standard exceptions IndexError, MemoryError and TypeError are used.

Basic Example

```
>>> from Polygon import *
>>> q = Polygon(((0.0, 0.0), (10.0, 0.0), (10.0, 5.0), (0.0, 5.0)))
>>> t = Polygon(((1.0, 1.0), (3.0, 1.0), (2.0, 3.0)))
>>> a = q - t  # gives a square with a triangular hole
```

Module cPolygon: core Polygon objects

cPolygon contains the following objects:

Symbol	*Description*
`Polygon`	Polygon type to hold a number of contours (a constructor)
`PolygonType`	Type object for Polygon
`setDataStyle`	set data style
`setTolerance`	*optional:* set tolerance for detection of coincident nodes
`getTolerance`	*optional:* get tolerance for detection of coincident nodes
`Error`	the exception raised when methods or operations fail
`withNumeric`	flag showing if the support for Numeric is enabled
`withNumarray`	for future use – not yet implemented!
`STYLE_TUPLE,` `STYLE_LIST,` `STYLE_ARRAY`	data style constants to be used with setDataStyle
`version, __version__,` `author, __author__,` `license, __license__`	variables to hold meta information on the package

Polygon objects

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 Numeric 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.

Polygon methods

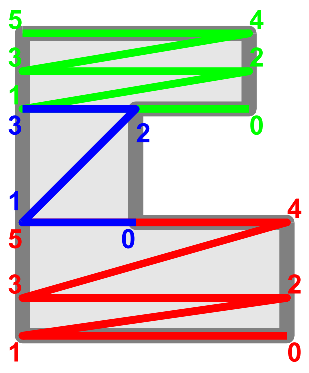
Method	*Arguments*	*Returns*	*Description*
`Polygon(|arg)`	optional one of : - a filename string, - a readable file object - a pointlist (sequence of 2-sequences) - a polygon object	Polygon object	Constructor: Create a new Polygon object. Files must contain data in gpc polygon format. If arg is another polygon, a copy is returned.
`addContour(c |, hole=0)`	- pointlist - and an optional hole flag	None	Add a contour (outline or hole).
`contour(i)`	- contour index	a contour as tuple of 2-tuples	gives the contour with index i
`isHole(|i)`	- optional contour index	an integer (0/1) or a tuple of integers	Returns the hole flag of a single contour (when called with index argument) or a list of all flags when called without arguments.
`isSolid(|i)`	- optional contour index	an integer (0/1) or a tuple of integers	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.
`nPoints(|i)`	- optional contour index	an integer	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))`!
`read(file)`	- a readable file object - or a filename string	None	Reads Polygon data from a file in gpc format.
`write(file)`	- a writable file object - or a filename string	None	Writes Polygon data to a file in gpc format.
`simplify()`	None	None	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.
`area(|i)`	- optional contour index	a float	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.
`center(|i)`	- optional contour index	a 2-tuple containing x and y float values	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!
`orientation(|i)`	- optional contour index	single integer or list of integers: 1 for ccw, -1 for cw, 0 for invalid contour.	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!
`isInside(x, y |, i)`	- two float values: x and y - and an optional contour index	1 for inside , 0 for outside (or inside a hole)	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 0 or 1, sorry!
`covers(p)`	- a polygon object	integer (0/1)	Tests if the polygon completely covers the other polygon p. At first bounding boxes are tested for obvious cases and then an optional clipping is performed.
`overlaps(p)`	- a polygon object	integer(0/1)	Tests if the polygon overlaps the other polygon p. At first bounding boxes are tested for obvious cases and then an optional clipping is performed.

Method	***Arguments***	***Returns***	***Description***
`boundingBox(|i)`	- optional contour index	tuple of four floats: xmin, xmax, ymin and ymax	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.
`aspectRatio(|i)`	- optional contour index	float	Returns the aspect ratio Δy / Δx of the bounding box of one contour (when called with index) or of the whole polygon.
`scale(xs, ys |, xc, yc)`	- two scaling values for x and y direction - and optional two floats for the center point	None	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.
`shift(xs, ys)`	- two shift values for x and y	None	Shifts the polygon by adding xs and ys.
`rotate(a |, xc, yc)`	- an angle (float) - and optional two floats for the center point	None	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.
`warpToBox(x0, x1, y0, y1)`	- four floats to specify the corners of the new bounding box	None	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
`flip(|x)`	- optional x value	None	Flips polygon in x direction. If a value for x is not given, the center of the bounding box is used.
`flop(|y)`	-optional y value	None	Flips polygon in y direction. If a value for y is not given, the center of the bounding box is used.
`triStrip()`		TriStrip (see below)	returns a TriStrip describing the Polygon area

Polygon operations

Operation (a and b are Polygon objects)	***Result***
`len(a)`	number of contours
`a[i]`	contour with index `i` as tuple of 2-tuples, exactly the same as `a.contour(i)`, slicing is not yet supported
`if a: ...`	logical value is true, if there are any contours in a (contours may be empty!)
`a + b` or `a | b`	a polygon containing the area that is covered by `a` or `b` or both
`a - b`	a polygon with the area of `a` that is not covered by `b`
`a & b`	a polygon with the area that is covered by both `a` and `b`
`a ^ b`	a polygon with the area that is covered by exactly one of `a` and `b`

TriStrips



A TriStrip is a list of triangles. The sum of all triangles fill the TriStrip area. The triangles usually are not in a good shape for FEM methods!

The object returned by the triStrip()-method represents a list of strips. Each strip stores triangle data by an 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 example on the right shows an area which is described by three strips.

The number of triangles in a strip is the number of points lowered by 2.

getTolerance() and setTolerance()

Both functions are only available if you compiled cPolygon with adjustable Tolerance.

With the tolerance gpc tests if two points are the same. Default value is `DBL_EPSILON`. You may read and adjust the tolerance with the above functions.

setDataStyle()

By calling this functions with `STYLE_TUPLE`, `STYLE_LIST` or `STYLE_NUMERIC` as the argument you influence the style of data returned by methods.

Module Polygon.py : additional methods

These methods are partially work-in-progress and used for testing. Some may be recoded as bounded methods and moved into cPolygon in future releases.

- `polyStar(| radius=1.0, center=(0.0, 0.0), nodes=38, iradius=None)`
  returns a polygon which may be a star, a triangle, an approximated circle or other shapes. If you specify iradius, the number of nodes is doubled and every second node gets the radius of iradius. All arguments are keyworded.

- `fillHoles(poly)`
  returns a copy of the polygon poly without the holes.

- `pointList(poly |, withHoles=1)`
  returns a flat list of all points in polygon poly. If withHoles is false, holes are excluded.

- `convexHull(poly)`
  returns a new polygon which is the convex hull of poly.

- `tile(poly, x=[], y=[] |, bb=None)`
  returns a list of polygons which are tiles of poly splitted at the border values specified in x and y. If you already know the bounding box of poly, you may give it as argument bb (4-tuple) to speed up the calculation.

- `tileEqual(poly, nx=1, ny=1 |, bb=None)`
  works like tile(), but splits into nx and ny parts.

- `isCoveredBy(poly, poly1)`
  returns if poly is completely covered by poly1, which is the simple expression '`not poly-poly1`'.

- `gnuplotPolygon(filename, poly0 |, poly1 ...)`
  writes a file to be viewed with gnuplot. You may use the gnuplot command
  `plot "filename" with linespoints`
  to get a raw idea of how the contours of the polygon(s) look.

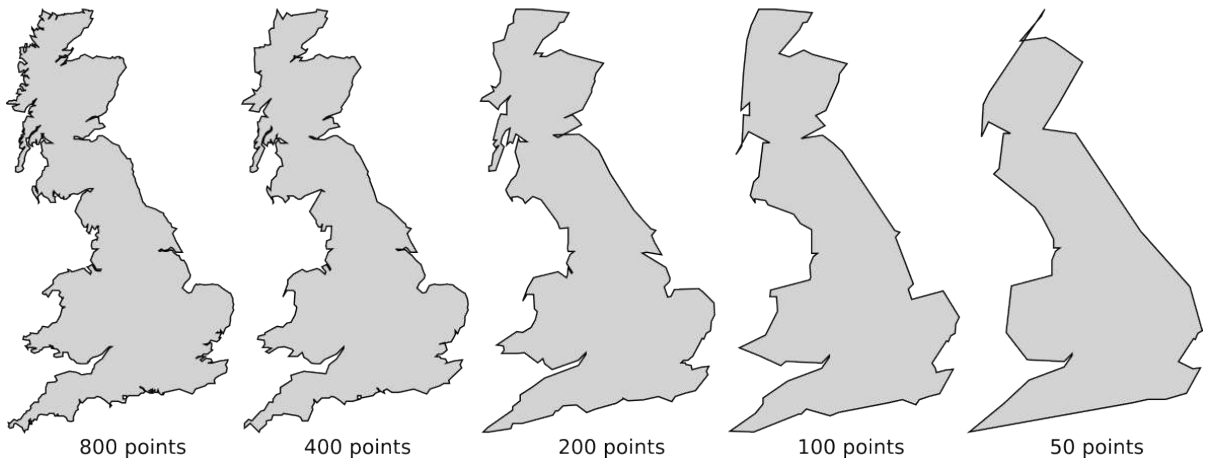
- `gnuplotTriStrip(filename, tri0 |, tri1 ...)`
  works like gnuplotPolygon() but writes tristrips.

- `drawSVG (polygons |, width=None, height=None, ofile=None, fill_color=None, fill_opacity=None, stroke_color=None, stroke_width=None)`
  return a SVG representation of the polygons, width and/or height will be adapted if not given. If `ofile` is None, a string is returned, else the contents will be written to the file. `ofile` may be an object with a `write()` method or a filename. `fill_color`, `fill_opacity`, `stroke_color` and `stroke_width` can be sequences of the corresponding SVG style attributes to use. Every polygon is written as a SVG path element. To edit the SVG files I recommend the applications Inkscape or Sodipodi. If you don't have a viewer you may use Adobe's SVG Plugin for your internet browser.

- `prunePoints(poly)`
  returns a new polygon which has exactly the same shape as poly, but unneeded points are removed. Unneeded means double points and points which are exactly on a straight line between the two neighbor points.

- `reducePoints(con, n)`

reduces the number of points of the contour con (not a polygon object!) and returns a new list of points, while trying to keep the “most important points”. **This method uses a very simple approach, which may give very bad results for some sort of contours!** It may even return self-intersecting contours which are not valid for further calculations with the Polygon package or other software. The next picture gives an idea of the results. It uses contour data derived from an example on the gpc homepage.


800 points
400 points
200 points
100 points
50 points