This project is a small Python library, `geometry.py`, to use in drawing figures for my geometry textbook. The library is contained in a single file at top level: `geometry.py`.

I am running Python 3.13 obtained via Homebrew, so I put a symbolic link to geometry.py in:

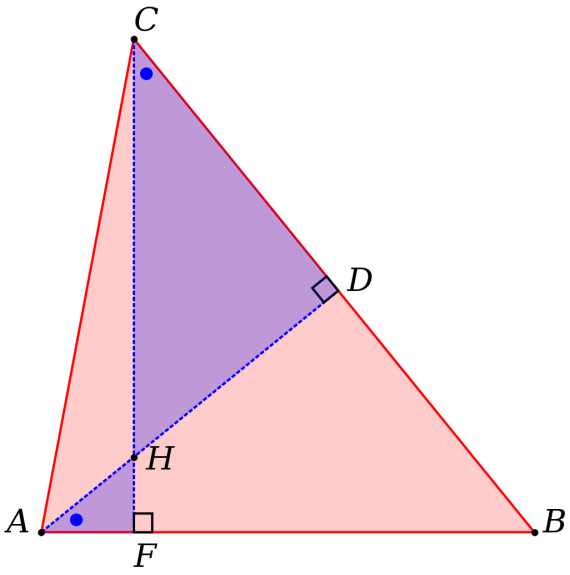
```
/usr/local/lib/python3.13/site-packages/geometry.py
```

The `projects` directory contains sub-directories with various scripts using it. Other scripts written during development to test particular issues are in `tests`.

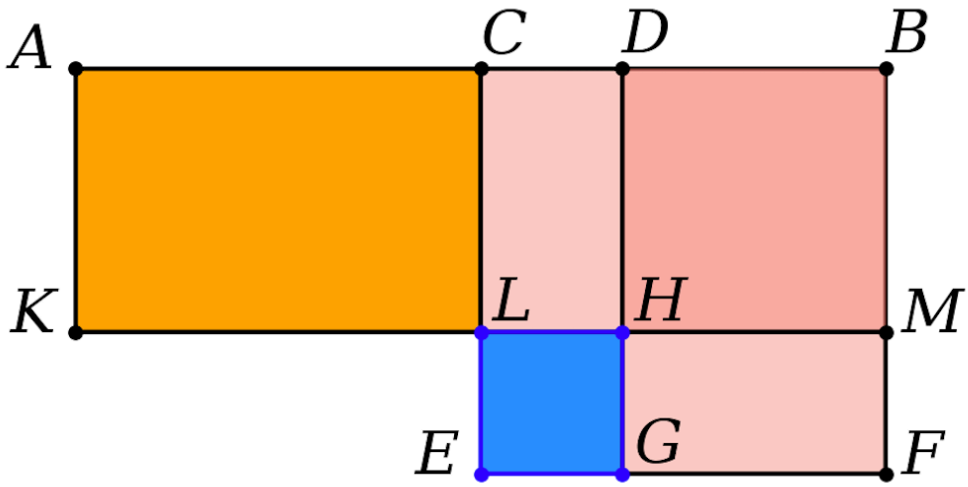
Here is a list of all the functions defined there.

Some examples of figures made using the library:

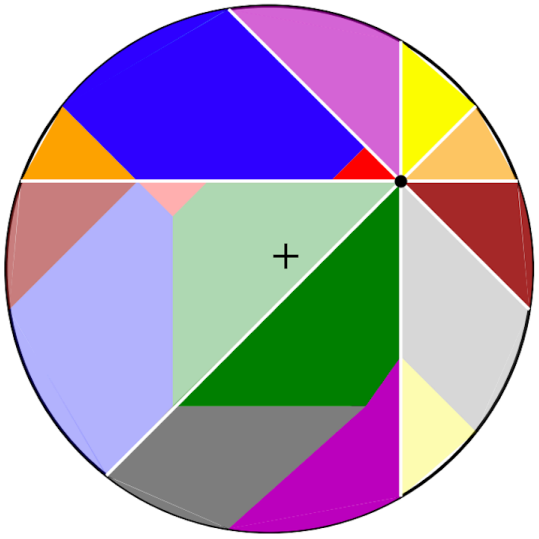
basic demo



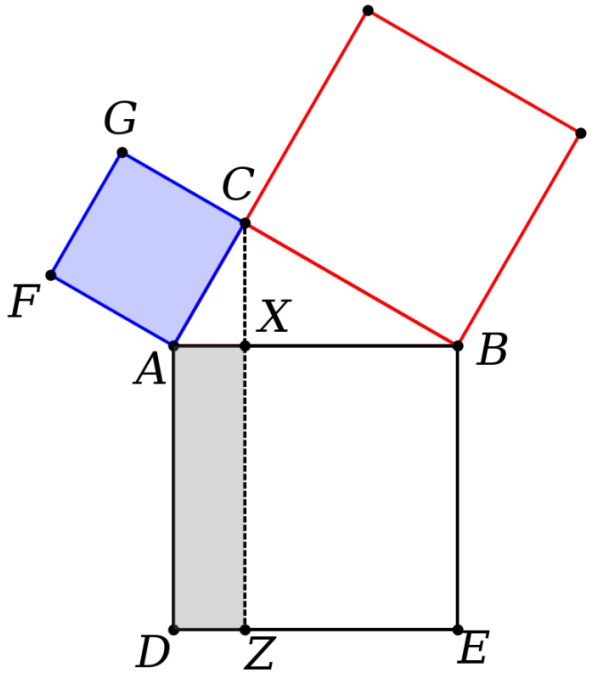
Euclid II.5



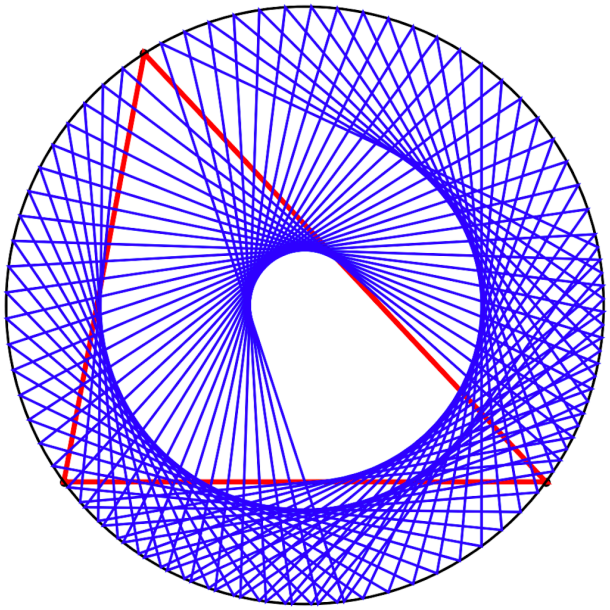
Pizza theorem



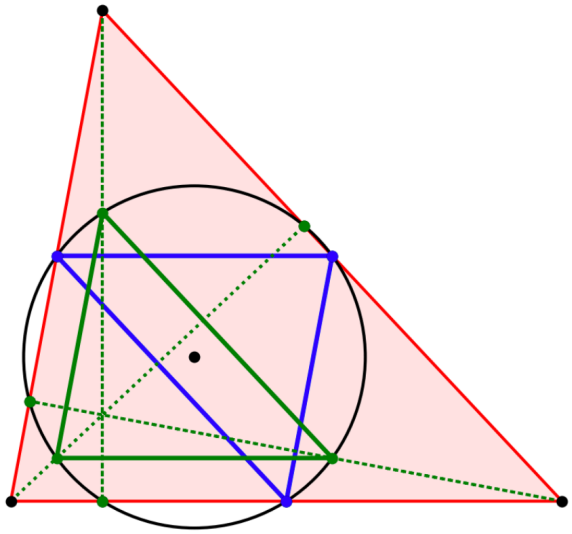
Euclid I.47



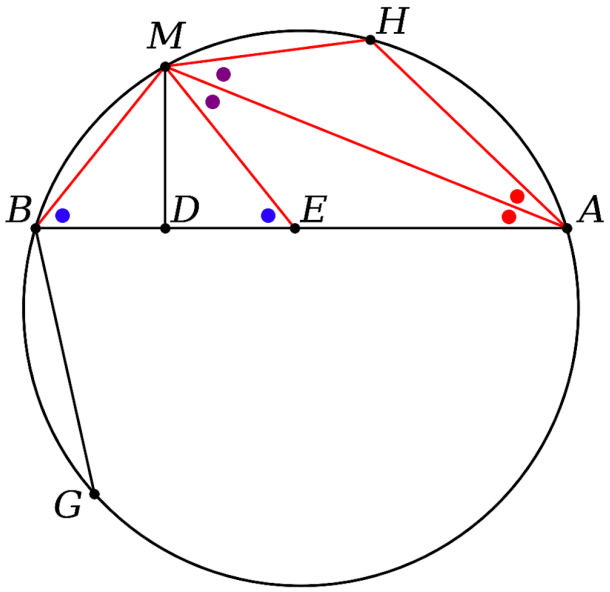
triangle rotation



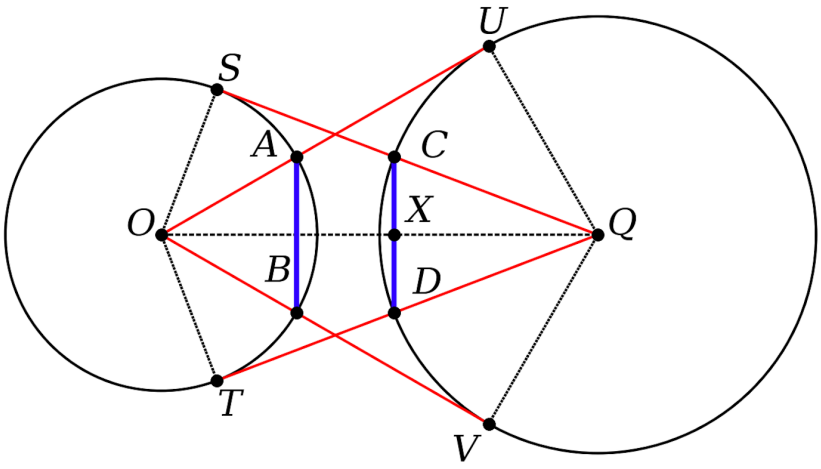
nine point circle



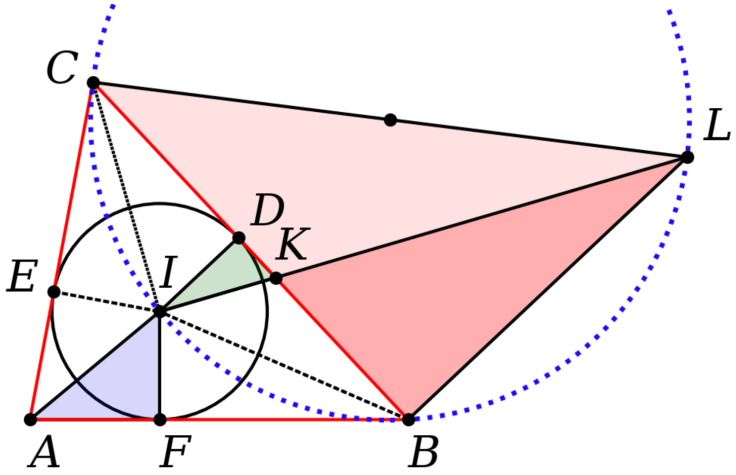
broken chord proof 1



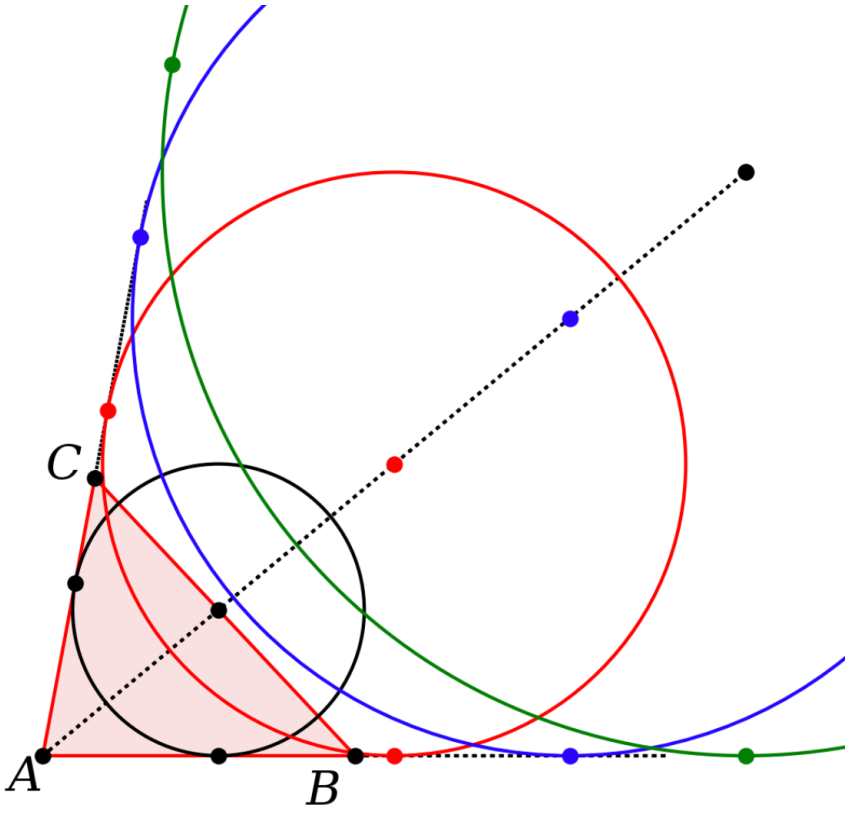
eyeball theorem



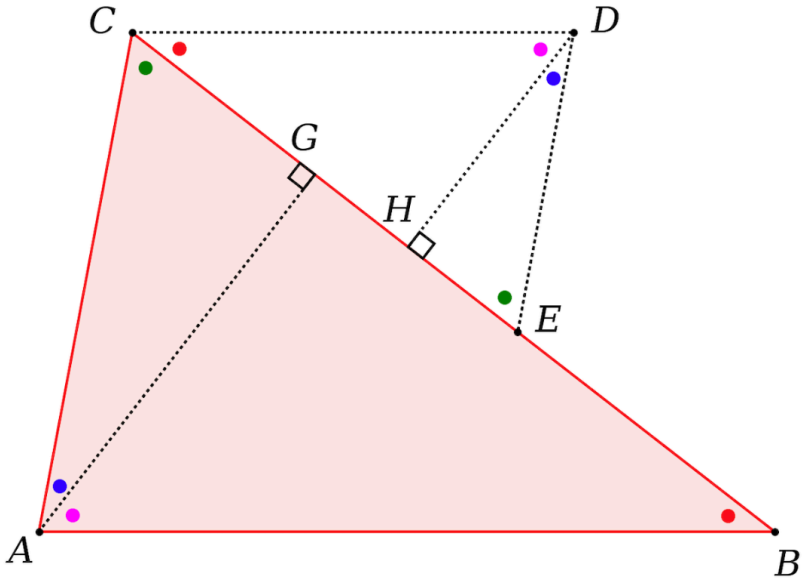
Heron's theorem



excircle



similar triangles



There are also a couple of write-ups, including one about Archimedes' broken chord theorem and another about excircles.

This is functional programming. The only objects we define are members of the class **Point**, to allow access via P.x and P.y.

The variable name **pL** found in most function definitions stands for *point list*, i.e. a list of Point objects. This may be a line segment, a triangle or another polygon.

We pretend to implement some of Euclid's constructions, but intersections between lines and circles are computed by analytic geometry. Under the hood, it is algebra.

When there are two points in a result to be returned, the order in which they are returned is sometimes challenging to determine. In the latest version, for two points, say, perpendicular to a line segment, we return the point "above" the line segment first, if you visualize the line segment as oriented left-to-right.

Some examples may differ. For circle-circle intersection, we return the point closer to the origin first.

For a perpendicular or angle bisector, the *length* of the perpendicular or bisector is arbitrary. It should be adjusted by the caller, using the following trick:

```
S,T = geo.get_perp_at_point_by_fractional_length([A,B],f=0.5)
X = geo.get_intersection_for_two_lines([A,B],[S,T])

d = 10   # or whatever the desired length is
f = d/geo.get_length([X,S])
geo.get_point_by_fractional_length([X,S],f)
```

As I fiddled with the code, inconsistency in the order of return of two points has messed up many a diagram. I believe that's all fixed now.

For most examples, output paths for figures are hard-coded so it will require a bit of configuration to get it to work on another machine. I have tried to automate that for newer code, but haven't fixed the old examples.

Here are a few of the functions we can call:

```
geo.get_intersection_for_two_lines([A,B],[C,D])
geo.get_point_perp_on_line_for_point(P,[A,B])
geo.get_perp_at_point_by_fractional_length([A,B],f=0.5)

get_intersection_line_segment_circle([A,B],[Q,r])
get_intersection_circle_circle([Q1,r1],[Q2,r2])
get_tangent_points_on_circle_for_point([Q,r],P)
```

These are from the caller's POV. In the library's function definition, you cannot have

`([A,B],[C,D])`, it is

```
geo.get_intersection_for_two_lines(pL1,pL2)
```

`matplotlib` gives error messages that can be challenging to interpret. If you forget an argument in a function call, it simply reports the *last* one missing. So, in drawing functions like

```
geo.outline_polygon(ax,[A,B,E,D],ec='k')
geo.draw_line_segments(ax,[[D,F],[C,F]])
```

If you forget `ax` in the first one, the error is:

```
TypeError: outline_polygon() missing 1 required
positional argument: 'pL'
```

If you forget to make a list of line segments by adding a second pair of brackets in the second one:

```
TypeError: draw_line_segments() got multiple values
for argument 'ec'
```

Other mistakes with brackets may result in Python trying to access a coordinate like `P.x` and complaining that a list doesn't have one.

```
AttributeError: 'tuple' object has no attribute 'x'
```

Finally, the functions in the library have long, but I hope explicit, names. I'm experimenting with shortcut definitions (here).

The shortcuts live in the library (at the end).

For this we might do `from geometry import *` in spite of the fact that it's generally not good practice. Or, one might call them like

```
geo.gtr()
```