HumanGeo
Blog (//blog.thehumangeo.com)

Drawing Boundaries In Python

MAY 12, 2014 ▪ KEVIN DWYER GEOSPATIAL (/CATEGORIES/#GEOSPATIAL) OPEN-SOURCE (/CATEGORIES/#OPEN-SOURCE) PYTHON (/CATEGORIES/#PYTHON) DATA-SCIENCE (/CATEGORIES/#DATA-SCIENCE)

As a technologist at HumanGeo, you're often asked to perform some kind of analysis on geospatial data, and quickly! We frequently work on short turnaround times for our customers so anything that gives us a boost is welcome, which is probably why so many of us love Python. As evidenced by the volume of scientific talks (https://us.pycon.org/2014/schedule/talks/list/) at PyCon 2014, we can also lean on the great work of the scientific community. Python lets us go from zero to answer within hours or days, not weeks.

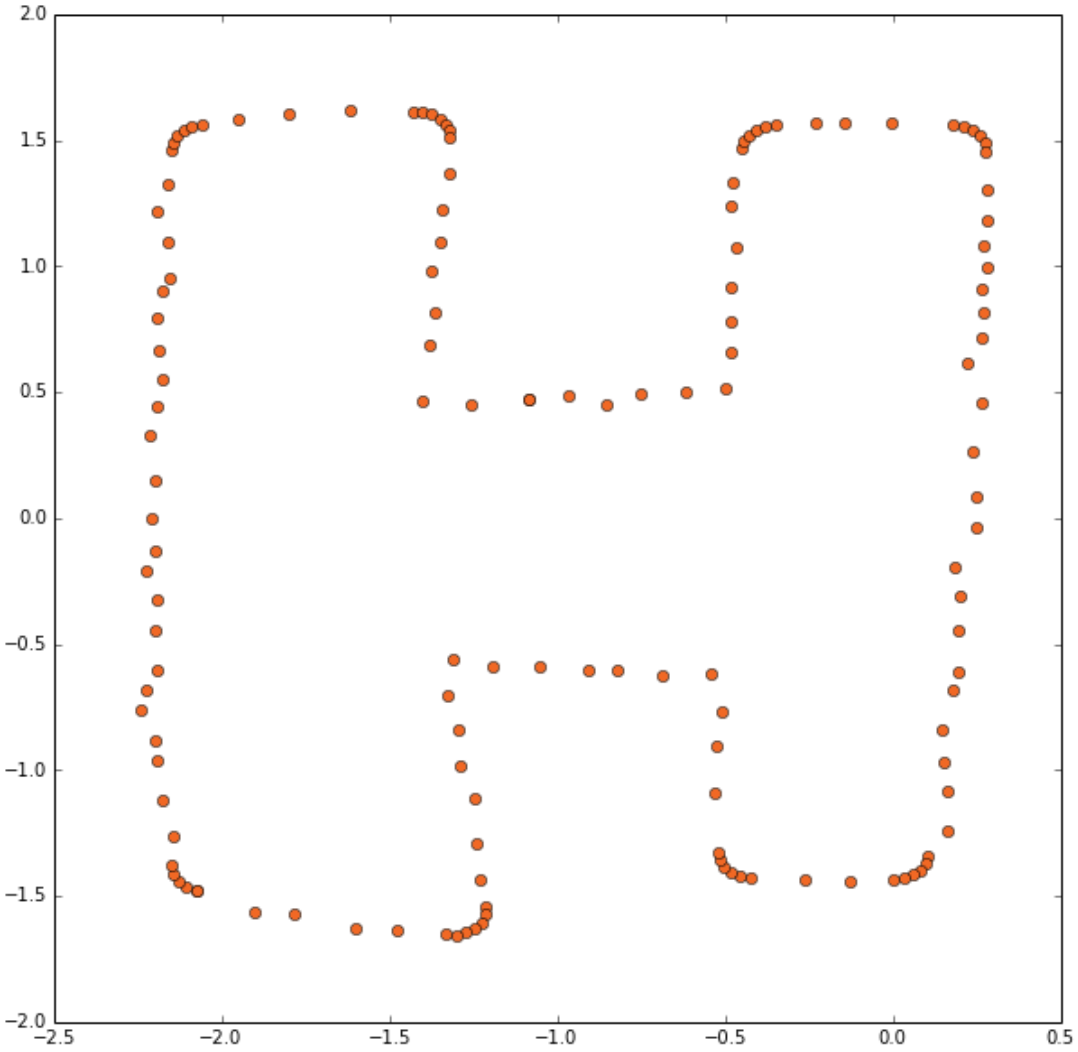
I recently had to do some science on the way we can observe clusters of points on the map - to show how regions of social significance emerge. Luckily I was able to lean heavily on Shapely (http://toblerity.org/shapely/) which is a fantastic Python library for performing geometric operations on points, shapes, lines, etc. As an aside, if you are doing any sort of geospatial work with Python, you'll want to `pip install shapely` . Once we found a cluster of points which we believed were identifying a unique region, we needed to draw a boundary around the region so that it could be more easily digested by a geospatial analyst. Boundaries are just polygons that enclose something, so I'll walk through some of your options and attempt to provide complete code examples.

The first step towards geospatial analysis in Python is loading your data. In the example below, I have a shapefile containing a number of points which I generated manually with QGIS. I'll use the fiona (http://toblerity.org/fiona/) library to read the file in, and then create point objects with shapely.

```
import fiona
import shapely.geometry as geometry
input_shapefile = 'concave_demo_points.shp'
shapefile = fiona.open(input_shapefile)
points = [geometry.shape(point['geometry'])
          for point in shapefile]
```

The points list can now be manipulated with Shapely. First, let's plot the points to see what we have.

```
import pylab as pl
x = [p.coords.xy[0] for p in points]
y = [p.coords.xy[1] for p in points]
pl.figure(figsize=(10,10))
_ = pl.plot(x,y,'o', color='#f16824')
```

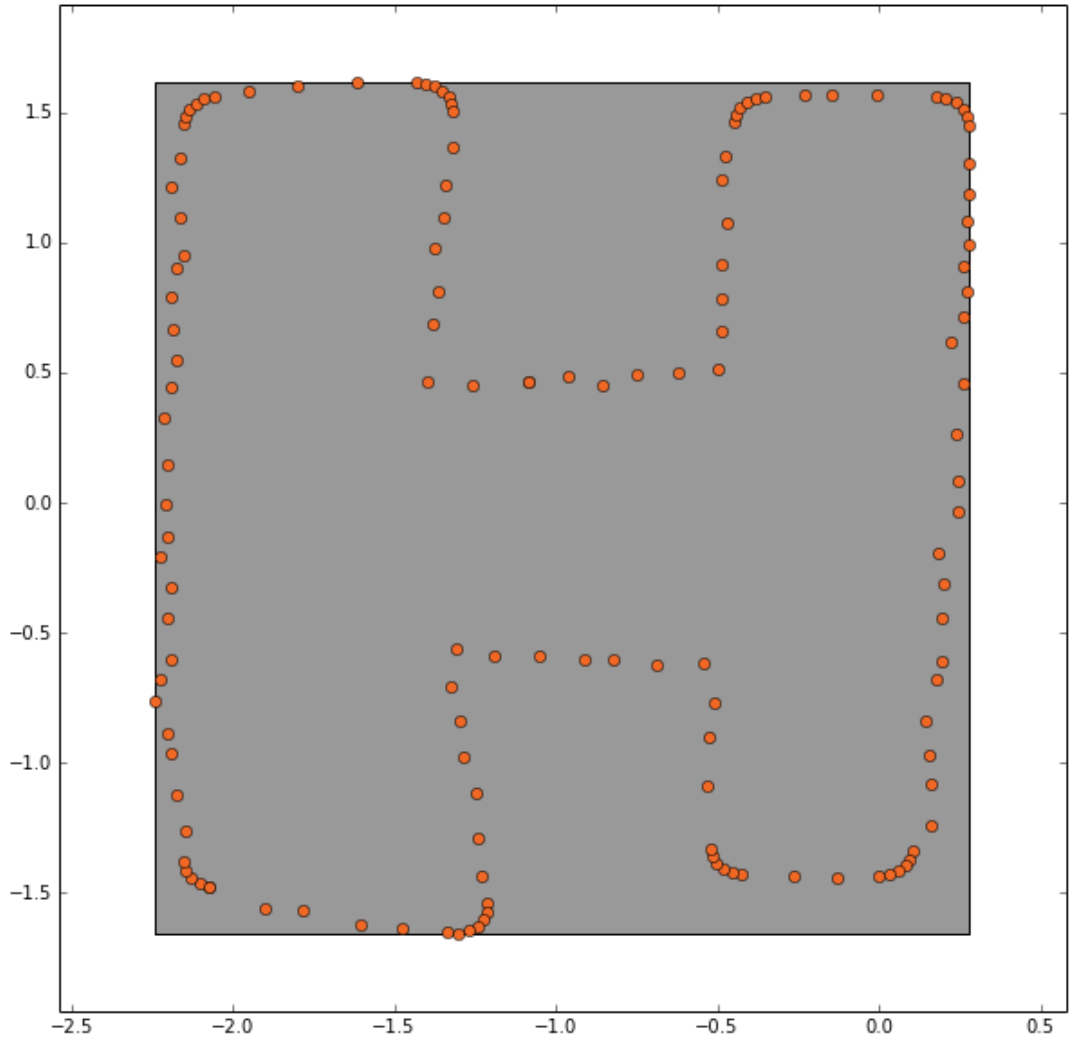


We can now interrogate the collection. Many shapely operations result in a different kind of geometry than the one you're currently working with. Since our geometry is a collection of points, I can instantiate a MultiPoint, and then ask that MultiPoint for its envelope, which is a Polygon. Easily done like so:

```
point_collection = geometry.MultiPoint(list(points))
point_collection.envelope
```

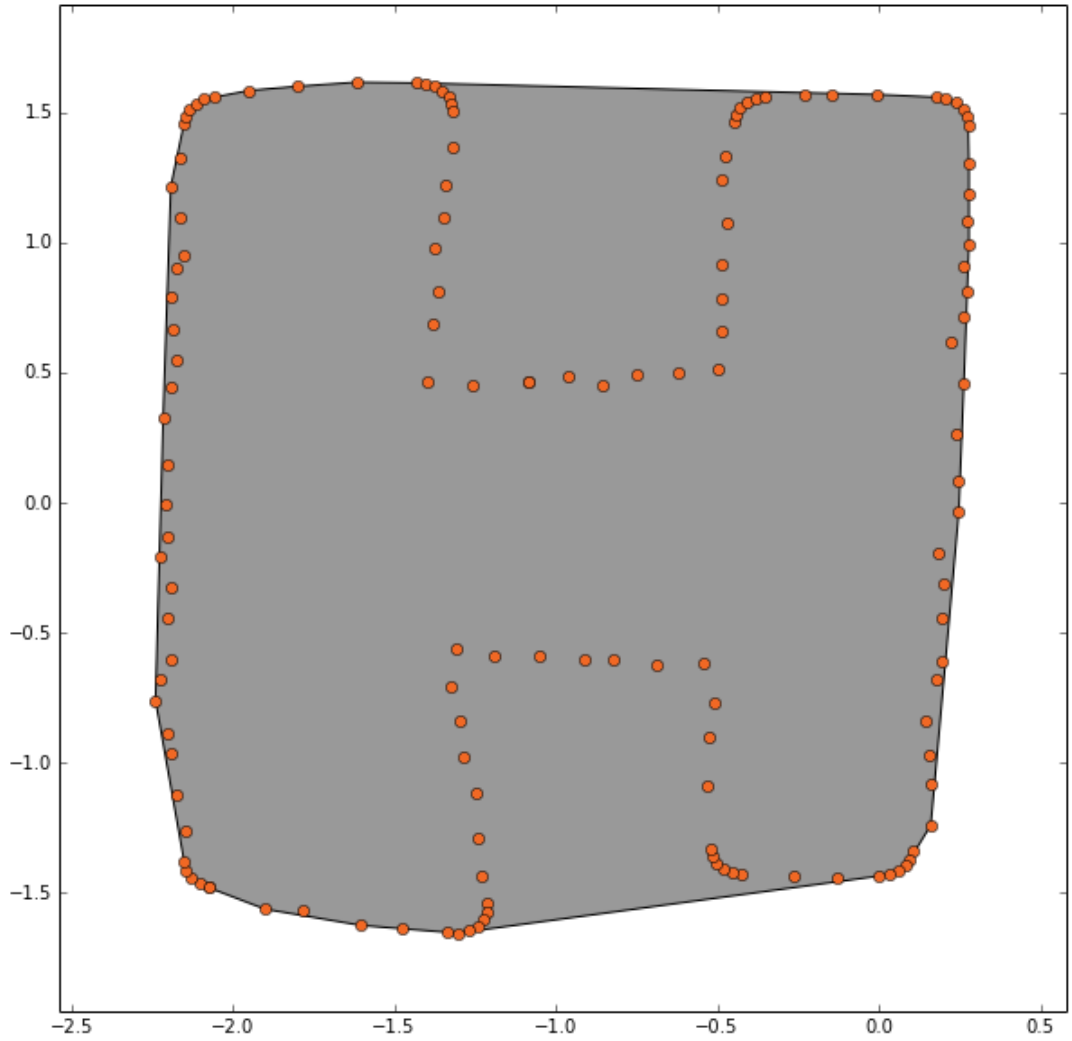
We should take a look at that envelope. matplotlib can help us out, but polygons aren't functions, so we need to use PolygonPatch.

```
from descartes import PolygonPatch
def plot_polygon(polygon):
    fig = pl.figure(figsize=(10,10))
    ax = fig.add_subplot(111)
    margin = .3
    x_min, y_min, x_max, y_max = polygon.bounds
    ax.set_xlim([x_min-margin, x_max+margin])
    ax.set_ylim([y_min-margin, y_max+margin])
    patch = PolygonPatch(polygon, fc='#999999',
                         ec='#000000', fill=True,
                         zorder=-1)
    ax.add_patch(patch)
    return fig

_ = plot_polygon(point_collection.envelope)
_ = pl.plot(x,y,'o', color='#f16824')
```



So without a whole lot of code, we were able to get the envelope of the points, which is the smallest rectangle that contains all of the points. In the real world, boundaries are rarely so uniform and straight, so we were naturally led to experiment with the convex hull of the points. Convex hulls are polygons drawn around points too - as if you took a pencil and connected the dots on the outer-most points. Shapely has convex hull as a built in function so let's try that out on our points.

```
convex_hull_polygon = point_collection.convex_hull
_ = plot_polygon(convex_hull_polygon)
_ = pl.plot(x,y,'o', color='#f16824')
```

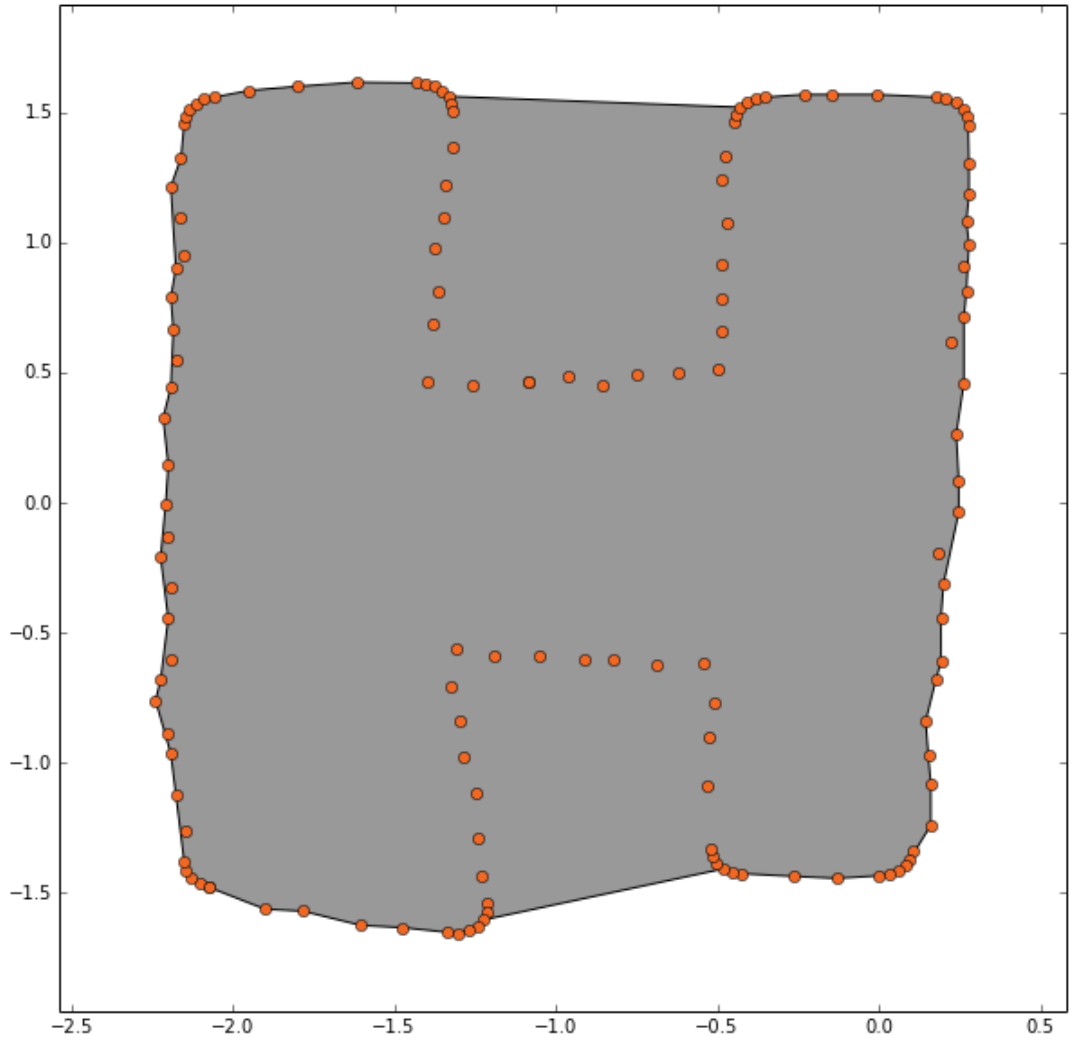


A tighter boundary, but it ignores those places in the "H" where the points dip inward. For many applications, this is probably good enough but we wanted to explore one more option which is known as a *concave* hull or alpha shape. At this point we've left the built-in functions of Shapely and we'll have to write some more code. Thankfully, smart people like Sean Gillies, the author of Shapely and fiona, have done the heavy lifting. His post on the fading shape of alpha (http://sgillies.net/blog/1155/the-fading-shape-of-alpha/) gave me a great place to start. I had to fill in some gaps that Sean left so I'll recreate the entire working function here.

```
from shapely.ops import cascaded_union, polygonize
from scipy.spatial import Delaunay
import numpy as np
import math
def alpha_shape(points, alpha):
    """
    Compute the alpha shape (concave hull) of a set
    of points.
    @param points: Iterable container of points.
    @param alpha: alpha value to influence the
        gooeyness of the border. Smaller numbers
        don't fall inward as much as larger numbers.
        Too large, and you lose everything!
    """
    if len(points) < 4:
        # When you have a triangle, there is no sense
        # in computing an alpha shape.
        return geometry.MultiPoint(list(points))
               .convex_hull
    def add_edge(edges, edge_points, coords, i, j):
        """
        Add a line between the i-th and j-th points,
        if not in the list already
        """
            if (i, j) in edges or (j, i) in edges:
                # already added
                return
            edges.add( (i, j) )
            edge_points.append(coords[ [i, j] ])
    coords = np.array([point.coords[0]
                       for point in points])
    tri = Delaunay(coords)
    edges = set()
    edge_points = []
    # loop over triangles:
    # ia, ib, ic = indices of corner points of the
    # triangle
    for ia, ib, ic in tri.vertices:
        pa = coords[ia]
        pb = coords[ib]
        pc = coords[ic]
        # Lengths of sides of triangle
        a = math.sqrt((pa[0]-pb[0])**2 + (pa[1]-pb[1])**2)
        b = math.sqrt((pb[0]-pc[0])**2 + (pb[1]-pc[1])**2)
```

```
        c = math.sqrt((pc[0]-pa[0])**2 + (pc[1]-pa[1])**2)
        # Semiperimeter of triangle
        s = (a + b + c)/2.0
        # Area of triangle by Heron's formula
        area = math.sqrt(s*(s-a)*(s-b)*(s-c))
        circum_r = a*b*c/(4.0*area)
        # Here's the radius filter.
        #print circum_r
        if circum_r < 1.0/alpha:
            add_edge(edges, edge_points, coords, ia, ib)
            add_edge(edges, edge_points, coords, ib, ic)
            add_edge(edges, edge_points, coords, ic, ia)
    m = geometry.MultiLineString(edge_points)
    triangles = list(polygonize(m))
    return cascaded_union(triangles), edge_points
concave_hull, edge_points = alpha_shape(points,
                                        alpha=1.87)
_ = plot_polygon(concave_hull)
_ = pl.plot(x,y,'o', color='#f16824')
```

That's a mouthful, but the gist is that we are going to compute Delaunay triangles which establish a connection between each point and nearby points and then we remove some of the triangles that are too far from their neighbors. This removal part is key. By identifying candidates for removal we are saying that these points are too far from their connected points so don't use that connection as part of the boundary. The result looks like this.

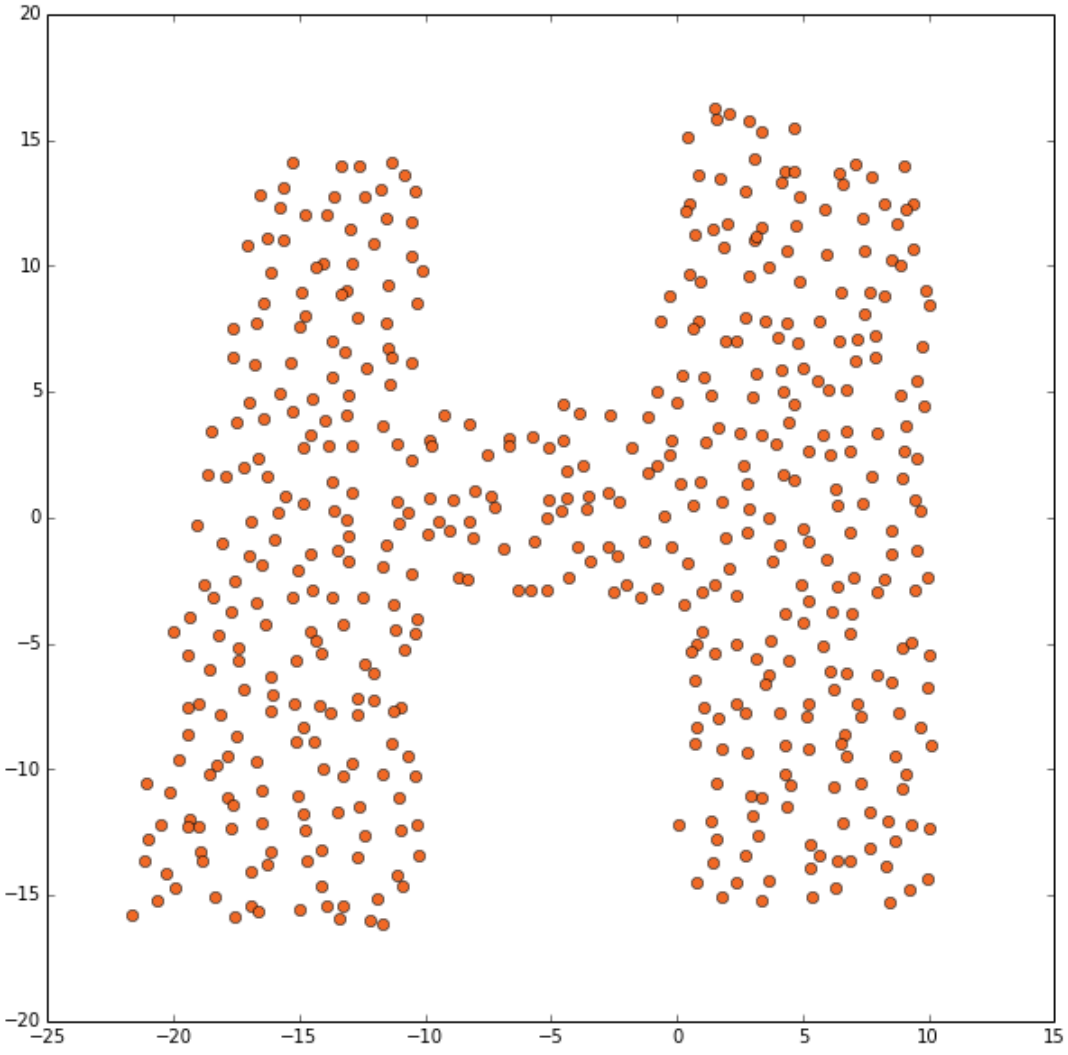


Better, but not great.

It turns out that the alpha value and the scale of the points matters a lot when it comes to how well the Delaunay triangulation method will work. You can usually play with the alpha value to find a suitable response, but unless you can scale up your points it might not help. For the sake of a good example, I'll do both: scale up the "H" and try some different alpha values.

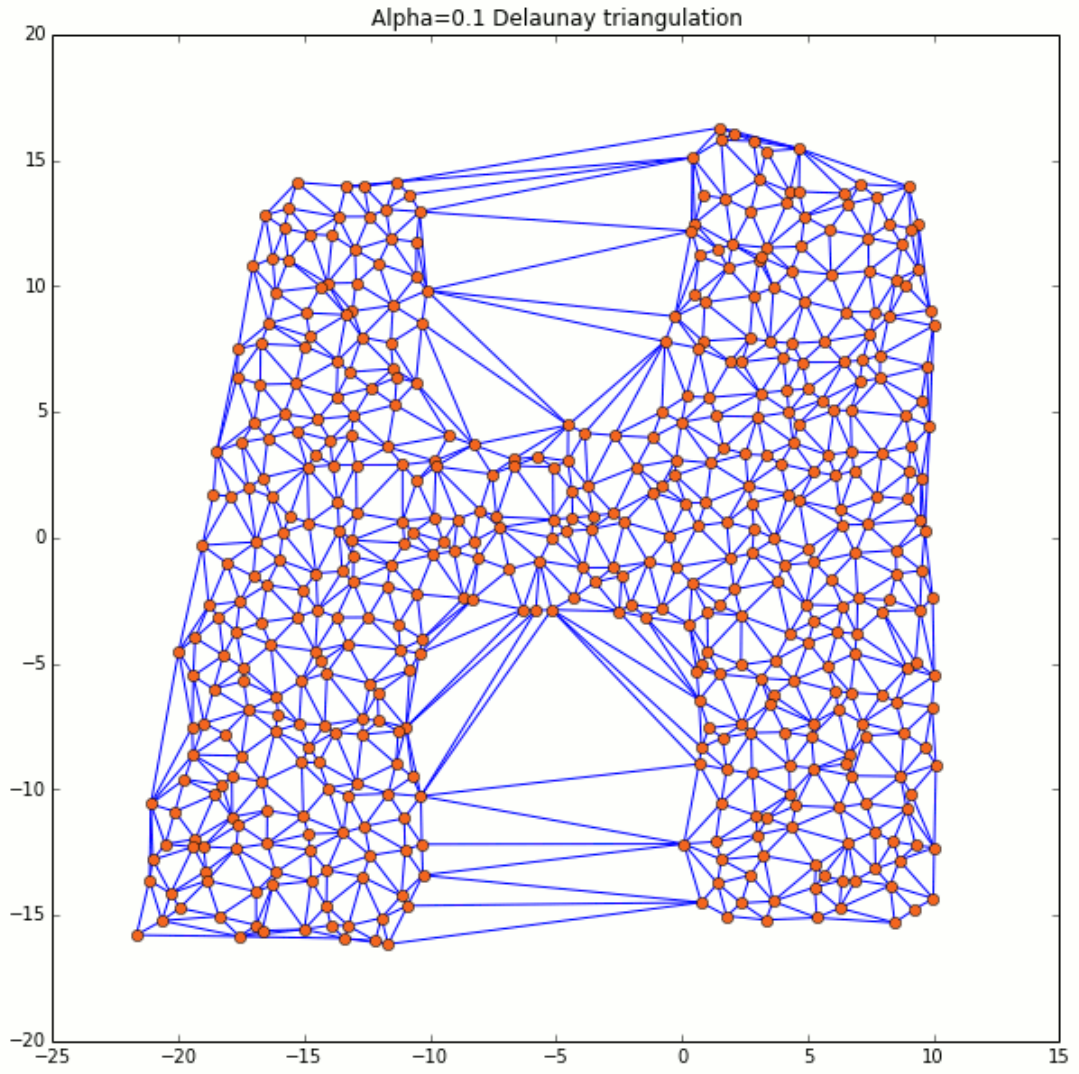
To get more points, I opened up QGIS, drew an "H" like polygon, used the tool to generate regular points, and then spatially joined them to remove any points outside the "H". My new dataset looks like this:

```
input_shapefile = 'demo_poly_scaled_points_join.shp'
new_shapefile = fiona.open(input_shapefile)
new_points = [geometry.shape(point['geometry'])
              for point in new_shapefile]
x = [p.coords.xy[0] for p in new_points]
y = [p.coords.xy[1] for p in new_points]
pl.figure(figsize=(10,10))
_ = pl.plot(x,y,'o', color='#f16824')
```



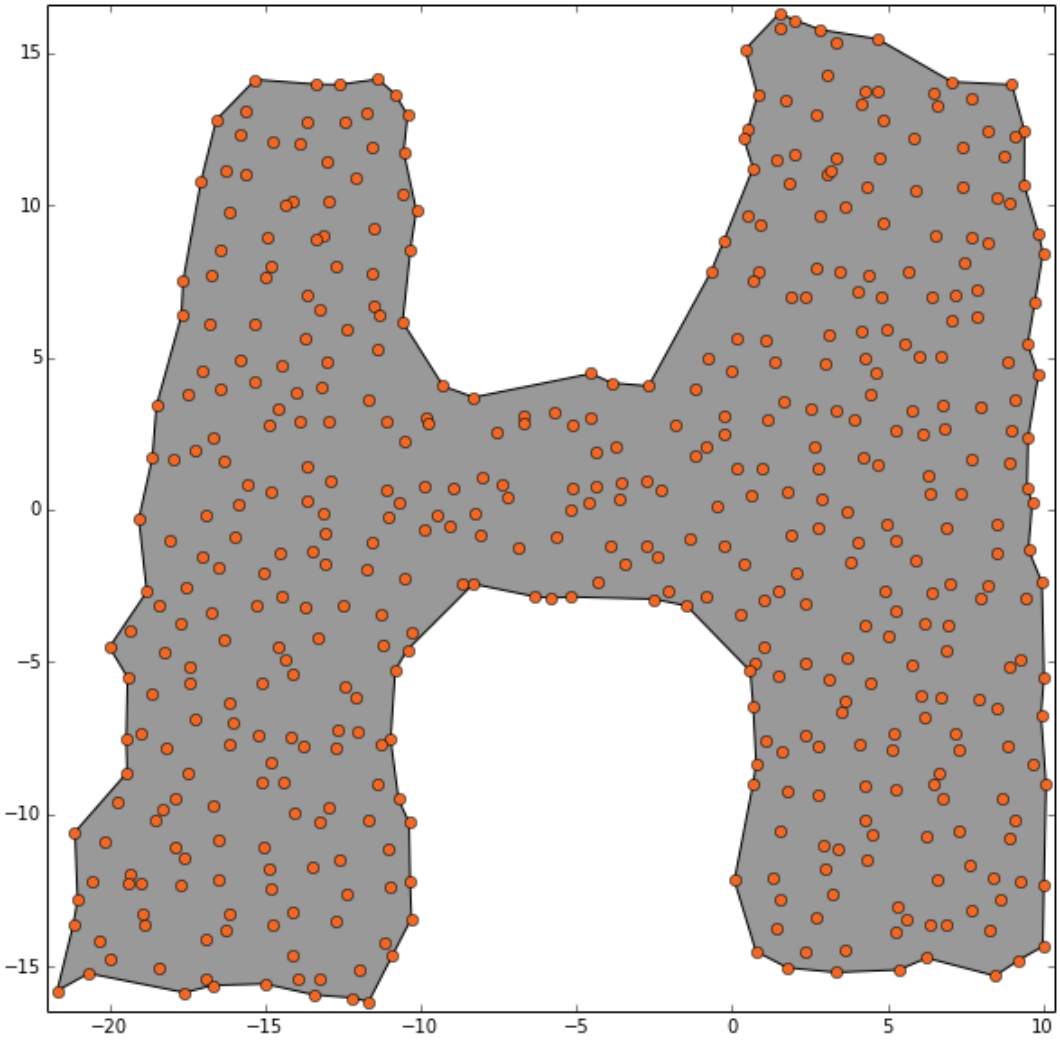
When we try the alpha shape transformation on these points we get a much more satisfying boundary. We can try a few permutations to find the best alpha value for these points with the following code. I combined each plot into an animated gif below.

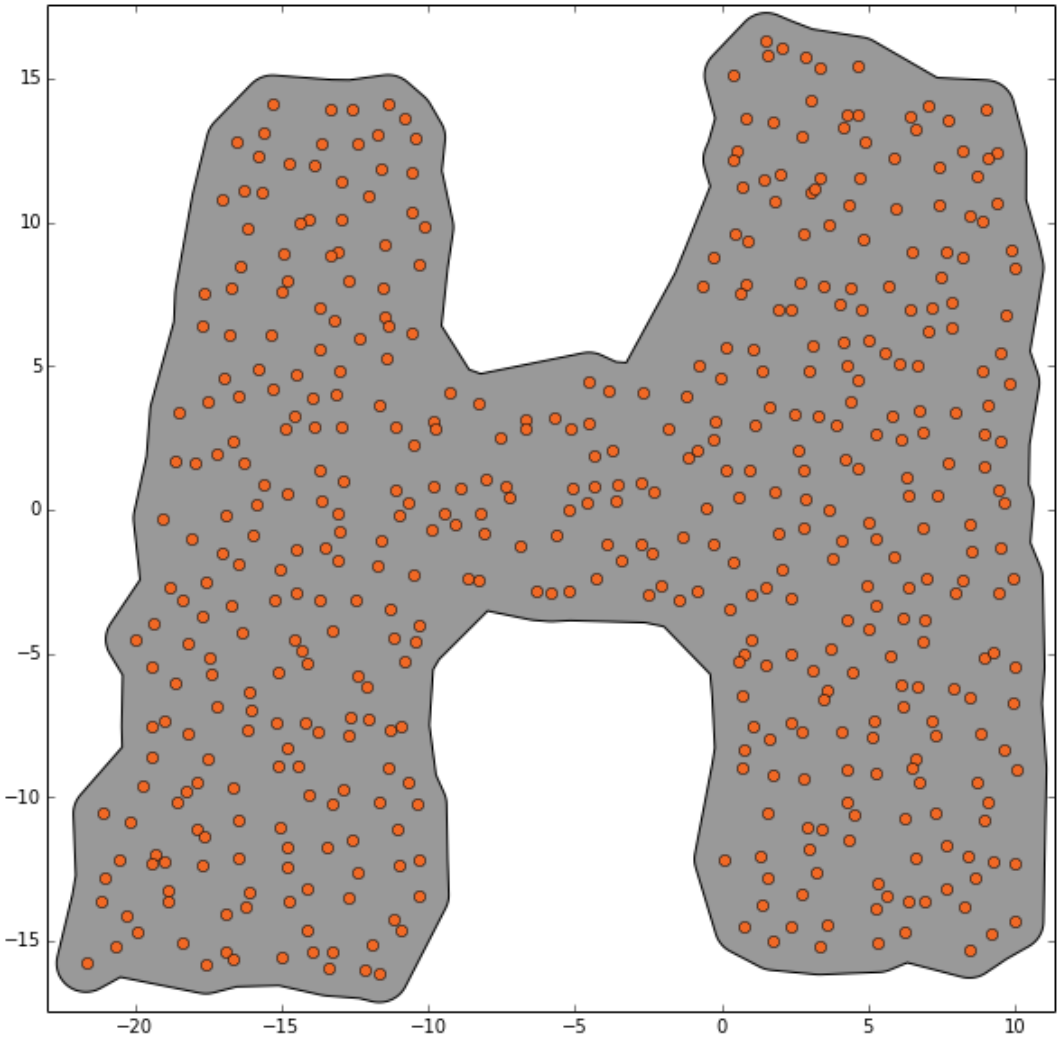
```
from matplotlib.collections import LineCollection
for i in range(9):
    alpha = (i+1)*.1
    concave_hull, edge_points = alpha_shape(new_points,
                                            alpha=alpha)
    #print concave_hull
    lines = LineCollection(edge_points)
    pl.figure(figsize=(10,10))
    pl.title('Alpha={0} Delaunay triangulation'.format(
        alpha))
    pl.gca().add_collection(lines)
    delaunay_points = np.array([point.coords[0]
                                for point in new_points])
    pl.plot(delaunay_points[:,0], delaunay_points[:,1],
            'o', hold=1, color='#f16824')

    _ = plot_polygon(concave_hull)
    _ = pl.plot(x,y,'o', color='#f16824')
```



So in this case, alpha of about 0.4 looks pretty good. We can use shapely's buffer operation to clean up that polygon a bit and smooth out any of the jagged edges.

```
alpha = .4
concave_hull, edge_points = alpha_shape(new_points,
                                        alpha=alpha)

plot_polygon(concave_hull)
_ = pl.plot(x,y,'o', color='#f16824')
plot_polygon(concave_hull.buffer(1))
_ = pl.plot(x,y,'o', color='#f16824')
```


15
10
5
0
−5
−10
−15
−20
−15
−10
−5
0
5
10



And there you have it. Hopefully this has been a useful tour through some of your geometric boundary options in python. I recommend exploring the Shapely manual to find out about all of the other easy geometric operations you have at your fingertips. Also, if you dig Python and playing with maps, we want to hear from you (http://thehumangeo.com/company.html#slide-5).

Careers Info