

# Open source geospatial programming in Python

## A crash course..

William Armstrong  
[william.armstrong@colorado.edu](mailto:william.armstrong@colorado.edu)

Geological Sciences  
Institute for Arctic and Alpine Research  
University of Colorado at Boulder

September 20, 2016

Talk available at [github.com/armstrwa/presentations](https://github.com/armstrwa/presentations)

# Today's goals

By the end of today's workshop, you will understand several fundamental Python concepts and be able to *programmatically*<sup>1</sup>

- Read, manipulate, and write shapefiles and raster data
- Perform simple analyses on geospatial data
- Plot data derived from geospatial sources

---

<sup>1</sup>Through program code, rather than a user interface – [wictionary.org](http://wictionary.org)

# Why geospatial Python?

- Provides a multitude of options for handling geospatial data analysis and display
- Allows easy automation of tasks and manipulation & analysis of large datasets
- Runs on Mac's (and Windows)..
- Free and open

# My motivation

Turning pretty pictures into quantitative data.

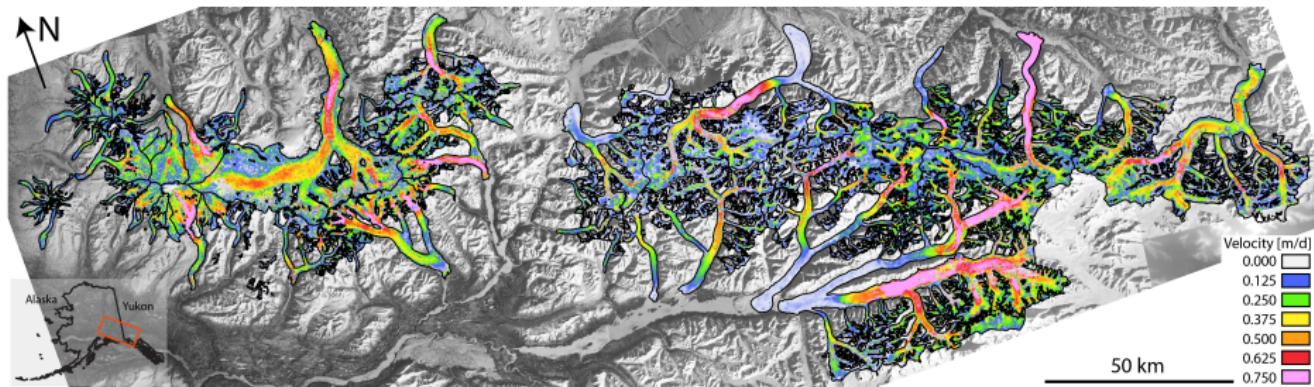


Figure: Glacier velocity on a range-wide scale over south-central Alaska.

# My motivation

Repeat analysis of satellite imagery to inform our understanding of glacier dynamics.

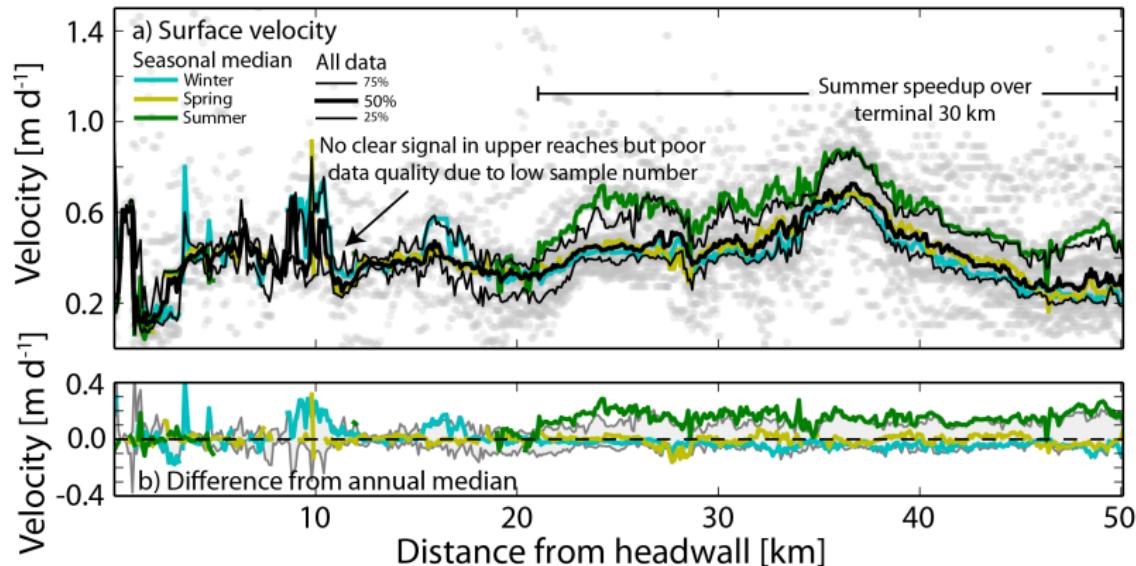


Figure: Seasonal glacier velocity along a down-glacier profile.

# Python basics: indexing data

Different data structures have different ways of accessing contained values.

- **Tuples:** Square brackets with [row][column] indexing. Start = 0.

---

```
1 t = ( (1,2) , (5,9) ) , t[0][1] = 9
```

---

- **Lists:** Square brackets with [order] indexing.

---

```
1 l = [ 1, 2, 5, 9], l[2] = 5
```

---

- **Dictionaries:** Square brackets with ['key'] indexing.

---

```
1 d = { 'age':( (33,31,19) ), \
2 'name': [ 'john' , 'claire' , 'steve' ] }
3 d[ 'age' ] = (33,31,19), d[ 'age' ][0] = 33
```

---

# Python basics: for loops

A **for loop** is a way to get Python to repeat a specified sequence of tasks many times. Python's for loop structure is very particular about indenting, unlike some other languages. A Python for loop looks like:

```
1 a = [ 13, 5, 2, 4, 9, 16]
2 for i in range(0, len(a)):
3     print "Value " + i + " is " + a[i]
4 print "Done with loop"
```



Figure: Around and around we go. Image from wikipedia.org

# Python basics: importing packages

Part of what allows Python to be so lightweight is that it only loads the tools you ask for. These tools are called “packages”; today we will be using:

---

```
1 # load a module and give it a short-hand
2 import matplotlib.pyplot as plt
3 import numpy as np
4 # load one module from a package
5 from shapely import geometry
6 # load a whole package
7 import fiona
8 import rasterio
```

---

# Geospatial datatypes

Python can readily handle geospatial data. Geospatial data comes in two basic classes:

**Raster:** A representation of the world as a surface divided into a regular grid of cells. E.g., satellite imagery, DEMs. *GDAL, Rasterio*.

**Vector:** A representation of the world using points, lines, and polygons. E.g., roads, study area boundaries. *OGR, Shapely, Fiona*.

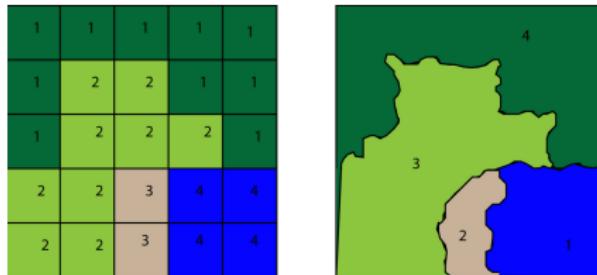


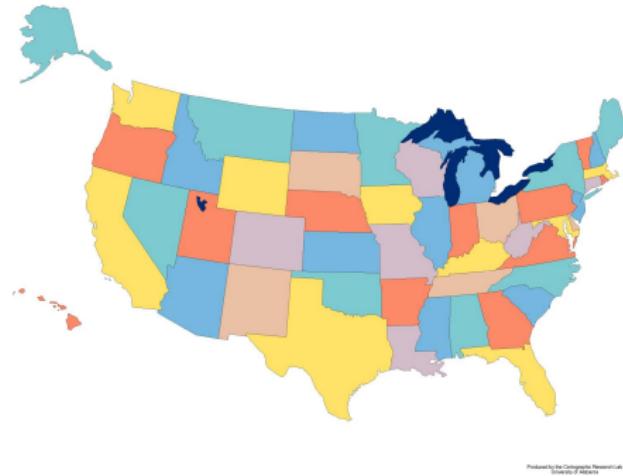
Figure: Raster (left) and vector (right) representations of a landscape.

# Today's dataset

State shapefiles from

[https://www.census.gov/geo/maps-data/data/cbf/cbf\\_state.html](https://www.census.gov/geo/maps-data/data/cbf/cbf_state.html)

First hit when Google “us state outlines shapefile” → “nation-based files”  
→ “state” → “cb\_2015\_us\_state\_20m.zip”



## Reading vector data: Fiona

Fiona and Shapely are lightweight wrappers for the underlying OGR module, which is pretty clunky and not ‘Pythonic’. Reading a shapefile with Fiona is pretty easy, as shown below.

```
1 stateFn = fullPath + 'cb_2015_us_state_20m.shp'
2 # open collection stateFn with 'r' for 'read'
3 states = fiona.open(stateFn, 'r')
4
5 # Some basic information
6 len(states) # number of features within collection
7 states.crs # collection coordinate reference system (CRS)
8 states.schema # Information on fields and geometry type
9 state = states[0] # get the first feature and name it 'state'
10
11 # loop over features, print state names and USPS abbrev.
12 for state in states:
13     print state['properties'][ 'NAME'], \
14         state['properties'][ 'STUSPS']
```

# Manipulating vector data: Shapely

Fiona reads shapefiles, but Shapely is where the real geospatial power lies.  
Creating a Shapely geometry from a Fiona input looks like:

---

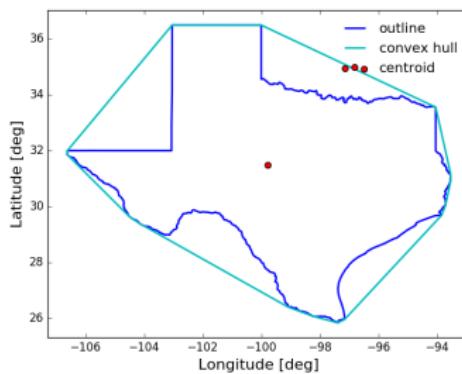
```
1 state = states[0] # get the first feature and name it 'state'
2 state.viewkeys() # ['geometry', 'type', 'id', 'properties']
3 shape = geometry.shape(state['geometry']) # make shapely geom
4 ext = shape.exterior # get the 'exterior' ring geom
5 x,y = ext.xy # 'exterior' ring coords
6 cent = ext.centroid # geometric centroid geom
7 cx,cy = cent.xy # centroid coords
8 hull = shape.convex_hull # convex hull geom
9 hx,hy = hull.exterior.xy # convex hull coords
```

---

See <http://toblerity.org/shapely/manual.html> for the many object attributes.

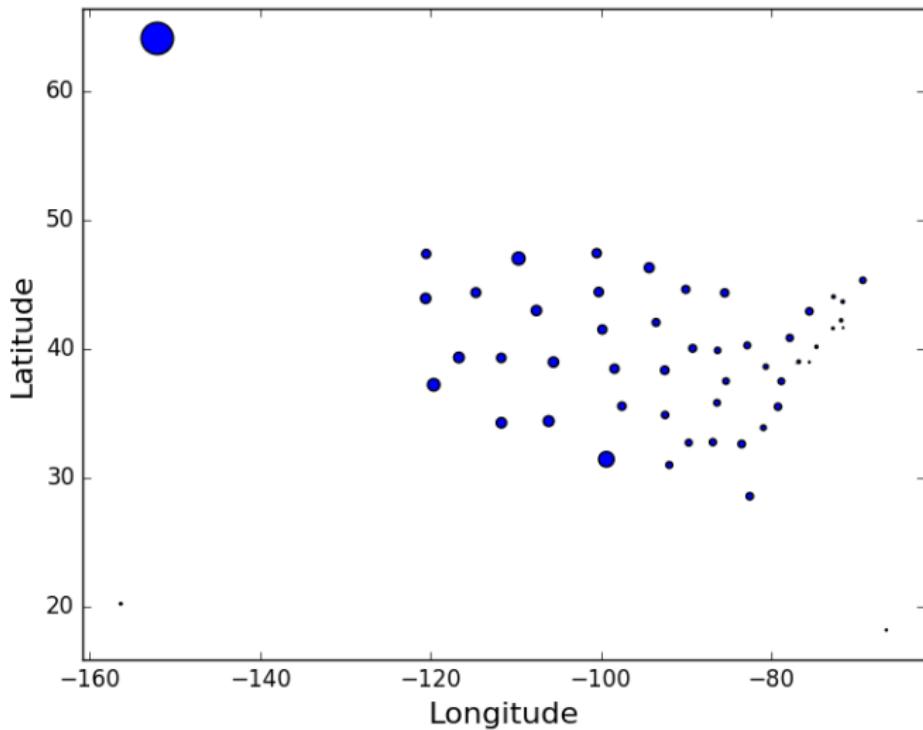
# Plotting data: Matplotlib

```
1 plt.plot(x,y,c='b',lw=2,label='outline') # c = color, lw = linewidth
2 plt.plot(hx,hy,c='c',lw=2,label='convex hull') # label on legend
3 plt.scatter(cx,cy,s=40,marker='o',facecolor='r',label='centroid')
4 plt.xlabel('Longitude [deg]', fontsize=16) # ^s = size
5 plt.ylabel('Latitude [deg]', fontsize=16)
6 plt.legend(loc=1,frameon=False) # loc 1 = upper right corner
7 plt.axis('tight') # less white space
8 plt.savefig('texas_shapely.png', r=300) # r = resolution
9 plt.show()
10 plt.close()
```



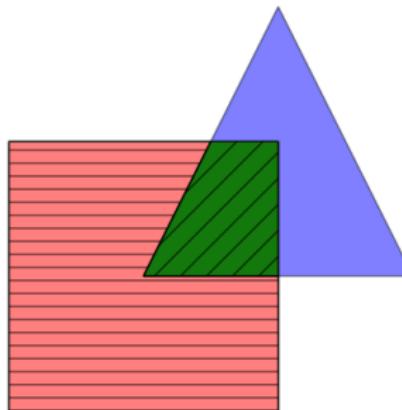
# Vector data: Fiona and Shapely

```
1 # Initialize empty lists for later storage
2 cxList = []
3 cyList = []
4 areaList = []
5
6 for state in states: # loop over all features in collection
7     # make temporary shapely geometry
8     tmpShp = geometry.shape(state['geometry'])
9     tmpX,tmpY = tmpShp.centroid.xy # x,y coords of centroid
10    tmpArea = tmpShp.area # get geom's area
11    # add data to lists for storage
12    cxList.append(tmpX)
13    cyList.append(tmpY)
14    areaList.append(tmpArea)
15
16 plt.scatter(cxList,cyList,s=areaList*10)
17 plt.axis('tight')
18 plt.xlabel('Longitude', fontsize=16)
19 plt.ylabel('Latitude', fontsize=16)
20 plt.savefig('statesByArea.png', r=300)
21 plt.show()
22 plt.close()
```



# Vector data: Shapely operations

```
1 # Dummy data with simple geometries
2 square = geometry.Polygon([(0,0),(1,0),(1,1),(0,1)])
3 triangle = geometry.Polygon([(0.5,0.5),(1.5,0.5),(1,1.5)])
4 square.intersects(triangle) # Returns true if overlap
5 isect = square.intersection(triangle) # 'overlap'
6 diff = square.difference(triangle) # 'subtracts'
7 uni = square.union(triangle) # 'combines'
```



# Vector data: Writing out

```
1 # Open file to write; define driver, crs, and schema
2 with fiona.open('out.shp', 'w', driver=source.driver, crs=source.crs
3                 schema = source.schema) as sink:
4     # Iterate over features to write
5     for feat in geoms:
6         # get geometry using shapely
7         geom = geometry.shape(feat['geometry'])
8         # get attributes from lists
9         prop = {'dist': dist, 'name': name, 'id': id}
10        # write geometry to file
11        sink.write({'geometry': mapping(geom),
12                    'properties': prop})
```

The schema defines the fields contained within a geometry.

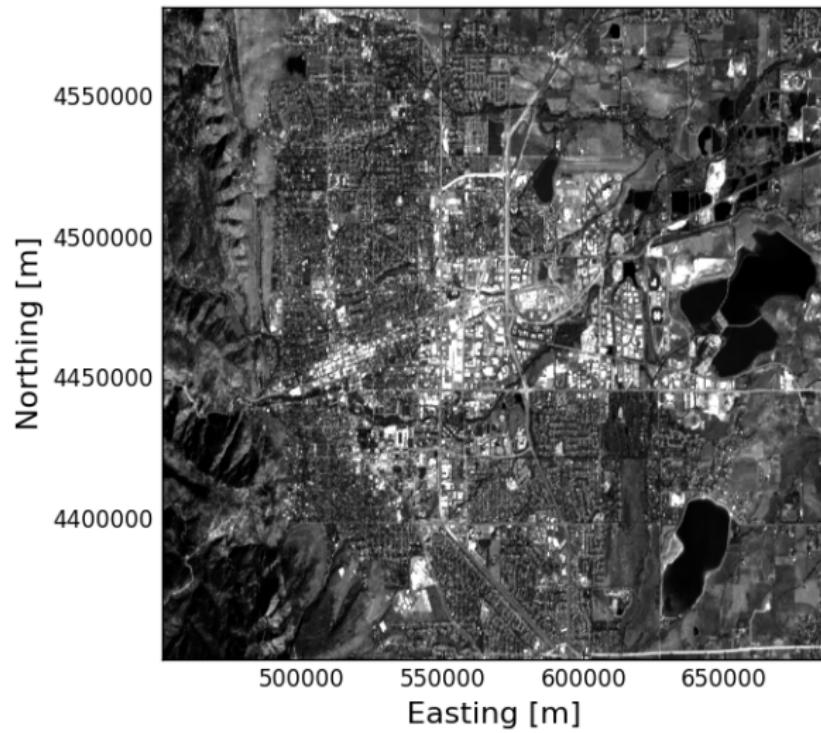
```
1 source.schema = {'geometry': 'Polygon',
2                   'properties': {'dist': 'float:13.3',
3                                 'name': 'str',
4                                 'id': 'int'}}
```

# Raster data: Plotting

Rasterio is a wrapper for GDAL, which, like OGR, is a little clunky and unintuitive to the average Python programmer. Opening, subsetting, and plotting an image with Rasterio and Matplotlib looks like:

```
1 src = rasterio.open(rasterFn) # open raster
2 # turn band 1 to numpy array; only use subset of rows and columns
3 sub = src.read(1,window=w) # w = ( (rMin,rMax), (cMin,cMax) )
4 # plot image; clim = min/max brightness; use 'gray' colormap
5 plt.imshow(sub,cmap='gray',clim=((bLow,bHigh)),
6             extent=((xmin,xmax,ymin,ymax))) # x,y coords not row,col
7 plt.show()
```

# Raster data: Plotting



# Raster data: Transformation

Transforming between image coordinates and pixel coordinates is very easy.

---

```
1 row,col = src.index(x,y) # get pixel coords from ground coords
2 x,y = src.affine * (row,col) # get ground from pixel coord
```

---

As is finding out important image info.

---

```
1 crs = src.crs # coordinate reference system
2 bnd = src.bounds # image bounding box coordinates
3 nRow = src.height # number of pixels N-S
4 nCol = src.height # number of pixels E-W
```

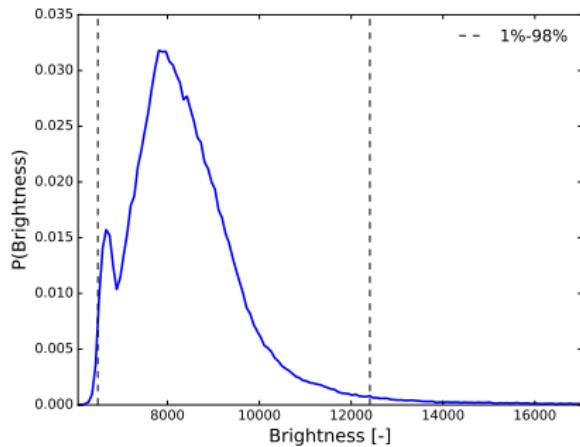
---

## Raster data: Analysis and manipulation

You can do anything with an image-derived array as you would with a regular numpy array. For example,

- Measure value at a certain pixel (lat/lon) over time
- Histogram to get image characteristics
- Difference to images to get change over time
- Perform mathematical operations combining multiple arrays to create a new raster
- Image enhancement and processing using e.g., `scikit.image`, `cv2`, or `scipy`

# Raster data: Analysis and manipulation



---

```
1 p2,p98 = np.percentile(sub,[2,98])  
2 hist,edges = np.histogram(sub,bins=500)
```

---

# Raster data: Writing

Writing out GeoTIFFs is straight forward. You create a new file with geospatial attributes (crs, height, etc.) and then specify with which dataset to populate the empty file.

```
1 ...
2 'w' = write, dtype = datatype (i.e., int, float)
3 crs = coordinate reference, driver = file format
4 transform = geographic transformation coefficients
5 ...
6 out = rasterio.open('boulderSub.tif', 'w', driver='GTiff',
7                      height=sub.shape[0], width=sub.shape[1],
8                      count=1, dtype=sub.dtype,
9                      crs=src.crs, transform=src.transform)
10 out.write(sub,1) # write data to file
11 out.close() # 'flush' file
```

The crs and transform arguments can be taken from an existing dataset, as done here, or created anew.

```
src.crs = CRS({'init':u'epsg:32613'})
```

```
src.transform = [450292.5,15.0,0.0,4582207.5,0.0,-15.0]
```

## Extensions: other useful packages

- **GDAL**: library underlying Rasterio. Has some really nice tools for working with DEMs.
- **OGR**: library underlying Shapely and Fiona. Has nice tools for translating vector data between formats.
- **PyProj**: projection and coordinate transformation.
- **GeoJSON**: storing geospatial information in a dict-like structure.
- **GeoPandas**: data structure and data analysis tool for geospatial data.
- **Descartes**: makes plotting Shapely geometries a little easier.

# Thanks! william.armstrong@colorado.edu

