Introduction to Python for Geospatial Applications

Author: Thailynn Munroe | University of Alabama in Huntsville | NASA SERVIR-Mekong

Training overview:

- General overview of Python and programming syntax
- Download of specific packages for training
- Working with vector data using GeoPandas
- Working with raster data using Rasterio

This tutorial is meant to be a broad overview of the Python programming language. By the end of this tutorial you should have a general understanding of how the Python language is structured, the syntax for writing applications, and how to utilize Python to read in, manipulate, and display geospatial data. Full Python documentation can be found at https://docs.python.org/3/tutorial/index.html. Some other great resources for general Python help include https://gis.stackexchange.com/ and https://stackoverflow.com/questions/tagged/python.

There will be a lot of vocabulary that may be unfamiliar, but the main focus of this training is to help you understand the structure and techniques used in Python. If at any point there is a word that you need the definition for, **please see the glossary at the end of this document.**

What is Python?

Python is an *object-oriented*, *high-level* programming language with many libraries, and several specifically for earth science applications. *High-level* refers to the fact that the syntax for this language is easy to read, and instead of coding in binary (the language that computers operate on) with 0's and 1's, you can use words that represent what is being carried out. In the background, Python compiles your commands and translates them into binary for the computer. *Object-oriented* means that this programming language is based on how the data stored in variables is organized and described by other parameters. This format makes it easier to manipulate and analyze data.

Structure of the Python language:

Python is made up of many *modules*, which contain *classes*. If you are familiar with Python, some common examples of modules would be Numpy and Matplotlib. A *module* is a .py file that groups together *classes*, which are like blueprints for describing an *object*. All objects have *attributes*, which are properties that help describe and define the object. Attributes can describe a class as a whole or an *instance* of a class. Python functions are known as *methods*. Methods are just actions one can perform to manipulate data, display, or the Python environment. Methods

take *arguments*, which generally tell the method what to manipulate and specify how. We will deal with all of these in more detail as we move through the tutorial.

Jupyter/IPython Notebooks:

The graphical user interface we will be working with today is the IPython Notebook. The IPython Notebook allows you to work with Python in an environment where you can write, comment, execute, and display outputs from sections of your code without running the entire code (this would be useful if you just want to check for errors in a part of your code, or if running your whole code would take a long time). These notebooks can be easily uploaded and shared on GitHub, which is a rich online community where programmers can share their work. Documentation for IPython Notebooks can be found at http://ipython.readthedocs.io/en/stable/. To reach the Jupyter Notebook, if you have Python installed on your computer you can follow the steps below:

1. Open up a terminal
2. Navigate to the directory where the files you want to be working with are stored
    a. Example -> cd users/tsmunroe/documents/anaconda
    b. This is the "working directory"
3. Type in: jupyter notebook
    a. It will open the webpage with the notebook interface

Error handling in Jupyter Notebooks:

Often times, you will run into errors when writing and executing codes. Jupyter Notebooks will return an error message where you would normally see the output. The unique thing about these error messages is that it will trace the error back through all the modules used to compile and execute the code in the background. This can help you figure out what is wrong in your code and how to fix it.

Importing a module:

In a python environment, you need to import the modules you are working with at the beginning of a file. There are many commonly used modules that come pre-installed with your Python manager; however, there are so many modules that if they were all pre-loaded, it would take a very long time to run your code. If you need a module that isn't already installed, you need to download and install it before you can use it. Once it is installed, you can import it into your current environment. The syntax for importing the Matplotlib module (previously described) would be as follows:

```
>>> import matplotlib
```

You can also import a single specific class from a module. After typing "import 'module.class'", you can use "as 'shortened version'" to create a nickname/alias for the module or class you are

importing. In the following example, we are only importing the *class* Pyplot. The GeoPandas module is built off of the Matplotlib framework, and therefore uses many of the same methods. The syntax for importing the Pyplot class (previously described) would be as follows:

```
>>> import matplotlib.pyplot as plt
```

We will also need to import the two geospatial modules we will be working with later into our environment. We can do that now:

```
>>> import geopandas as gpd
>>> import rasterio
```

Notice how we shortened the name of the Geopandas module to "gpd". Throughout the code we can now use "gpd" to call the Geopandas module and any classes within the module.

GeoPandas

GeoPandas is a cartography focused Python module built off of a few other Python libraries including Pandas, Descartes, and Matplotlib. GeoPandas takes all the data processing and displaying methods and from these modules and creates a single module makes it easy to work with vector data. GeoPandas works with two main levels of data: GeoDataFrames and GeoSeries. A GeoDataFrame can have many different GeoSeries, i.e. a country shapefile, a cities shapefile, and a roads shapefile. Both GeoDataFrames and GeoSeries are vector files that have attributes that describe them (similar to what you might find in an attribute table). You can also perform operations to a single GeoSeries, or between two GeoSeries.

Read in vector data:

Just as we had to import the relevant geospatial Python modules into our Python environment, we also have to import, or read in, the data. First we will see an example of reading in data from the GeoPandas database. Once we have GeoPandas imported into our Python environment, the command for reading in a data frame or a shapefile is gpd.read_file(). Remember that "gpd" is the nickname we gave the GeoPandas module. The read_file() method takes a *file* path as an argument. Normally, the file path would be a string object that points to the location of a shapefile on your local drive. Here, GeoPandas needs to first get the path of the file within its geodatabase before it can read it in. GeoPandas has a method for that as well. Try loading in a built-in data frame to your Python environment using the following code:

```
>>> earth =
gpd.read_file(gpd.datasets.get_path("naturalearth_lowres"))
```

Now we will look at some of the layers and attributes stored in the "earth" GeoDataFrame by using the "head()" method. This will print out the rows of each "Continent" GeoSeries and their attribute fields including names of countries and population estimates. After looking at the output of "head()" we will have a better understanding of the information we can display in our plots. Try the method below:

```
>>> earth.head()
```

Your output should look similar to this:

Out[6]:

	continent	gdp_md_est	geometry	iso_a3	name	pop_est
0	Asia	22270.0	POLYGON ((61.21081709172574 35.65007233330923,...	AFG	Afghanistan	28400000.0
1	Africa	110300.0	(POLYGON ((16.32652835456705 -5.87747039146621...	AGO	Angola	12799293.0
2	Europe	21810.0	POLYGON ((20.59024743010491 41.85540416113361,...	ALB	Albania	3639453.0
3	Asia	184300.0	POLYGON ((51.57951867046327 24.24549713795111,...	ARE	United Arab Emirates	4798491.0
4	South America	573900.0	(POLYGON ((-65.50000000000003 -55.199999999999...	ARG	Argentina	40913584.0

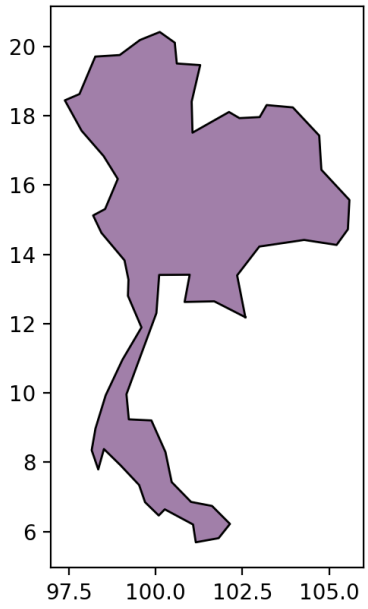
For our plot, we will call the specific shapefile for Thailand *from within* the GeoDataFrame. Notice that each group of information (i.e. each GeoSeries) has two columns. We will look for the name of the country by calling the "name" attribute of the earth GeoDataFrame that is set to "Thailand", then set the output of that expression to a separate variable:

```
>>> thailand = earth[(earth.name == "Thailand")]
```

We have just created a new object. This object now has its own attributes and different methods we can perform on it. Now we need to plot the data, which stores it in the background, and then display the result:

```
>>> thailand.plot()
>>> plt.show()
```

Your result should look like this:



Manipulate the display of vector data:

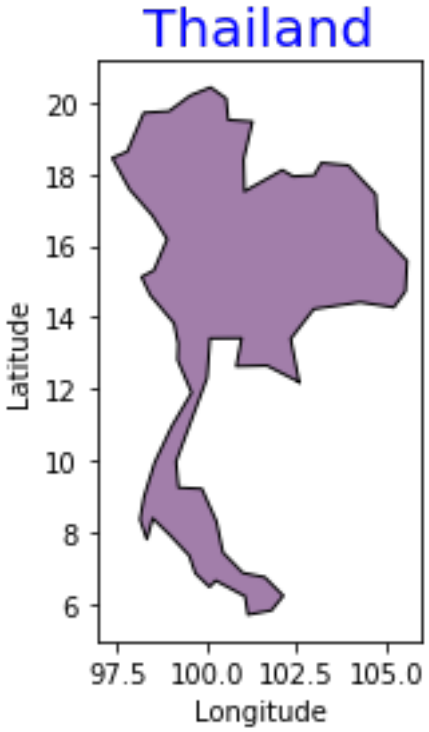
GeoPandas comes with many different methods for manipulating the display of a graph or plot based off of Matplotlib. Notice in the map of Thailand there are two axes. There are methods to change the labels, spacing, color, grid, title, background, you name it. The full documentation for Matplotlib can be found at https://matplotlib.org/users/. For now, we will just look at the general syntax for adjusting some simple aspects of the plot. If we wanted to add a title and labels to the x and y axes, we would call the methods "title()", "xlabel()", and "ylabel()". We could also change the font size, type, color, and orientation. These methods are shown below:

```
>>> thailand.plot()
>>> plt.title("Thailand", fontsize = 20, color = "blue")
>>> plt.xlabel("Longitude")
>>> plt.ylabel("Latitude")
```

Once you are done setting the labels, you need to show the plot again:

```
>>> plt.show()
```

Your new plot should look similar to this:



Work with different attributes of the vector data:

We can create a plot of the countries in the Mekong Region by using the same method we did earlier to find Thailand within the earth GeoDataFrame. After we do that, we will explore different ways we could display the data based on their unique attributes.

Example:

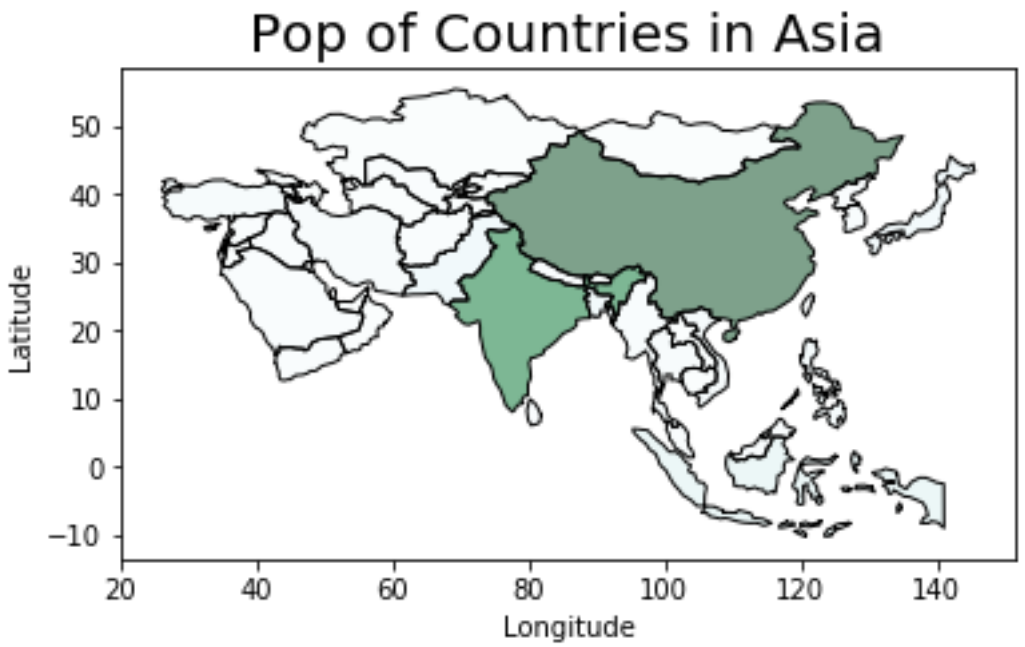
Let's call all the countries in Asia:

```
>>> countries = earth[(continent == "Asia")]
```

Now let's create a "choropleth" map and plot the Asian countries in order of population. When we call the "plot()" method this time, we can choose the attribute column we would like to use as an argument. We can also choose the color scheme we want to display. A full list of Matplotlib color schemes can be found at https://matplotlib.org/examples/color/colormaps_reference.html. Now try the code below:

```
>>> countries.plot(column='pop_est', cmap='BuGn')
>>> plt.title("Pop of Countries in Asia", fontsize = 20)
>>> plt.xlabel("Longitude")
>>> plt.ylabel("Latitude")
>>> plt.show()
```

Your new map should look similar to this:



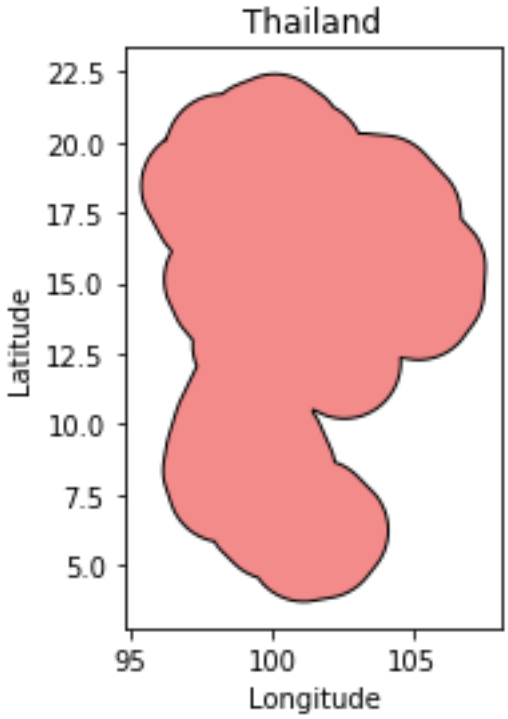
Perform some geoprocessing to the vector data:

We can easily perform simple geoprocessing on our shapefiles in the same manner we have been manipulating the display of our plots. It is only a matter of calling the right method for the object you are wanting to manipulate. GeoPandas has a comprehensive list of the "Geometric Manipulations" the module can perform at http://geopandas.org/geometric_manipulations.html.

Among these are the familiar methods of buffer, centroid, and rotate. Each of these methods operates on a GeoSeries object, and returns a separate GeoSeries object. We will demonstrate the buffer method on the shapefile of Thailand. Buffer simply takes the distance around the feature as an argument as shown below:

```
>>> bufthai = thailand.buffer(2)
>>> bufthai.plot()
>>> plt.title("Thailand")
>>> plt.xlabel("Longitude")
>>> plt.ylabel("Latitude")
>>> plt.show()
```

Your new plot should look similar to this:



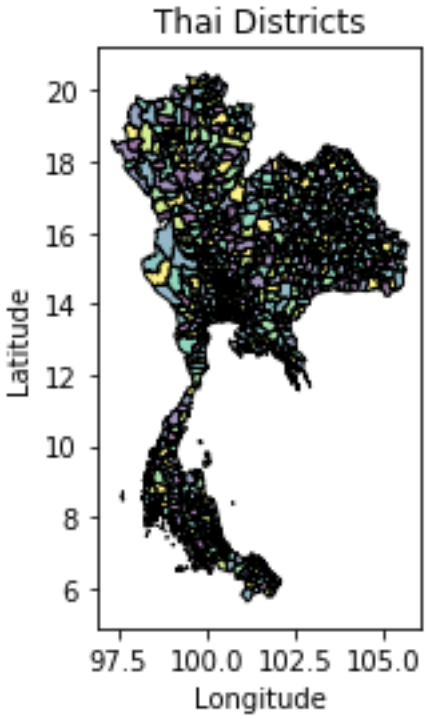
Note:

If you perform an operation between two GeoSeries, for instance intersection, it will perform the action feature to feature. So, the first feature in each GeoSeries would be compared to each other, the second features in each GeoSeries would be compared to each other, etc. If you want to perform an action between two GeoSeries, you need to combine all the features within the series into a single feature (i.e. many points into a single multipoint feature). To do this you would use the method “unary_union()”.

We can also read in shapefiles from your local drive. You need the .shp and the .shx files in the working directory in order to read in a shapefile. The .shx file contains information about the shapefile that the computer needs access to. In this case we will look districts in Thailand. Let’s read in and plot the districts:

```
>>> thai = gpd.read_file('THA_adm0.shp')
>>> thai.plot()
>>> plt.title('Thai Districts')
>>> plt.xlabel('Longitude')
>>> plt.ylabel('Latitude')
>>> plt.show()
```

Your outputs should look similar to this:



Note: If you want to display background maps, another module called basemap can be installed and imported to display these maps in the background.

Save your vector dataset:

If you want to save your vector dataset to your local machine, GeoPandas has a method called "to_file()". You need to specify the "driver" which tells you what type of file it will be saved as. You can use the following syntax:

```
>>> thai.to_file('new_shape.shp', driver = 'ESRI Shapefile')
```

Rasterio

Rasterio is a Python module that makes working with rasters and their associated geospatial information. With Rasterio, you can perform many of the raster operations commonly performed in our field. These include georeferencing, resampling, creating masks, and performing raster calculations. The Rasterio module works with rasters in the form of Numpy arrays. As mentioned before, arrays are a common way to store large numerical datasets. In the case of a raster dataset, the array returns a matrix with the same dimensions as the raster, with a single value for each pixel. In this format, you can perform calculations or manipulations on each pixel

or sections of pixels relatively fast and easily. The full documentation of Rasterio can be found at https://mapbox.github.io/rasterio/topics/plotting.html, and another tutorial for some simple Rasterio handlings can be found here https://github.com/mapbox/rasterio/blob/master/docs/quickstart.rst.

Read in raster data:

Just as there was a method to read in vector data, there is a Rasterio specific method for reading in raster data. The method "open()" takes a file path as an argument, and returns a raster dataset object. We need to double check that we have imported Rasterio into the environment, then we can open a raster using its local file name:

```
>>> tif = rasterio.open(r'/Users/tsmunroe/anaconda/shape-detection/L5_B1.tif')
```

Currently, the raster is open in read mode. Now that the raster is open in our Python environment, we can find out some information about its attributes using different methods. First, let's check the dimensions of the raster. We can do this using the "print()" method, and printing the height and width attributes:

```
>>> print(tif.height)
>>> print(tif.width)
```

A geographic dataset will also be georeferenced, which helps display the image in the correct place on a map. Rasterio allows us to explore the geographic attributes of the raster as well. We can look at the bounding box of the raster, and the coordinate reference system (crs), which helps translate the bounding box values to places on the globe.

```
>>> print(tif.bounds)
>>> print(tif.crs)
>>> print(tif.meta)
```

The output statement from this code should look similar to this:

```
BoundingBox(left=587085.0, bottom=1493685.0, right=821415.0, top=1701615.0)
CRS({'init': 'epsg:32648'})
{'driver': 'GTiff', 'dtype': 'int16', 'nodata': -9999.0, 'width': 7811, 'height': 6931, 'count': 1, 'crs': CRS({'ini
t': 'epsg:32648'}), 'transform': Affine(30.0, 0.0, 587085.0,
       0.0, -30.0, 1701615.0)}
```

These numbers tell you the left most bound of the image, the bottom most bound of the image, etc. In this case, the values are in decimal degrees, but they could be in meters or any other unit. The coordinate reference system helps to determine where on the map the

Often times, a raster dataset has more than one band associated with it. In the case of an RBG image, one raster dataset has 3 bands. If you want to work with a single band, first you need to know the index of the band you want to work with. The band indices can be found by looking at the index attribute. These bands are actually stored as arrays, so if we want to look at the pixel values in a single band, we can read the array that contains them:

```
>>> print(tif.indexes)
>>> tif.read(1)
```

Manipulate the display of raster data:

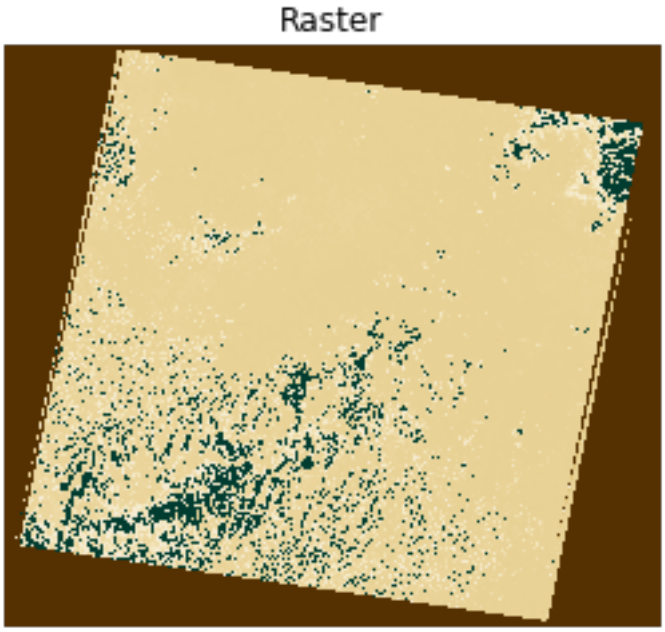
If we want to display the raster we can use the Matplotlib class Pyplot that we have been working with. Remember that it is already loaded into our environment with the nickname “plt”. The method “imshow()” shows an image within a plot. The first argument is the raster dataset object to plot, and the second is the color scheme you want it displayed in. Because we opened the raster with “rasterio.open()”, the filepath is read in as a “Dataset reader” object. For this reason, you need to specify the band index you want displayed, which is done using the “read()” method:

```
>>> plt.imshow(tif.read(1), cmap='BuGn')
>>> plt.show()
```

Other Pyplot functions are available for this plot as well. Try changing the title on your plot.

```
>>> plt.imshow(tif.read(1),cmap="BrBG")
>>> plt.axis('off')
>>> plt.title('Raster')
>>> plt.show()
```

Your output might look similar to this:



Perform a simple raster calculation:

Rasterio allows you to perform raster calculations to the entire image all at once. In the next exercise, we will be calculating an index to find water in a raster image. Here we will look at how easy it can be to perform raster calculations:

```
>>> tif1 = rasterio.open(r'/Users/tsmunroe/anaconda/shape-
detection/L5_B2.tif')
>>> tif2 = rasterio.open(r'/Users/tsmunroe/anaconda/shape-
detection/L5_B3.tif')
>>> new = tif1.read(1)-tif2.read(1)
>>> plt.imshow(new)
>>> plt.title('New Raster')
>>> plt.show()
```

Save your raster dataset:

If you want to save your vector dataset to your local machine, you need to open a new raster file in "write" mode. Recall earlier that we were working with a raster dataset in read mode, where we could only look at properties of the dataset, but not change the original dataset. Now we will create a new file in "write" mode, where we can change the data within the file. We need to define the file as a tiff, with all the same parameters as the image we were working with before:

```
>>> image = tif2.read(1)
>>> meta = image.meta
>>> with rasterio.open('test.tif','w',**meta) as dst:
          dst.set_crs(image.crs)
          dst.write(new,1)
```

Appendix

Print your data:

There are many ways to display your data or the output of your code depending on what your project entails. The print function allows you to view the information stored in variables, or view the result of a function or mathematical operation (we will go more in depth on functions in a bit). The print function can be very useful when trying to debug (fix errors) your code or to visualize and understand different data types. Try the following print statement example:

```
>>> strng = "Hello SARI friends!"
>>> print(strng)
```

In this case, "strng" is the variable that houses the *string* "Hello SARI friends!" A string is one of Pythons main data types.

Data types:

Functions in Python each work with different data types. Data types include numeric types, sequences, sets, and mapping types. Numeric types include integers, floats (decimals), and complex numbers. Sequence types include strings (combinations of letters and/or numbers), lists, arrays, and tuples (lists that cannot be manipulated). A set refers to "an unordered collection of unique items". In our case, an example of a set could be overlapping polygons, like animal habitats. Finally, the mapping data type is also known as a dictionary. It is not geographic mapping. A dictionary is similar to a list, but certain "keys" in the list each link to a "value". All of these are stored as *objects*, and they have *attributes,* or information that describes them. Below are some examples of the different data types.

Examples:

Numeric data types (integer, float):

```
>>> num = 1
>>> dec = 1.2345
```

With numeric data types, you can perform mathematic operations between numbers and assign the result to a new variable. These operations are denoted by + (add), - (subtract), * (multiply), and / (divide). You can perform those same mathematical operations to variables that are already defined and save them to a new variable, or update the old one. Try printing the following operations:

```
>>> mult = 5*300
>>> div = mult/20
>>> div = div*4
```

Algebra in Python follows the normal order of operations, and you can use unlimited parentheses to denote priority in operations.

Sequence data types (string, list, array):

```
>>> path = r"\\Mac\Home\Downloads\map_template\example_doc.mxd"
```

There are special operations you can perform on strings called "slicing". This allows you to grab just a section of the string. Imagine each character in the string is located at an index. The indexing starts at the 0^{th} place. So, in our string example above, "\" is in the 0^{th} place. The indices of the characters in the string above go from 0 to 15, including spaces and special characters. If you wanted to slice the string to only return the word "Downloads", you would use the following syntax:

```
>>> print(path[11:20])
```

Notice how it seems like we are including the space following the word, but the slicing operation returns the characters at the index locations up to but not including that end index you provide. You can also count backwards from the end of the string using negative numbers. If you are starting from the 0^{th} place you do not need to specify that. This will be especially handy when working with file paths and saving outputs from codes. We can also add two strings together by using the a "+". This is called concatenation, and will return a new string. Take a look at the following example:

```
>>> new_path = path[:-15]+ "new_file.doc"
```

The following is an example of a list:

```
>>> lst = ["This","is","a","list","of","strings"]
```

Lists simply hold values in a row, and you can rearrange, add, delete, index through, and many other things to the variables inside the list. Lists can hold numbers or strings or even other lists. Many times, programmers will use lists as an index for an iterative process. Because lists can be so helpful, for a more in-depth explanation visit https://docs.python.org/3/tutorial/datastructures.html

The following is an example of an array:

```
>>> arr = np.array([1,2,3], [3,4,5], [2,3,4])
```

Special note on arrays:

Arrays are a special type of sequence, as this is the format that is most commonly used to work with quantitative datasets, such as *in situ* measurements. With an array, you can perform an operation on every number within the array all at once. For example, if you wanted to divide every number in the array by 2, you could use the following code:

```
>>> arr_div = arr/2
```

If you were using a list, the computer would be going over each number one by one. Often times, **images are also read in or formatted as an array**. A 5-pixel by 10-pixel image with 3 bands would be called a 3D array, and would consist of 5 columns and 10 rows, 3 times. Each number would represent the value of a pixel. For an image object that is stored as an array, there are also sometimes descriptive attributes that would help you determine things like bit depth and units. You can turn lists into arrays by defining the list with the Numpy class array, as shown above.

Mapping data type (dictionary):

```
>>> dict = {"a":"apple","b":"banana","c":"cat"}
>>> print("c: ",dict["c"])
```

Try doing print statements for some of these variables and see what they look like. You can also print the outputs of functions directly. Here is an example: If you wanted to know the type of a variable, you can use the attribute "type()" to find out the variable type.

```
>>> print(type(num))
```

Working with methods:

Methods are the tools used to manipulate your data, perform actions, and set parameters within your python environment. We already saw a few methods with "print()" and "type()". Python is

a powerful language for creating your own methods as well, but for now we will focus on methods that already exist.

Glossary

Attribute – characteristics that describe an object. There are class (global) as well as instance (local) attributes.

Class – a structure or "blueprint" that outlines all the information needed to define certain things. This site, http://www.jesshamrick.com/2011/05/18/an-introduction-to-classes-and-inheritance-in-python/ explains classes with many metaphors.

Concatenation – adding two strings together to make a new string

Driver – specifies the type of file to write data as

Environment – the graphical user interface where you write and execute your code

High-level – a language written in terms that seem more like "human language", and needs to be translated into "machine language"

Low-level – a language that more closely resembles thy syntax that machines use to communicate and execute operations. Low-level languages run much faster.

Method – a function that performs some manipulation on an object

Module – a group of classes and objects that are somewhat related to each other

Object – a "variable" that has different attributes and can be manipulated, described, and/or displayed

Object-oriented – meaning that the methods and attributes are all built for specific objects and work