Lab 3: Introduction to Python I

Attribution

The content for this lab is taken **directly** from the AGU 2021 Python for Earth Sciences workshop, developed and led by Rebekah Esmaili (http://www.rebekahesmaili.com/) (bekah@umd.edu (mailto:bekah@umd.edu)), Research Scientist, STC/JPSS. We are grateful for Rebekah's generous support of Open Science and sharing all her hard work!

Check out the original GitHub for the workshop, which also contains additional modules and great links to resources:

https://github.com/modern-tools-workshop/AGU-python-workshop-2021 (https://github.com/modern-tools-workshop/AGU-python-workshop-2021)

Why Python?

Pros

- General-purpose, cross-platform
- Free and open source
- Reasonably easy to learn
- Expressive and succinct code, forces good style
- Being interpreted and dynamically typed makes it great for data analysis
- Robust ecosystem of scientific libraries, including powerful statistical and visualization packages
- Large community of scientific users and large existing codebases
- Major investment into Python ecosystem by Earth science research agencies, including NASA, NCAR, UK Met Office, and Lamont-Doherty Earth Observatory. See Pangeo.
- Reads Earth science data formats like HDF, NetCDF, GRIB

Cons

- Performance penalties for interpreted languages, although many libraries are wrappers for compiled languages. Avoid large loops in favor of matrix/vector operations when possible.
- Multithreading is limited due to the Global Interpreter Lock, but other parallelism is available
- See Julia for a modern scientific language which is trying to overcome these challenges

Why we use Python 3?

- Python 2 reached it's "end of life" as of January 2020
- No more updates or bugfixes
- No further official support
- Subtle differences: https://www.geeksforgeeks.org/important-differences-between-python-2-x-and-python-3-x-with-examples/ (https://www.geeksforgeeks.org/important-differences-between-python-2-x-and-python-3-x-with-examples/)

Lesson Objectives

- You will learn to:
  - Import relevant packages for scientific programming
  - Read ascii data
  - Basic plotting and visualization

Python outside of class

- You may use the same binder links to run these notebooks from anywhere
- (-- link to instructions for installing conda and using notebook locally --)
- (-- add google colab --)

Basic Python Syntax

The most basic Python command is to write words to the screen. In jupyter notebooks, the result will appear below the line of code. To run the above command in Jupyter notebook, highlight the cell and either chick the run button (▶) or press the **Shift** and **Enter** keys

In [1]:
```
# This is a comment, python will not run this!
print("Hello Earth")
```

```
Hello Earth
```

In Python, variables are dynamically allocated, which means that you do not need to declare the type or size prior to storing data in them. Instead, Python will automatically guess the variable type based on the content of what you are assigning:

In [2]:
```python
var_int = 8
var_float = 15.0
var_scifloat = 4e8
var_complex = complex(4, 2)
var_greetings = 'Hello Earth'
```

Python has many built in functions, the syntax is usually:

```
function_name(inputs)
```

You have already used two functions: *print()* and *complex()*. Another useful function is *type()*, will tell us if the variable is an integer, a float, a complex number, or a string.

In [3]:
```python
 type(var_int),  type(var_float), type(var_scifloat), type(var_complex), type(var_greetings)
```

Out[3]:
```
(int, float, float, complex, str)
```

Python has the following built-in operators:

- Addition, subtraction, multiplication, division: +, -, *, /
- Exponential, integer division, modulus: **, //, %

In [4]:
```python
2+2.0, var_int**2, var_float//var_int, var_float%var_int
```

Out[4]:
```
(4.0, 64, 1.0, 7.0)
```

Exercise 1:

1. Use *type()* to see if the following are floats and integers:
    - 2+2
    - 2*2.0
    - var_float/var_int

Solution:

In [5]:
```python
type(2+2)
```

Out[5]:
```
int
```

In [6]:
```python
type(2*2.0)
```

Out[6]:
```
float
```

In [7]:
```python
type(var_float/var_int)
```

Out[7]:
```
float
```

Working with lists

Lists are useful for storing scientific data. Lists are made using square brackets. They can hold any data type (integers, floats, and strings) and even mixtures of the two.

In [8]:
```python
numbers_list = [4, 8, 15, 16, 23]
```

You can access elements of the list using the index. Python is zero based, so index 0 retrieves the first element.

In [9]:
```python
numbers_list[3]
```

Out[9]:
```
16
```

New items can also be appended to the list using the append function, which has the syntax:

```
variable.function(element(s))
```

The list will be updated *in-place*.

In [10]:
```
numbers_list.append(42)
print(numbers_list)
```

```
[4, 8, 15, 16, 23, 42]
```

Perhaps we want to calculate the sum of the values in two lists. However, we cannot use the + like we did with single values. For list objects, the + will *combine* lists.

In [11]:
```
numbers_list
```

Out[11]:
```
[4, 8, 15, 16, 23, 42]
```

To perform mathematical operations, you can convert the above list to an array using the NumPy package.

Exercise 2:

1. Confirm that 'numbers_list+numbers_list' does not add list items element by element, but appends a copy of itself.
2. Try multiplying numbers_list by a number.
3. Show only the first 4 elements of numbers_list.
4. Using 'append', add your name to the list. Names are strings, so use quotes, i.e. "Bob". Does it work?

Solution:

In [12]:
```
numbers_list+numbers_list
```

Out[12]:
```
[4, 8, 15, 16, 23, 42, 4, 8, 15, 16, 23, 42]
```

In [13]:
```
numbers_list*3
```

Out[13]:
```
[4, 8, 15, 16, 23, 42, 4, 8, 15, 16, 23, 42, 4, 8, 15, 16, 23, 42]
```

In [15]:
```
numbers_list[0:4]
```

Out[15]:
```
[4, 8, 15, 16]
```

In [16]:
```
numbers_list.append("Lisa")
numbers_list
```

Out[16]:
```
[4, 8, 15, 16, 23, 42, 'Lisa']
```

If you successfully added your name to our 'numbers_list', then it won't be just numbers. Let's fix it before using it as a numerical array in the next section.

In [17]:
```
numbers_list = [4, 8, 15, 16, 23, 42]
```

Importing Packages

Packages are collection of modules, which help simplify common tasks. NumPy (https://numpy.org/) is useful for mathematical operations and array manipulation.

- Provides a high-performance multidimensional array object and tools for working with these arrays.
- Fundamental package for scientific computing with Python.
- Included with with the Anaconda package manager.
- For more examples than presented below, please refer the NumPy Quick Start (https://numpy.org/devdocs/user/quickstart.html)

The basic syntax for calling packages is to type the import [package name]. However, some packages have long names, so you can use import [package name] as [alias].

In [18]:
```
import numpy as np
```

If you do not see any error after running the line above, then the package was successfully imported.

Working with arrays

I can use NumPy's array constructor *np.array()* to convert our list to a NumPy array and perform the matrix multiplication. For example, I can double each element of the array:

In [19]:
```
numbers_array = np.array(numbers_list)
numbers_array*2
```

Out[19]:
```
array([ 8, 16, 30, 32, 46, 84])
```

Another difference between arrays and lists is that lists are only one-dimensional. NumPy can be any number of dimensions. For example, I can change the dimensions of the data using the *reshape()* function:

In [20]:
```
numbers_array_2d = numbers_array.reshape(3,2)
numbers_array_2d
```

Out[20]:
```
array([[ 4,  8],
       [15, 16],
       [23, 42]])
```

In [21]:
```
numbers_array_2d.shape
```

Out[21]:
```
(3, 2)
```

The original numbers_array has a length of 6, the new array has 2 rows and 3 columns.

Exercise 3:

1. Create a longer list, called 'long_list', by multiplying 'numbers_list' by 5.
2. Convert the list into a numpy array, called 'long_array'
3. Reshape into a 2D array.
4. Reshape into a 3D array.

Note: For 3 and 4, you will get errors unless the dimensions are compatible with the original array length. Read the error and try again.

Solution:

In [22]:
```
long_list = numbers_list*5
```

In [23]:
```
long_array = np.array(long_list)
```

In [24]:
```
long_array.reshape(3,10)
```

Out[24]:
```
array([[ 4,  8, 15, 16, 23, 42,  4,  8, 15, 16],
       [23, 42,  4,  8, 15, 16, 23, 42,  4,  8],
       [15, 16, 23, 42,  4,  8, 15, 16, 23, 42]])
```

In [25]:
```
long_array.reshape(3,2,5)
```

Out[25]:
```
array([[[ 4,  8, 15, 16, 23],
        [42,  4,  8, 15, 16]],

       [[23, 42,  4,  8, 15],
        [16, 23, 42,  4,  8]],

       [[15, 16, 23, 42,  4],
        [ 8, 15, 16, 23, 42]]])
```

If you are having troubles with the above exercise, make sure 'numbers_list' is set correctly, or reset it here:

In [26]:
```
numbers_list = [4, 8, 15, 16, 23, 42]
```

Reading ASCII data

The Pandas package has a useful function for reading text/ascii data called *read_csv()*. The function name is somewhat a misnomer, as *read_csv* will read any delimited data using the *delim=* keyword argument. Below, you will import the [Pandas (https://pandas.pydata.org/)](https://pandas.pydata.org/) package and we will read in a dataset. Note that the path below is relative to the current notebook and you may have to change the code if you are running locally on your computer:

data/VIIRSNDE_global2020258.v1.0.txt

We will look at the Visible Infrared Imaging Radiometer Suite (VIIRS) Active Fire product, a product that classifies if a pixel contains fire with various confidence levels. More information can be found at [https://www.ospo.noaa.gov/Products/land/fire.html (https://www.ospo.noaa.gov/Products/land/fire.html)](https://www.ospo.noaa.gov/Products/land/fire.html). We will examine the data on Sept 15, 2020 (day of year 258).

In [27]:
```python
import pandas as pd
```

The default seperator is a comma (,), however my data also contains space. I use the "\s*" to indicate space following the comma should be ignored. The engine="python" keyword ensures that this will work across different operating systems.

In [28]:
```python
fname = "data/VIIRSNDE_global2020258.v1.0.txt"
fires = pd.read_csv(fname, sep=',\s*', engine='python')
```

You can inspect the contents within the notebook using the *head()* function, which will return the first five rows of the dataset. Pandas automatically stores data in structures called *DataFrames*. DataFrames are two dimensional (rows and columns) and resemble a spreadsheet. The leftmost column is the row index and is not part of the *fires* dataset.

In [29]:
```python
fires.head()
```

Out[29]:

	Num	Lon	Lat	Mask	Conf	brt_t13(K)	frp(MW)	line	sample	YearDay	Time
0	2	29.991129	-29.555208	9	100	338.333923	29.883327	53	NDE	2020258	1
1	2	29.981384	-29.601839	7	17	300.099274	4.842572	60	NDE	2020258	1
2	2	30.085478	-29.868237	8	76	315.574402	10.423400	97	NDE	2020258	1
3	2	30.084040	-29.874882	8	53	310.038391	7.675260	98	NDE	2020258	1
4	2	30.082544	-29.881517	8	51	302.806458	5.290376	99	NDE	2020258	1

You can access individual columns of data using the column name. For example, below you can extract the pixel brightness temperature (brt):

In [30]:
```python
fires["brt_t13(K)"]
```

Out[30]:
```
0        338.333923
1        300.099274
2        315.574402
3        310.038391
4        302.806458
            ...
56303    305.149933
56304    300.437561
56305    305.149933
56306    307.136230
56307    302.398987
Name: brt_t13(K), Length: 56308, dtype: float64
```

Exercise 2: Import an ascii file

1. Import the dataset "20200901_20200930_Monterey.lev15.csv" and save it to a variable called *aeronet*.
2. Print the first few lines using *.head()*
3. Find a column that doesn't have only missing values (-999), and calculate the mean using the following syntax *variable["column"].mean()*

Solution:

In [31]:
```python
fname = "data/20200901_20200930_Monterey.lev15.csv"
aeronet = pd.read_csv(fname, sep=',\s*', engine='python')
```

In [32]:
```
aeronet.head()
```

Out[32]:

	Date(dd:mm:yyyy)	Time(hh:mm:ss)	Day_of_Year	Day_of_Year(Fraction)	AOD_1640nm	AOD_1020nm	AOD_870nm	AOD_865nm	AOD_779nm	AOD_675nm	...
0	0.071296	20:53:18	245	245.870347	0.061169	0.167012	0.238173	-999	-999	0.400838	...
1	0.071296	20:58:18	245	245.873819	0.061155	0.168417	0.239952	-999	-999	0.404648	...
2	0.071296	21:03:18	245	245.877292	0.063135	0.173143	0.246827	-999	-999	0.414668	...
3	0.071296	21:08:18	245	245.880764	0.061754	0.170541	0.241485	-999	-999	0.405998	...
4	0.071296	21:18:18	245	245.887708	0.059059	0.163919	0.232041	-999	-999	0.391191	...

5 rows × 113 columns

In [36]:
```
aeronet["AOD_1640nm"].mean()
```

Out[36]:
```
-1.7637041143410852
```

Working with masks and masked arrays

When working with data, sometimes there are numbers I want to remove. For instance, I may want to work with data below a certain threshold. You can subset the data using identity operations:

- less than: <
- less than or equal to: <=
- greater than: >
- greater than or equal to: >=
- equals: ==
- not equals: !=

Their use will return either a True or False statement. For the *fires* dataset, you can find which elements of the array that meet some condition, such as only examining larger fires that have a Fire Radiative Power (FRP) above 50 MW:

In [37]:
```
masked_nums = (fires['frp(MW)'] > 50)
print(masked_nums)
```

```
0        False
1        False
2        False
3        False
4        False
         ...
56303    False
56304    False
56305    False
56306    False
56307    False
Name: frp(MW), Length: 56308, dtype: bool
```

Sometimes you may want to filter by two conditions. For example, insteading of filtering the FRP data, you may only want to examine values within a latitude and longitude domain. In Python, I can combine multiple conditions using and (&) and or (|) statements. Below, I extract the data in 5°x5° box arond Monterey, California:

In [38]:
```
masked_nums = (fires['Lat'] > 35.0) & (fires['Lat'] < 40.0) & (fires['Lon'] > -125.0) & (fires['Lon'] < -120.0)
print(masked_nums)
```

```
0        False
1        False
2        False
3        False
4        False
         ...
56303    False
56304    False
56305    False
56306    False
56307    False
Length: 56308, dtype: bool
```

The above mask can be used in place of an index. Below, you can create a new variable that takes the FRP using the *fires['frp(MW)']* variable and subsets it with the array of *masked_nums*:

In [39]:

```
monterey_fires = fires['frp(MW)'][masked_nums]
print(monterey_fires)
```

```
16686     7.838871
16688    11.660147
16689    15.899877
16690    17.872414
16691    12.954104
           ...
55235    22.411970
55236    33.313660
55237    25.284723
55239    43.701473
55240    26.098984
Name: frp(MW), Length: 317, dtype: float64
```

From this new variable, you can compute the average in this region and compare them to the global average for that day:

In [40]:

```
monterey_fires.mean(), fires['frp(MW)'].mean()
```

Out[40]:

```
(91.59595084542588, 49.94695660808411)
```

You can use the size command to compare the dimensions of original array and the one that filtered out values that were outside of our latitude and longitude bounds. You will notice that these two arrays have different sizes.

In [41]:

```
fires['frp(MW)'].size, monterey_fires.size
```

Out[41]:

```
(56308, 317)
```

There are cases where you will want to preserve the size and shape of the original array. For these situations, you can utilize the NumPy *masked array* module. The syntax is *np.ma.array()*, and you will add a keyword argument *mask=*, which is set to the inverse (~) of the *mask_nums*.

In [42]:

```
monterey_fires_ma = np.ma.array(fires['frp(MW)'], mask=~masked_nums, fill_value=-999)
monterey_fires_ma
```

Out[42]:

```
masked_array(data=[--, --, --, ..., --, --, --],
             mask=[ True,  True,  True, ...,  True,  True,  True],
       fill_value=-999.0)
```

Then, you can calculate the mean values and confirm that they are the same as the previous example:

In [43]:

```
monterey_fires_ma.mean()
```

Out[43]:

```
91.59595084542588
```

However, the key difference will be the size, which retains the shape of the unmasked data:

In [44]:

```
monterey_fires_ma.size
```

Out[44]:

```
56308
```

Exercise 3: Filtering data

Using the dataset imported in the previous example (*aeronet*):

1. Create a mask that filters the "AOD_870nm" column to only include values that are above 0.
2. Create a new variables, *day_of_year*, with the mask applied to aeronet["Day_of_Year(Fraction)"].
3. Create a new variables, *aod_870*, with the mask applied to aeronet["AOD_870nm"].
4. Compare the mean value of *aeronet["AOD_870nm"]* to *aod_870*.
5. Why are they different?

Solution

In [45]:
```
masked_nums = (aeronet['AOD_870nm'] > 0)
print(masked_nums)
```

```
0        True
1        True
2        True
3        True
4        True
        ...
1027    False
1028     True
1029     True
1030     True
1031     True
Name: AOD_870nm, Length: 1032, dtype: bool
```

In [46]:
```
day_of_year = aeronet["Day_of_Year(Fraction)"][masked_nums]
print(day_of_year)
```

```
0       245.870347
1       245.873819
2       245.877292
3       245.880764
4       245.887708
           ...
1026    261.828287
1028    261.972998
1029    262.016840
1030    262.022951
1031    262.036343
Name: Day_of_Year(Fraction), Length: 1031, dtype: float64
```

In [47]:
```
aod_870 = aeronet["AOD_870nm"][masked_nums]
print(aod_870)
```

```
0       0.238173
1       0.239952
2       0.246827
3       0.241485
4       0.232041
          ...
1026    0.167687
1028    0.153517
1029    0.068082
1030    0.069707
1031    0.057226
Name: AOD_870nm, Length: 1031, dtype: float64
```

In [48]:
```
print(aeronet["AOD_870nm"].mean())
print(aod_870.mean())
```

```
-0.3344563769379846
0.6341813957322987
```

Why are they different?: aod_870 is more because we masked out the negative numbers.

Basic figures and plots

Python has several packages to create visuals for remote sensing data, either in the form of imagery or plots of relevant analysis. Of these, the most widely used and oldest packages is [Matplotlib (https://matplotlib.org/)](https://matplotlib.org/). Matplotlib plots are highly customizable and has additional toolkits that can enhance functionality, such as creating maps using the [Cartopy (https://scitools.org.uk/cartopy/docs/latest/)](https://scitools.org.uk/cartopy/docs/latest/) package, which I will describe more in the next session.

In [49]:
```
import matplotlib.pyplot as plt
```

Suppose you want to learn what the global distribution of fire radiative power is. From inspecting the frp(MW) column earlier, these values extend to many decimal places. Rather than use a continuous scale, I can instead group in the data into 10 MW bins, from 0 to 500 MW:

In [50]:
```
bins10MW = np.arange(0, 500, 10)
```

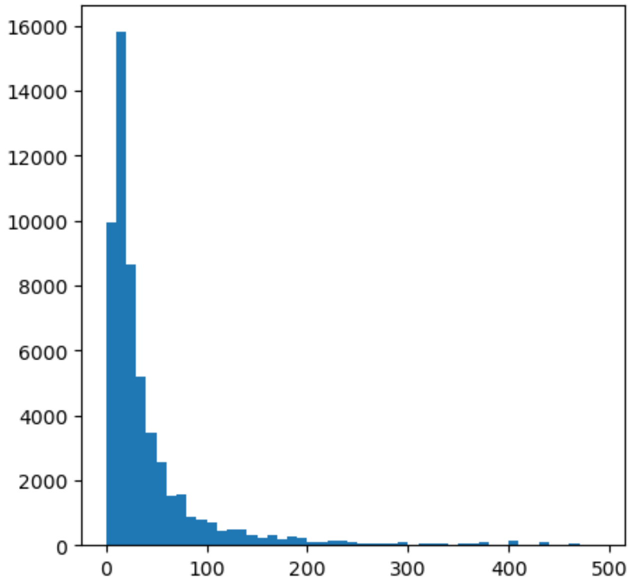
I can use these bins to create a histogram. Line by line, the code below will do as follows. Each additional line is layering elements on this empty graphic. The entire block of code must be run at once and not split into multiple cells.

1. *plt.figure()* creates a blank canvas.

2. I add the histogram to the figure using *plt.hist()*, which automatically will count the number of rows with fire radiative power in the bins that I defined above in the bins10W variable. I must then pass in the data (fires['frp(MW)']) and the bins (bins10MW) into plt.hist.
3. *plt.show()* tells matplotlib the plot is now complete and to render it:

In [51]:

```
plt.figure(figsize=[5,5])
plt.hist(fires['frp(MW)'], bins=bins10MW)
plt.show()
```

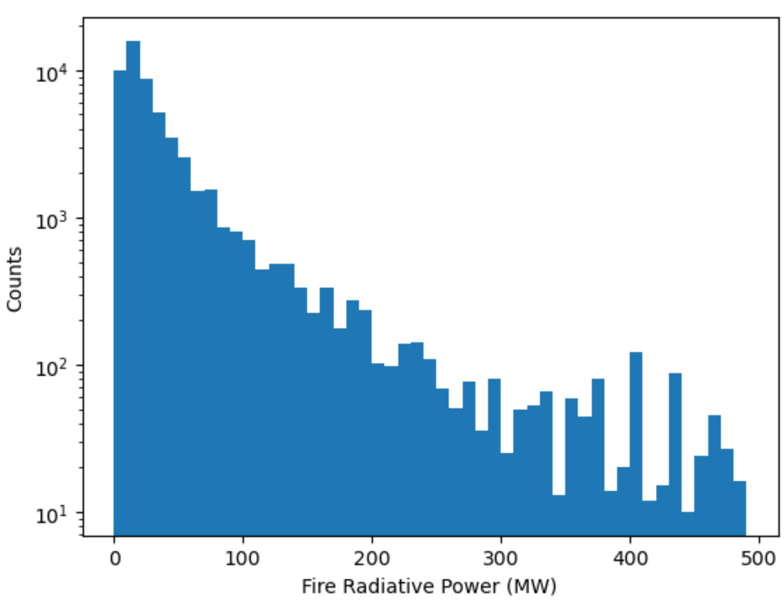


Below, you will remake this plot but add some aesthetic additions, such as labels to the x and y axis using *set_xlabel()* and *set_ylabel()*. Since there are thousands more fires with fire radiative power less than 100 MW than fires with higher values the data are likely lognormal. The plot will be easier to interpret of I rescale the y-axis to a log scale while leaving the x-axis linear.

The command *plt.subplot()* will return an axis object to a variable (*ax*). There are three numbers passed in (111), which correspond to rows, columns, and index. In this example, there is one row and one column, and therefore, only one index.

In [52]:

```
plt.figure()

ax = plt.subplot(111)

ax.hist(fires['frp(MW)'], bins=bins10MW)

ax.set_yscale('log')

ax.set_xlabel("Fire Radiative Power (MW)")
ax.set_ylabel("Counts")

plt.show()
```



You can also plot the data in 2-dimensions. For example, each row in *fires* has a latitude and longitude coordinates pair. I will take these two coordinates and plot using *plt.scatter()*. The first argument is the x-coordinate and the second is the y-coordinate (the order matters).

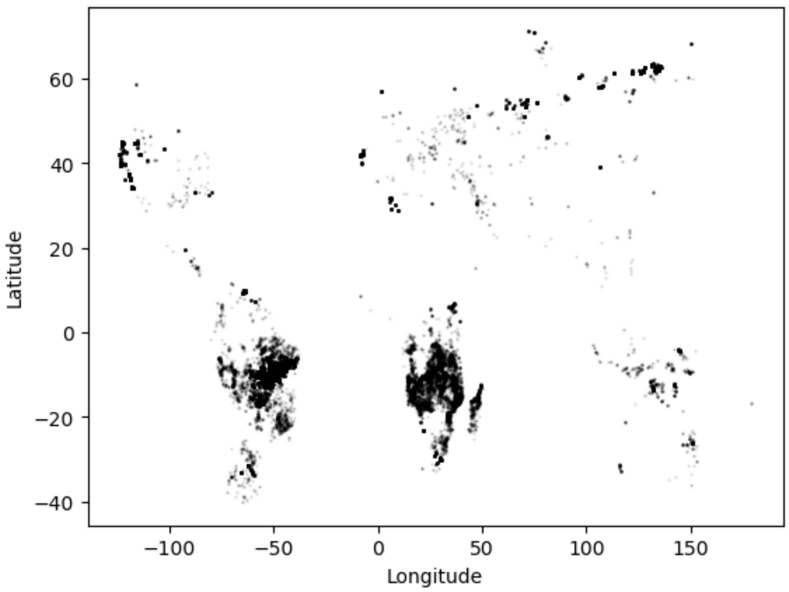
There are some command line options *plt.scatter()*:

- s: size with respect to the default
- c: color, which can be either from a predefined name list or a hexadecimal value
- alpha: opacity, where smaller values are transparent.

Like in the previous example, I have chosen to label the latitude and longitude axes:

In [54]:

```
fig = plt.figure()
ax = plt.subplot(111)

ax.scatter(fires['Lon'], fires['Lat'], s=0.5, c='black', alpha=0.1)

ax.set_xlabel('Longitude')
ax.set_ylabel('Latitude')

plt.show()
```



You can almost see the outline of the continents from the data above. In the next session, you will learn how to overlay maps onto your plots.

Exercise 4: Create a scatterplot

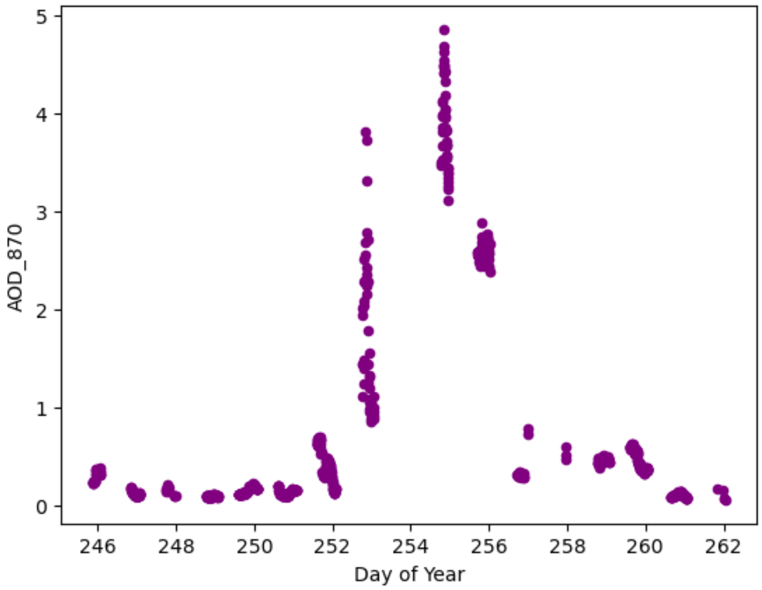
Use the variables *aod_870* and *day_of_year* that you made in Exercise 3 to:

1. Create a scatter plot showing the *day_of_year* (x-axis) and *aod_870* (y-axis)
2. Add y-axis and x-axis labels using *.set_xlabel()* and *.set_ylabel()*
3. Adjust the color and size of the scatterplot

Solution

In [55]:

```
fig = plt.figure()
ax = plt.subplot(111)

ax.scatter(day_of_year, aod_870, s=20, c='purple', alpha=1)

ax.set_xlabel('Day of Year')
ax.set_ylabel('AOD_870')

plt.show()
```



Summary:

You learned:

- Very basic built-in Python functions and operations
- How to import three packages: numpy, pandas, and matplotlib
- Worked with arrays and lists
- How to create a simple plot

Next lesson:

- More advanced plots, such as using maps
- Importing scientific datasets, such as netcdf and grib

Attribution

The content for this lab is taken directly from the AGU 2021 Python for Earth Sciences workshop, developed and led by Rebekah Esmaili (bekah@umd.edu (mailto:bekah@umd.edu)), Research Scientist, STC/JPSS. We are grateful for Rebekah's generous support of Open Science and sharing all her hard work!

Check out the original GitHub for the workshop, which also contains additional modules and great links to resources:

https://github.com/modern-tools-workshop/AGU-python-workshop-2021 (https://github.com/modern-tools-workshop/AGU-python-workshop-2021)