

Introduction to Python for Data Analysis

Brooke Luetgert, PhD

December 10, 2019

Welcome to the RCC workshop on Data Analysis in Python!

You will find all course materials at:

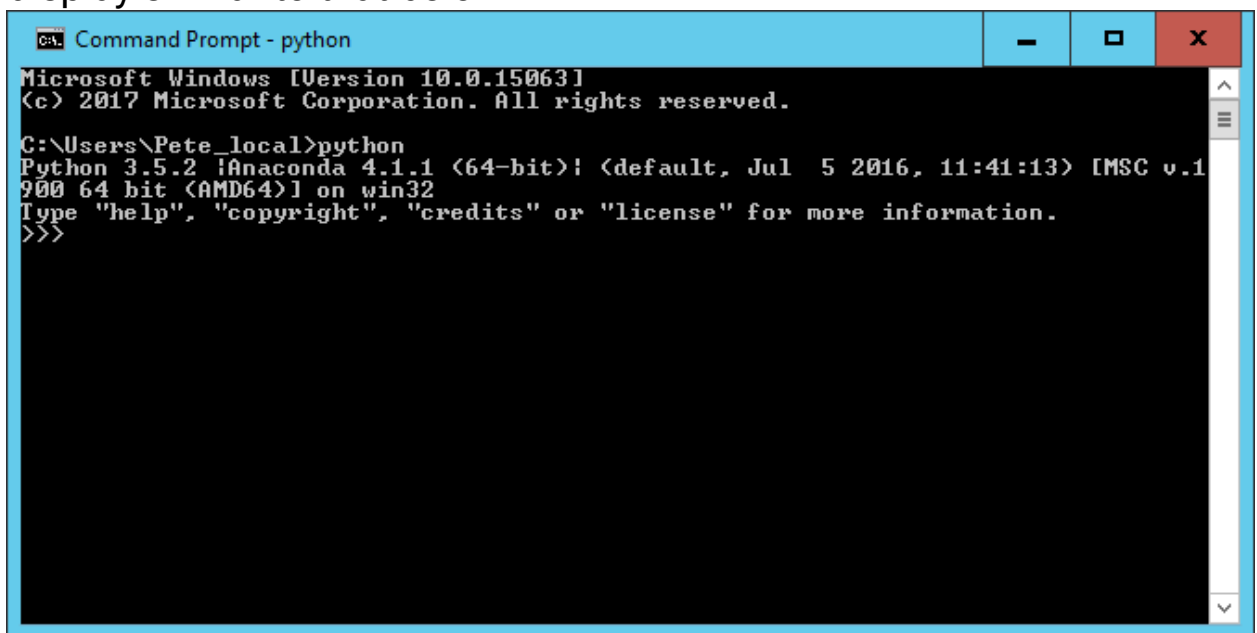
<https://github.com/luetgert/WorkshopMaterials>

To begin, we will download the Anaconda environment to make sure we're all on the same page. We will be using Python 3.7 and a number of additional packages that are essential to data analysis. To be clear, Python is the language that we are writing in. We will be using Jupyter Notebooks because it allows us to code in multiple languages, take notes as we work, view plots as we go and is accessible through any web browser.

Download the Anaconda for your operating system:

1. Go to the Anaconda website <https://www.anaconda.com/distribution> There are versions of Anaconda available for Windows, macOS, and Linux. The website will detect your operating system and provide a link to the appropriate download.
2. There will be two options, one for Python 2.x and another for Python 3.x. We will take the Python 3.x option. Python 2.x will eventually be phased out but is still provided for backward compatibility with some older optional Python modules. The majority of popular modules have been converted to work with Python 3.x. The actual value of x will vary depending on when you download. At the time of writing I am being offered Python 3.6 or Python 2.7.

3. For Windows and Linux there is the option of either a 64 bit (default) download or a 32 bit download. Unless you know that you have an old 32 bit pc you should choose the 64 bit installer.
4. Run the downloaded installer program. Accept the default settings until you are given the option to add Anaconda to your environmental Path variable. Despite the recommendation not to and the subsequent warning, you should select this option. This will make it easier later on to start Jupyter notebooks from any location.
5. The installation can take a few minutes. When finished you should be able to open a cmd prompt (Type cmd from Windows start and into the cmd window type python. You should get a display similar to that below.



```
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. All rights reserved.

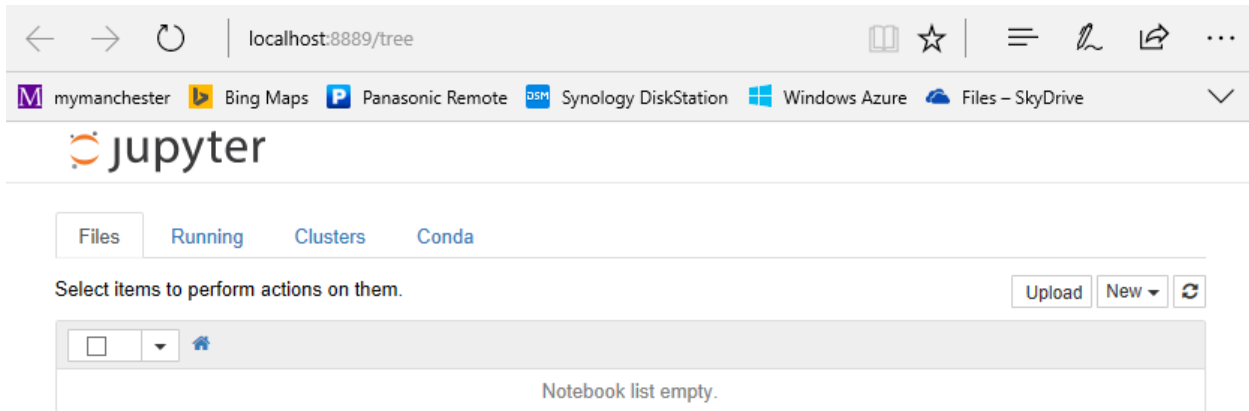
C:\Users\Pete_local>python
Python 3.5.2 |Anaconda 4.1.1 (64-bit)| (default, Jul 5 2016, 11:41:13) [MSC v.1
900 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

- 6.
7. The `>>>` prompt tells you that you are in the Python environment. You can exit Python with the `exit()` command.

Running Jupyter Notebooks in Windows

1. From file explorer navigate to where you can select the folder which contains your Jupyter Notebook notebooks (it can be empty initially).

2. Hold down the **shift** key and right-click the mouse
3. The pop-up menu items will include an option to start a cmd window or in the latest Windows release start a 'PowerShell' window. Select whichever appears.
4. When the window opens, type the command **jupyter notebook**.
5. Several messages will appear in the command window. In addition your default web browser will open and display the Jupyter notebook home page. The main part of this is a file browser window starting at the folder you selected in step 1.
6. There may be existing notebooks which you can select and open in a new tab in your browser or there is a menu option to create a new notebook.



7. The Jupyter package creates a small web services and opens your browser pointing at it. If your browser does not open, you can open it manually and specify 'localhost:8888' as the URL.
8. Port 8888 is the default port used by the Jupyter web service, but if it is already in use it will increment the port number automatically. Either way the port number it does use is given in a message in the cmd/powershell window.
9. Once running, the cmd/powershell window will display additional messages, e.g. about saving notebooks, but there is no need to

interact with it directly. The window can be minimized and ignored.

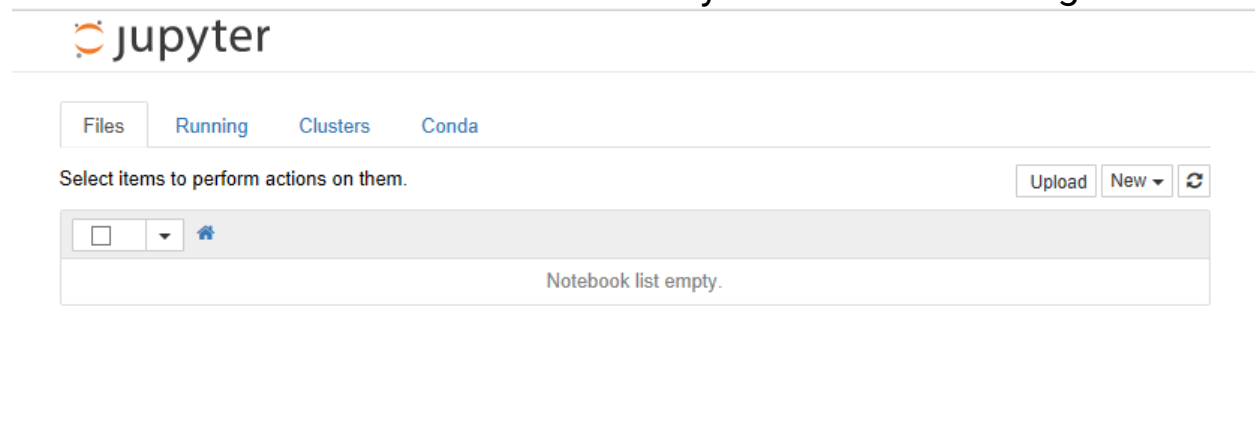
10. To shut Jupyter down, select the cmd/powershell window and type **Ctrl+c** twice and then close the window.

Introducing Jupyter notebooks

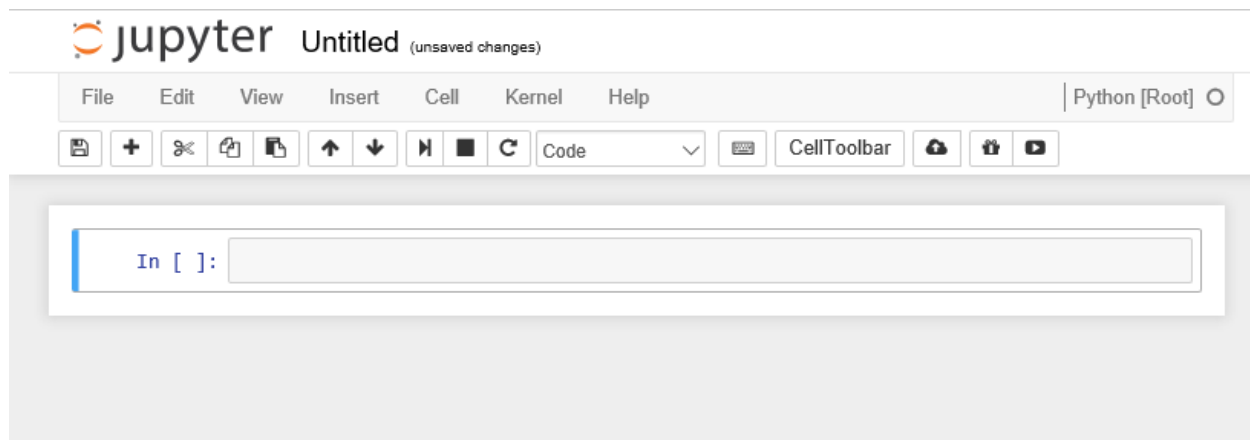
Jupyter originates from IPython, an effort to make Python development more interactive. Since its inception, the scope of the project has expanded to include **Julia**, **Python**, and **R**, so the name was changed to “Jupyter” as a reference to these core languages. Today, Jupyter supports even more languages, but we will be using it only for Python code. Specifically, we will be using **Jupyter notebooks**, which allows us to easily take notes about our analysis and view plots within the same document where we code. This facilitates sharing and reproducibility of analyses, and the notebook interface is easily accessible through any web browser. Jupyter notebooks are started from the terminal using

```
$ jupyter notebook
```

Your browser should start automatically and look something like this:



When you create a notebook from the New option, the new notebook will be displayed in a new browser tab and look like this.

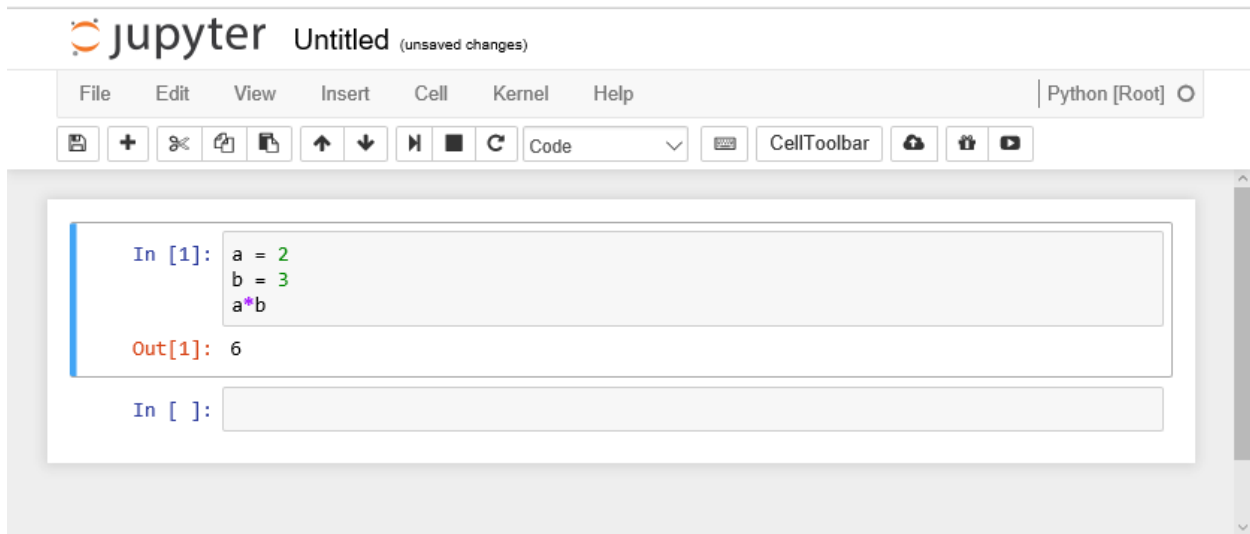


Initially the notebook has no name other than 'Untitled'. If you click on 'Untitled' you will be given the option of changing the name to whatever you want.

The notebook is divided into **cells**. Initially there will be a single input cell marked by `In []:`.

You can type Python code directly into the cell. You can split the code across several lines as needed. Unlike working in the command line, the code is not interpreted line by line. To interpret the code in a cell, you can click the Run button in the toolbar or from the Cell menu option, or use keyboard shortcuts (e.g., **Shift+Return**). All of the code in that cell will then be executed.

The results are shown in a separate `Out [1]:` cell immediately below. A new input cell (`In []:`) is created for you automatically.



When a cell is run, it is given a number along with the corresponding output cell. If you have a notebook with many cells in it you can run the cells in any order and also run the same cell many times. The number on the left hand side of the input cells increments, so you can always tell the order in which they were run. For example, a cell marked `In [5]:` was the fifth cell run in the sequence.


Although there is an option to do so on the toolbar, you do not have to manually save the notebook. This is done automatically by the Jupyter system.

Not only are the contents of the `In []:` cells saved, but so are the `Out []:` cells. This allows you to create complete documents with both your code and the output of the code in a single place. You can also change the cell type from Python code to Markdown using the `Cell > Cell Type` option. **Markdown** is a simple formatting system which allows you to create documentation for your code, again all within the same notebook structure.

The notebook itself is stored as specially-formatted text file with an `.ipynb` extension. These files can be opened and run by others with

Jupyter installed. This allows you to share your code inputs, outputs, and Markdown documentation with others. You can also export the notebook to HTML, PDF, and many other formats to make sharing even easier.

New cells

From the insert menu item you can insert a new cell anywhere in the notebook either above or below the current cell. You can also use the  button on the toolbar to insert a new cell below.

Change cell type

By default new cells are created as code cells. From the cell menu item you can change the type of a cell from code to markdown. Markdown is a markup language for formatting text, it has much of the power of HTML, but is specifically designed to be human-readable as well. You can use Markdown cells to insert formatted textual explanation and analysis into your notebook. For more information about Markdown, check out these resources:

- [Jupyter Notebook Markdown Docs](#)
- [Markdown - a Visual Guide](#)
- [Mastering Markdown from Github](#)
- [Markdown official open source project](#)

Hiding output

When you run cells of code the output is displayed immediately below the cell. In general this is convenient. The output is associated with the cell that produced it and remains a part of the notebook. So if you copy or move the notebook the output stays with the code.

However lots of output can make the notebook look cluttered and more difficult to move around. So there is an option available from the `cell` menu item to 'toggle' or 'clear' the output associated either with an individual cell or all cells in the notebook.

Creating variables and assigning values

Variables and Types

In Python variables are created when you first assign values to them.

```
a = 2
b = 3.142
```

All variables have a data type associated with them. The data type is an indication of the type of data contained in a variable. If you want to know the type of a variable you can use the built-in `type()` function.

```
print(type(a))
print(type(b))
s = "Hello World"
print(type(s))
<class 'int'>
<class 'float'>
<class 'str'>
```

Arithmetic operations

For now we will stick with the numeric types and do some arithmetic. All of the usual arithmetic operators are available.

In the examples below we also introduce the Python comment symbol `#`. Anything to the right of the `#` symbol is treated as a comment. To a large extent using Markdown cells in a notebook reduces the need for comments in the code in a notebook, but occasionally they can be useful.

We also make use of the built-in `print()` function, which displays formatted text.

```
print("a =", a, "and b =", b)
print(a + b)    # addition
print(a * b)    # multiplication
print(a - b)    # subtraction
print(a / b)    # division
print(b ** a)   # exponentiation
print(2 * a % b) # modulus - returns the remainder
a = 2 and b = 3.142
5.1419999999999995
6.284
-1.142
0.6365372374283896
9.872164
0.8580000000000001
```

We need to use the `print()` function because by default only the last output from a cell is displayed in the output cell.

In our example above, we pass four different parameters to the first call of `print()`, each separated by a comma. A string `"a = "`, followed by the variable `a`, followed by the string `"b = "` and then the variable `b`.

The output is what you would probably have guessed at.

All of the other calls to `print()` are only passed a single parameter. Although it may look like 2 or 3, the expressions are evaluated first and it is only the single result which is seen as the parameter value and printed.

In the last expression `a` is multiplied by 2 and then the modulus of the result is taken. Had I wanted to calculate `a % b` and then multiply the result by two I could have done so by using brackets to make the order of calculation clear.

Arithmetic expressions can be arbitrarily complex, but remember people have to read and understand them as well.

Exercise

1. Create a new cell and paste into it the assignments to the variables `a` and `b` and the contents of the code cell above with all of the print statements. Remove all of the calls to the print function so you only have the expressions that were to be printed and run the code. What is returned?
2. Now remove all but the first line (with the 4 items in it) and run the cell again. How does this output differ from when we used the print function?
3. Practice assigning values to variables using as many different operators as you can think of.
4. Create some expressions to be evaluated using parentheses to enforce the order of mathematical operations that you require

Using built-in functions

Python has a reasonable number of built-in functions. You can find a complete list in the [official documentation](#).

Additional functions are provided by 3rd party packages which we will look at later on.

For any function, a common question to ask is: What parameters does this function take?

In order to answer this from Jupyter, you can type the function name and then type `shift+tab` and a pop-up window will provide you with various details about the function including the parameters.

Exercise

For the `print()` function find out what parameters can be provided

Getting Help for Python

You can get help on any Python function by using the help function. It takes a single parameter of the function name for which you want the help

```
help(print)
```

Help on built-in function print in module builtins:

```
print(...)  
    print(value, ..., sep=' ', end='\n', file=sys.stdout,  
flush=False)
```

Prints the values to a stream, or to sys.stdout by default.

Optional keyword arguments:

file: a file-like object (stream); defaults to the current sys.stdout.

sep: string inserted between values, default a space.

end: string appended after the last value, default a newline.

flush: whether to forcibly flush the stream.

There is a great deal of Python help and information as well as code examples available from the internet. One popular site is [stackoverflow](#) which specializes in providing programming help. They have dedicated forums not only for Python but also for many of the popular 3rd party Python packages. They also always provide code examples to illustrate answers to questions.

You can also get answers to your queries by simply inputting your question (or selected keywords) into any search engine.

****A couple of things you may need to be wary of:** There are currently 2 versions of Python in use, in most cases code examples will run in

either, but there are some exceptions. Secondly, some replies may assume a knowledge of Python beyond your own, making the answers difficult to follow. But for any given question there will be a whole range of suggested solutions so you can always move on to the next.

Working with Pandas

Now we need data. If you are not familiar, Kaggle is a data analysis competitions website. Kaggle is a great place to find fun, new data. We can start with looking at [Avocado Prices](#). You will need to login/create an account to use Kaggle, but if you prefer not to, the data are also hosted here: [Avocado Prices](#).

Unzip the file using whatever you use to zip/unzip things, and you're left with a CSV file. CSV files are common file types for data analysis because the structure is meant to be organized by columns and rows, where the values are separated by commas and the rows are separated by new lines in the document. So, let's read this csv in with Pandas.

Create a working directory for our Python script and data and title it as you wish. It is recommended to create one folder per project. You may wish to do this in a directory or on the desktop. I will be doing the latter, but feel free to do as you wish. We have a file called avocado.csv and we want to use pandas to load that .csv into a workable dataframe.

Within Jupyter notebooks, we open our project in the same file folder as our data. In the first box, we write:

```
import pandas as pd  # convention to import and use  
pandas like this
```

```
df = pd.read_csv("datasets/avocado.csv") # df
stands for dataframe. Also a common convention to
call this df
```

A dataframe is a type of pandas object that is basically a "table" like object with columns and rows, which we can also perform various calculations and statistical operations..etc on. We can print it out:
df

18249 rows A-- 14 columns

Okay, that's a bit messy to print that out everytime. Often, we just want to see a small snippet of our dataframe just to make sure everything is what we expect. Most people will use the `.head()` method for this:
df.head()

You can pass a parameter to the head, which is how many rows you want. Like
df.head(3)

	Unnamed: 0	Date	AveragePrice	Total Volume	4046	4225	4770	Total Bags	Small Bags	Large Bags	XLarge Bags	type	year	region
0	0	2015-12-27	1.33	64236.62	1036.74	54454.85	48.16	8696.87	8603.62	93.25	0.0	conventional	2015	Albany
1	1	2015-12-20	1.35	54876.98	674.28	44638.81	58.33	9505.56	9408.07	97.49	0.0	conventional	2015	Albany
2	2	2015-12-13	0.93	118220.22	794.70	109149.67	130.50	8145.35	8042.21	103.14	0.0	conventional	2015	Albany

If, instead, you want to see the end. You can do that too with `.tail()`

```
df.tail(6)
```

We can also reference specific columns, like:

```
df['AveragePrice'].head()
0      1.33
1      1.35
2      0.93
3      1.08
4      1.28
Name: AveragePrice, dtype: float64
```

Also, you can use attribute-like dot notation like:

```
df.AveragePrice.head()
0      1.33
1      1.35
2      0.93
3      1.08
4      1.28
Name: AveragePrice, dtype: float64
```

A common goal with data analysis is to visualize data. We all love pretty graphs, plus they help us generalize data usually pretty well. So, how might we graph this data. Looking at the data, it's clear that it's actually organized by date, but also region, so we could plot line graphs of individual regions over time.

Next, how might we get an individual region? We'd need to filter for that region column! Let's see how we might do that:

```
albany_df = df[df['region']=="Albany"]
```

Ok, so that might look a bit dense, but let's parse that out.

```
albany_df = df[ df['region'] == "Albany" ]
```

We're just saying that the `albany_df` is the `df`, where the `df['region']` column is equal to Albany. The result is a new dataframe where this is the case:

```
albany_df.head()
```

Okay, so one more thing you will often see is dataframes are "indexed" by something. Let's see what this dataframe is indexed by:

```
albany_df.index
```

```
Int64Index([0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
            ...17612], dtype='int64', length=338)
```

In this case, the index is worthless to us. It's just incrementing row counts, which we have no use for here. Instead, we should ask ourselves, how is this Albany avocado data organized? How does each row relate to the other? Well, by date. That's the main way this data is organized. So really, we want Date to be our index! We can do this with `set_index`.

```
albany_df.set_index("Date")
```

338 rows A-- 13 columns

Wait, what? Why did it print out like that? Some of the methods in pandas will modify your dataframe `in place`, but MOST are going to simply do the thing and return a new dataframe. So if we just check real quick:

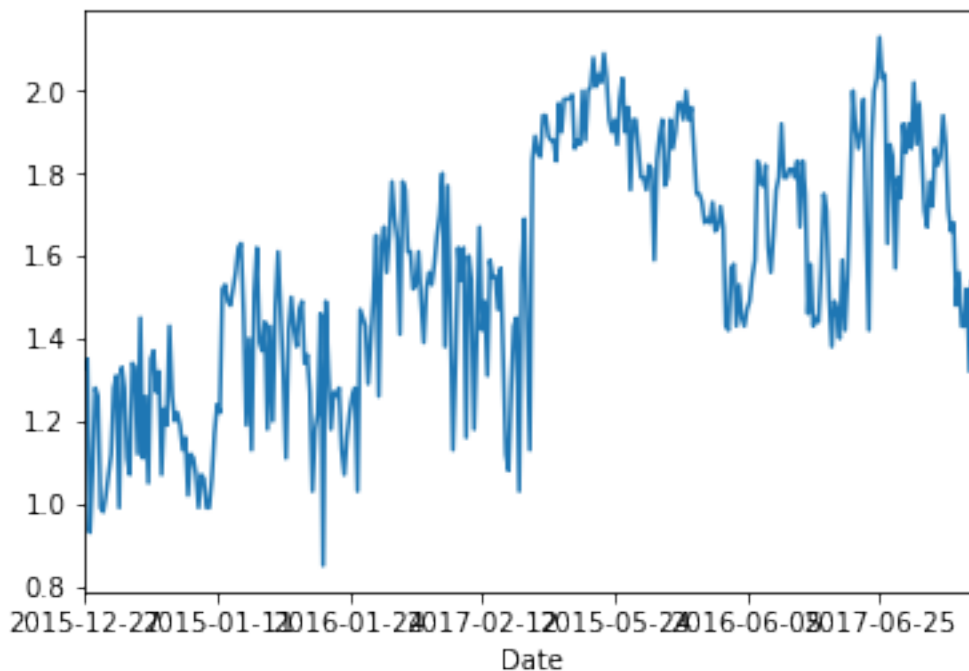
```
albany_df.head()
```

We can see that the `albany_df` is not impacted. There are two ways we can handle for this. One is to re-define:

```
albany_df = albany_df.set_index("Date")
albany_df.head()
```

The other option we can use is the `inplace` parameter. Something like:

```
albany_df.set_index("Date", inplace=True)  
would also work. Okay, now that we've done that, let's plot!  
albany_df[ 'AveragePrice' ].plot()
```



When we call `.plot()` on a dataframe, it is just assumed that the x axis will be your index, and then Y will be all of your columns, which is why we specified one column in particular.

This graph is a bit messy, however, especially with the dates, which also look out of order and such.

Welcome to a Matplotlib

Matplotlib is capable of creating most kinds of charts, like line graphs, scatter plots, bar charts, pie charts, stack plots, 3D graphs, and geographic map graphs.


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

This line imports the integral pyplot, which we're going to use throughout this entire series. We import pyplot as plt, and this is a traditional standard for python programs using pyplot.

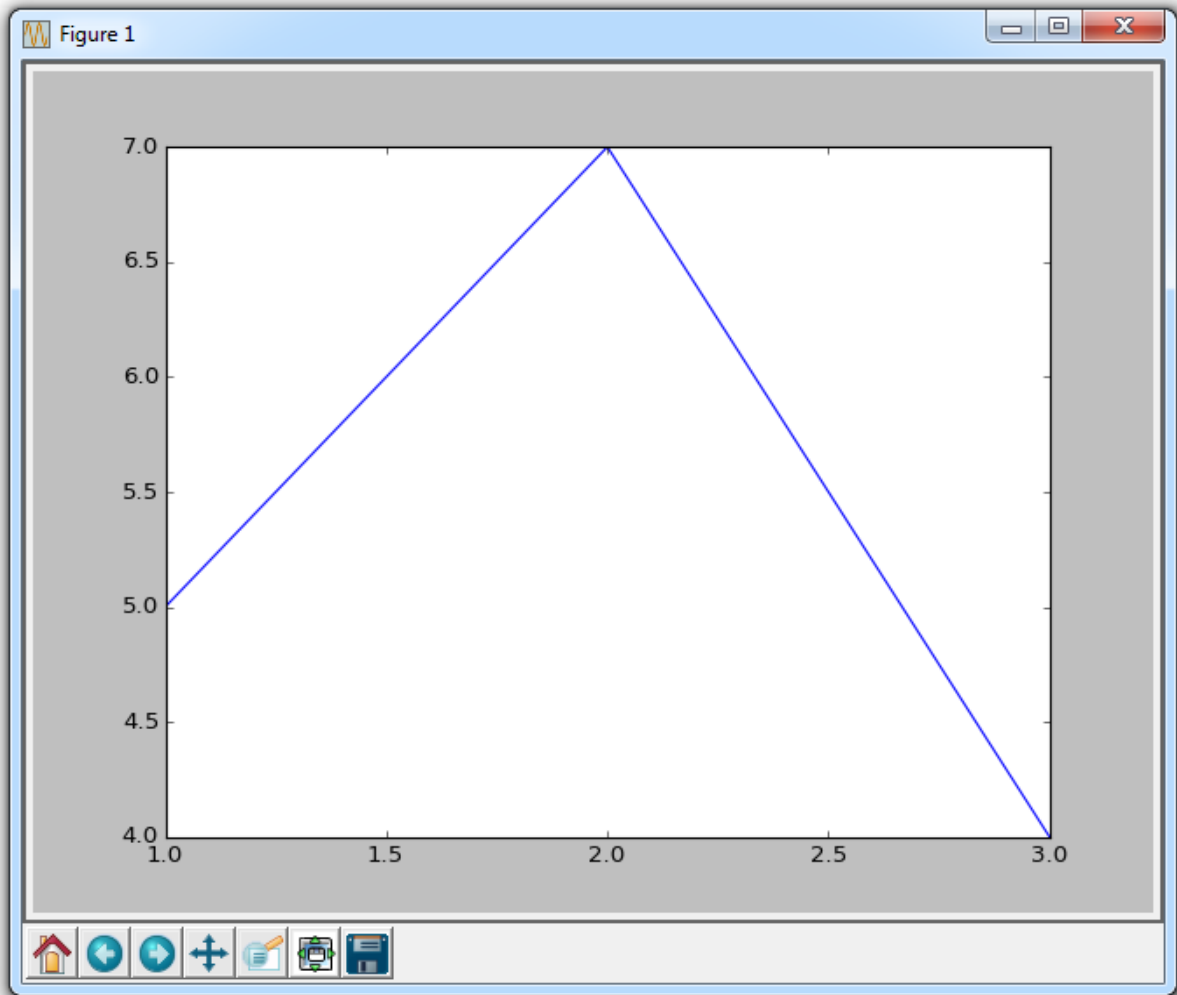
```
plt.plot([1,2,3],[5,7,4])
```

Next, we invoke the .plot method of pyplot to plot some coordinates. This .plot takes many parameters, but the first two here are 'x' and 'y' coordinates, which we've placed lists into. This means, we have 3 coordinates according to these lists: 1,5 2,7 and 3,4.

The plt.plot will "draw" this plot in the background, but we need to bring it to the screen when we're ready, after graphing everything we intend to.

```
plt.show()
```

With that, the graph should pop up. If not, sometimes it can pop under, or you may have gotten an error. Your graph should look like:



Now we need to cover legends, titles, and labels within Matplotlib. Graphs may be self-explanatory, but having a title to the graph, labels on the axis, and a legend that explains what each line is always good practice.

To start:

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

```
x = [1,2,3]
```

```
y = [5,7,4]
```

```
x2 = [1,2,3]
```

```
y2 = [10,14,12]
```

This way, we have two lines that we can plot. Next:

```
plt.plot(x, y, label='First Line')
```

```
plt.plot(x2, y2, label='Second Line')
```

Here, we plot as we've seen already, only this time we add another parameter "label." This allows us to assign a name to the line, which we can later show in the legend. The rest of our code:

```
plt.xlabel('Plot Number')
```

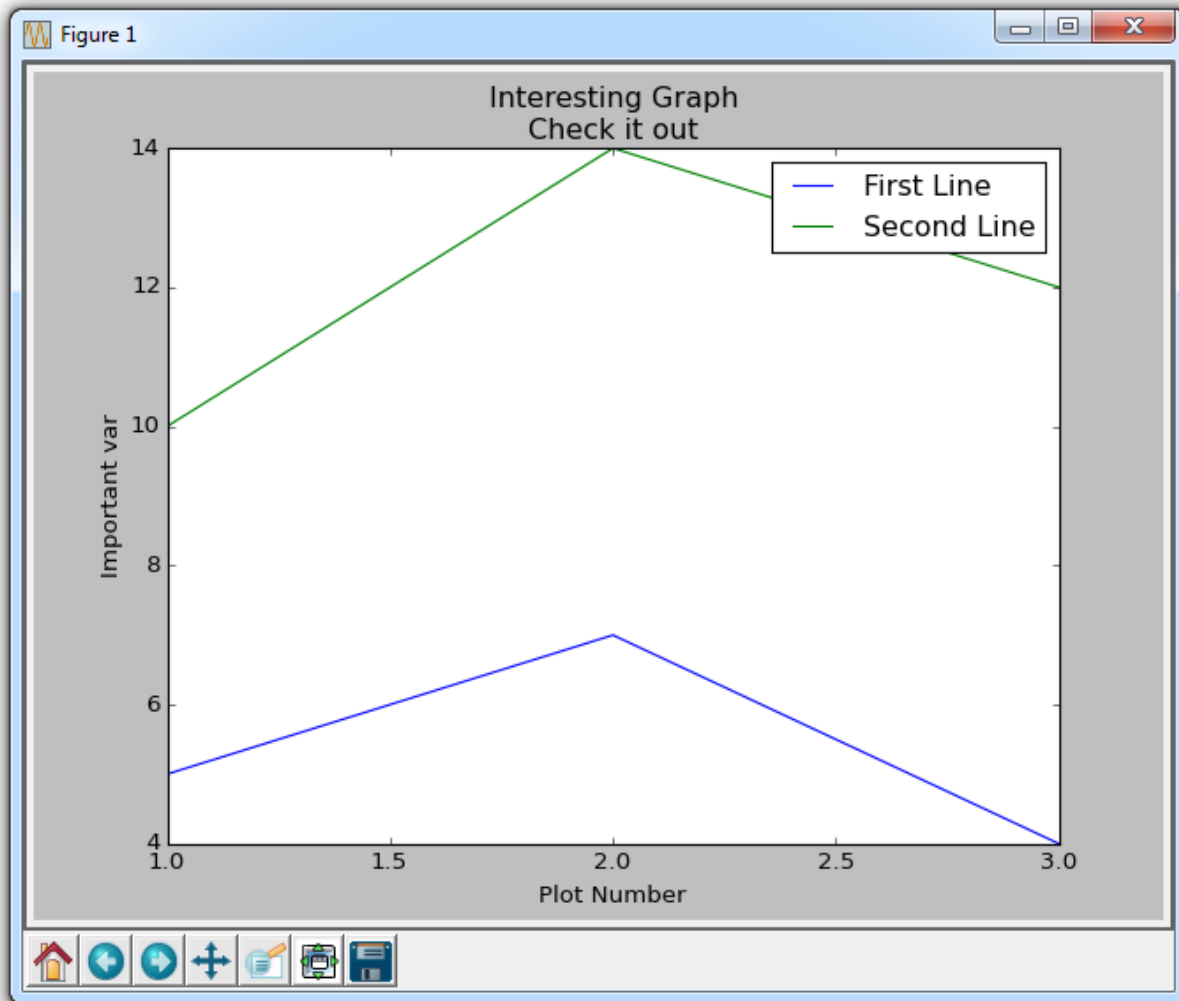
```
plt.ylabel('Important var')
```

```
plt.title('Interesting Graph\nCheck it out')
```

```
plt.legend()
```

```
plt.show()
```

With `plt.xlabel` and `plt.ylabel`, we can assign labels to those respective axis. Next, we can assign the plot's title with `plt.title`, and then we can invoke the default legend with `plt.legend()`. The resulting graph:



We will now cover bar charts and histograms with Matplotlib. Let's start with a bar chart.

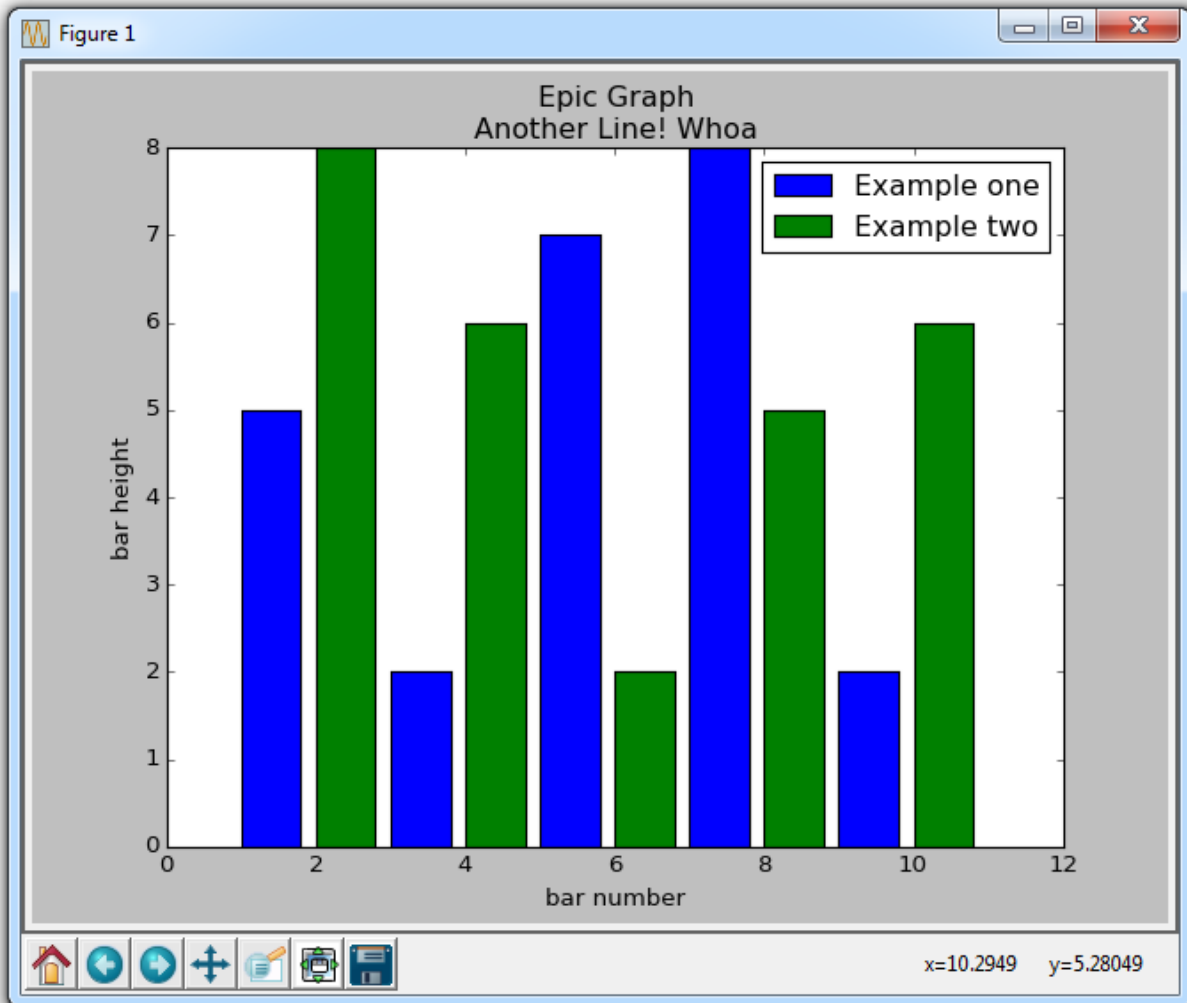
```
import matplotlib.pyplot as plt
    plt.bar([1,3,5,7,9],[5,2,7,8,2],
label="Example one")

plt.bar([2,4,6,8,10],[8,6,2,5,6], label="Example
two", color='green')
plt.legend()
plt.xlabel('bar number')
plt.ylabel('bar height')

plt.title('Epic Graph\nAnother Line! Whoa')

plt.show()
plt.savefig("barplot")
```

The `plt.bar` creates the bar chart for us. If you do not explicitly choose a color, then, despite doing multiple plots, all bars will look the same. You can use color to color just about any kind of plot, using colors like g for green, b for blue, r for red, and so on. You can also use hex color codes, like #191970



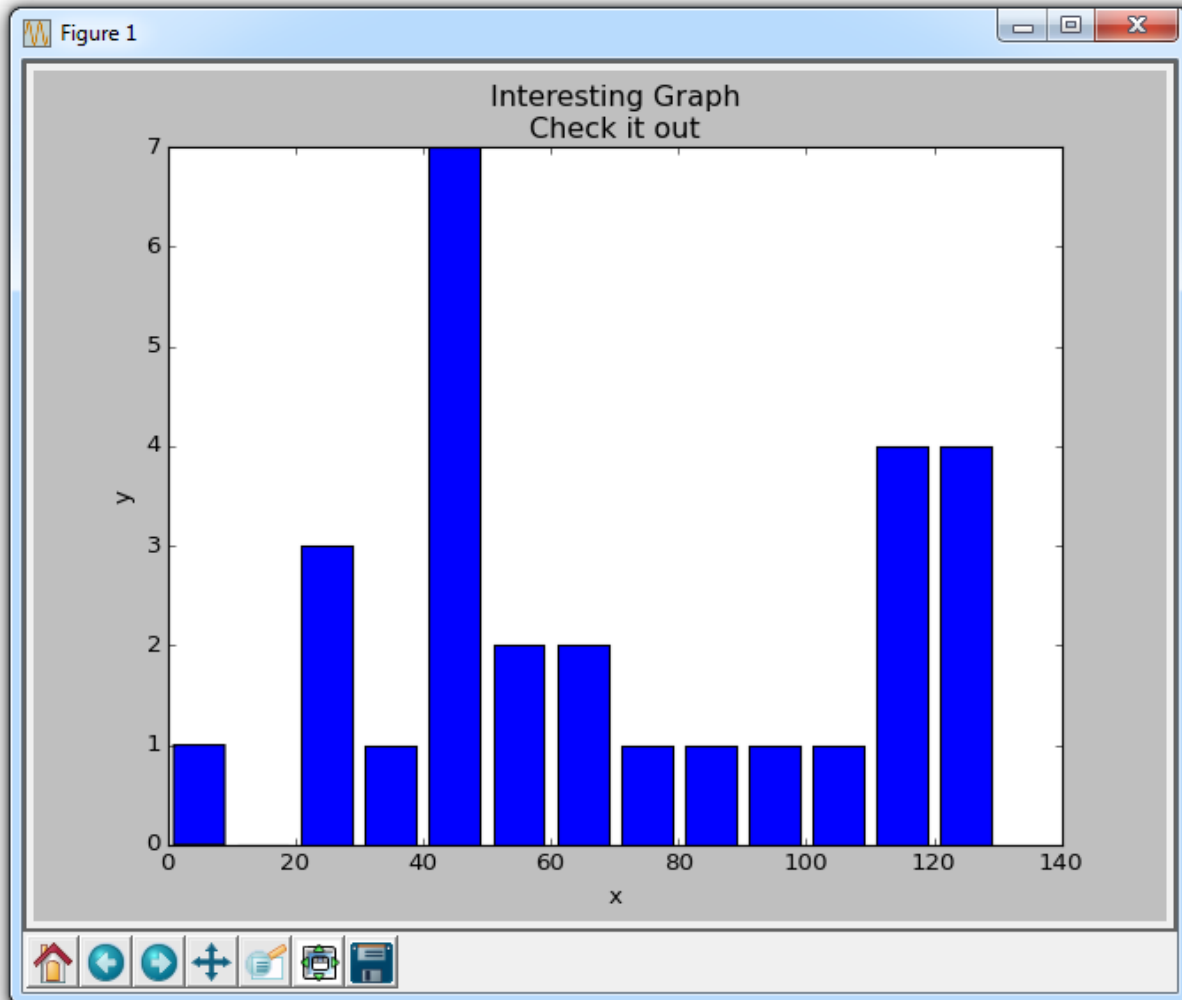
Next, we can cover histograms. Very much like a bar chart, histograms tend to show distribution by grouping segments together. Examples of this might be age groups, or scores on a test. Rather than showing every single age a group might be, maybe you just show people from 20-25, 25-30... and so on. Here's an example:

```
import matplotlib.pyplot as plt

population_ages =
[22, 55, 62, 45, 21, 22, 34, 42, 42, 4, 99, 102, 110, 120, 121, 12
2, 130, 111, 115, 112, 80, 75, 65, 54, 44, 43, 42, 48]
```

```
bins =  
[0,10,20,30,40,50,60,70,80,90,100,110,120,130]  
  
plt.hist(population_ages, bins, histtype='bar',  
rwidth=0.8)  
  
plt.xlabel('x')  
plt.ylabel('y')  
plt.title('Interesting Graph\nCheck it out')  
plt.legend()  
plt.show()
```

The resulting graph is:



For `plt.hist`, you first put in all of the values, then you specify into what "bins" or containers you will place the data into. In our case, we are plotting a bunch of ages, and we want to display them in terms of increments of 10 years. We give the bars a width of 0.8, but you can choose something else if you want to make the bars thicker, or thinner.

Now, let's look at scatter plots! The idea of scatter plots is usually to compare two variables, or three if you are plotting in 3 dimensions, looking for correlation or groups.

Some sample code for a scatter plot:

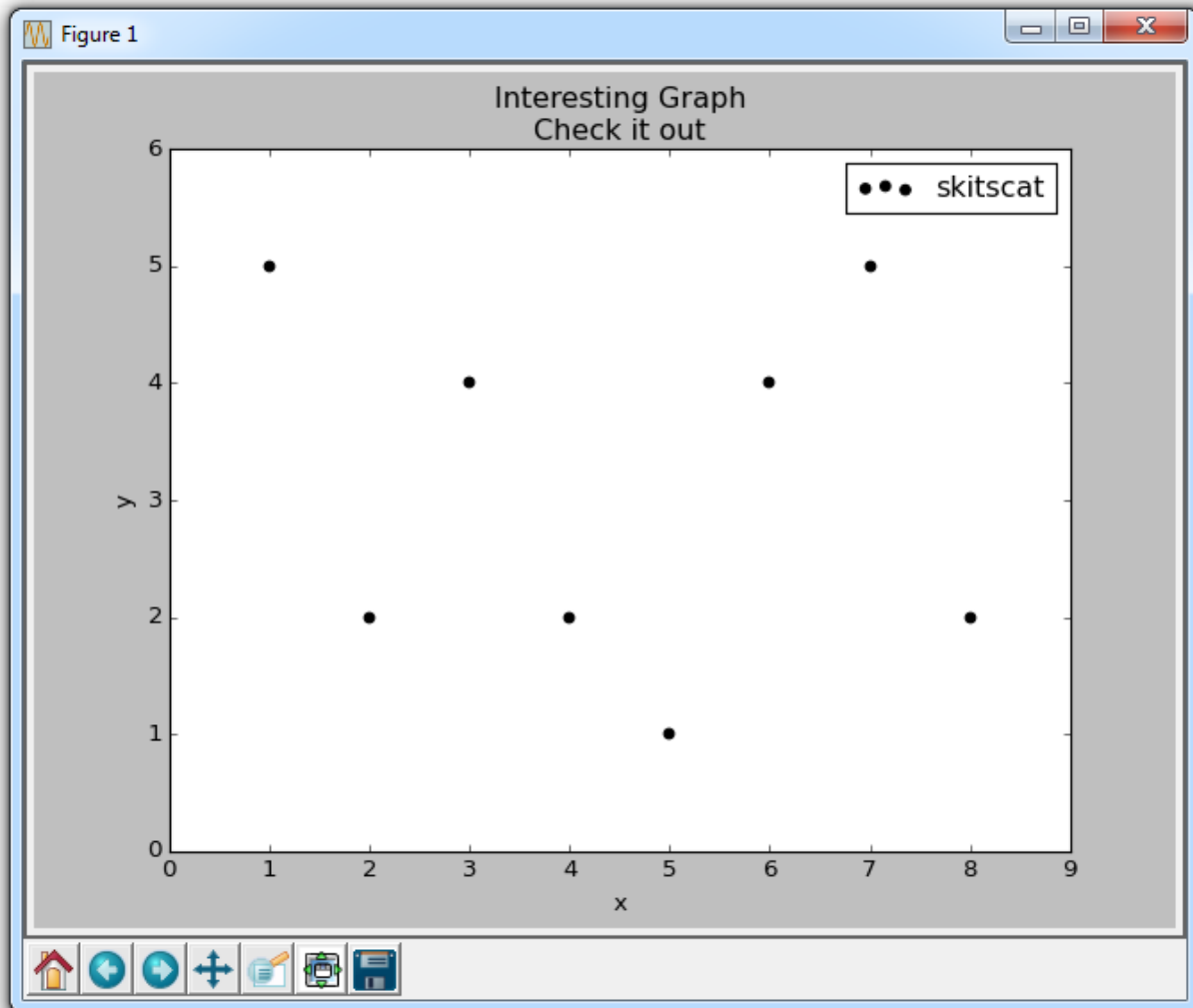
```
import matplotlib.pyplot as plt

x = [1,2,3,4,5,6,7,8]
y = [5,2,4,2,1,4,5,2]

plt.scatter(x,y, label='skitscat', color='black',
s=25, marker="o")

plt.xlabel('x')
plt.ylabel('y')
plt.title('Interesting Graph\nCheck it out')
plt.legend()
plt.show()
```

The result:



The `plt.scatter` allows us to not only plot on x and y , but it also lets us decide on the color, size, and type of marker we use.

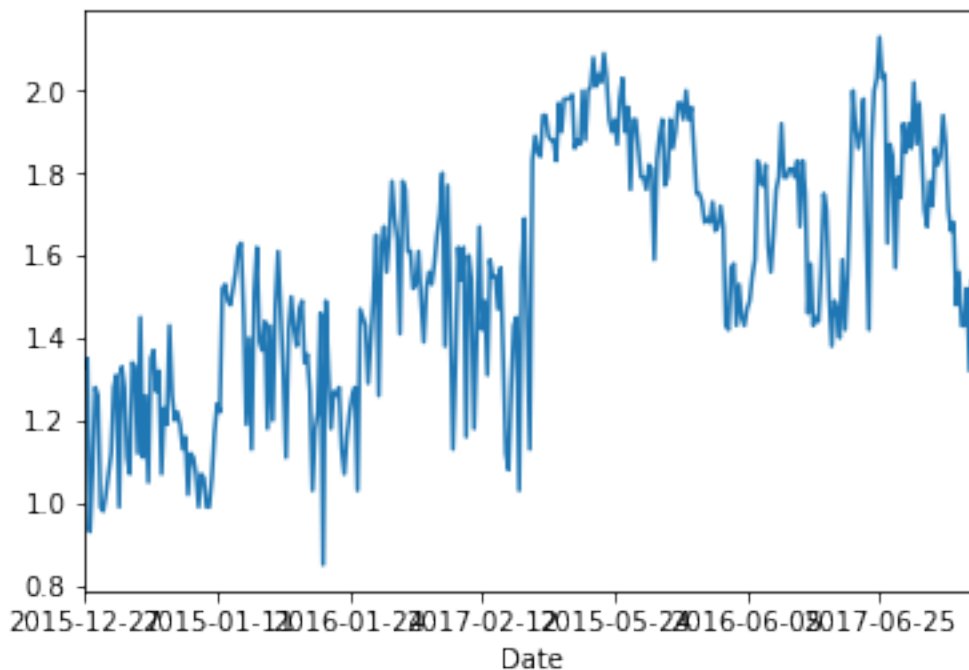
Let's return to our Avacado price data. Where we left off, we were graphing the price from Albany over time, but it was quite messy. Here's a recap:

```
import pandas as pd

df = pd.read_csv("datasets/avocado.csv")

albany_df = df[df['region']=="Albany"]
albany_df.set_index("Date", inplace=True)

albany_df["AveragePrice"].plot()
```

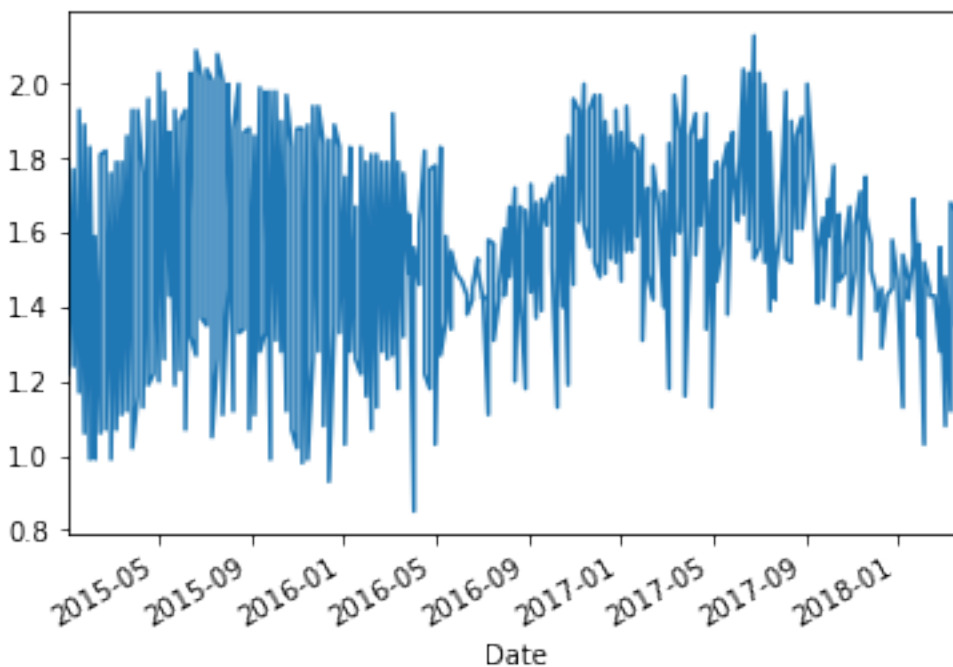


So dates are funky types of data, since they are strings, but also have order, at least to us. When it comes to dates, we have to help computers out a bit. Luckily for us, Pandas comes built in with ways to handle for dates. First, we need to convert the date column to datetime objects:

```
df = pd.read_csv("datasets/avocado.csv")
```

```
df['Date'] = pd.to_datetime(df['Date'])
albany_df = df[df['region']=="Albany"]
albany_df.set_index("Date", inplace=True)
```

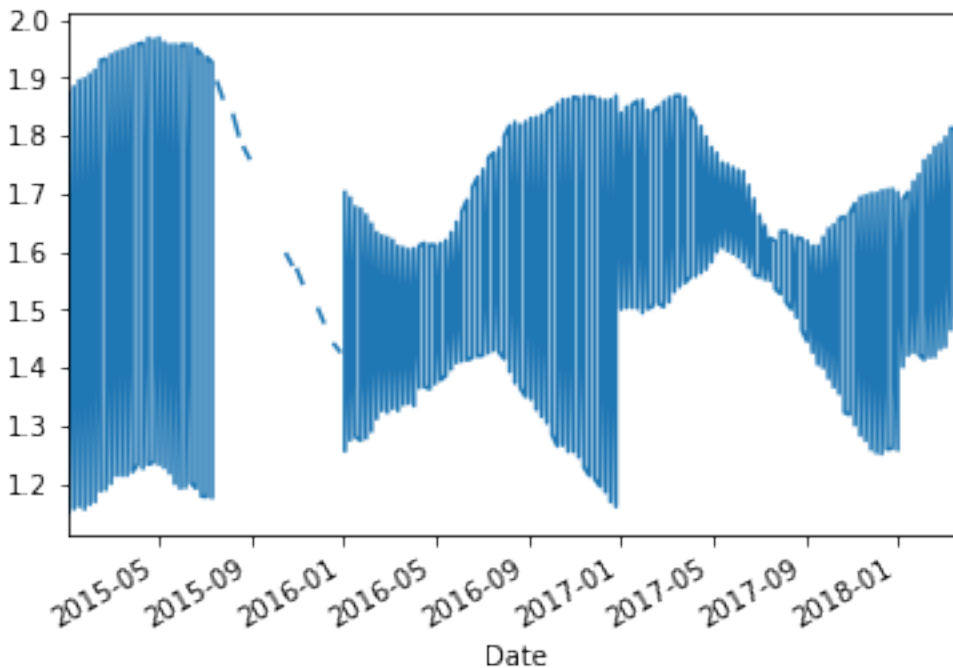
```
albany_df["AveragePrice"].plot()
<matplotlib.axes._subplots.AxesSubplot at
0x11fa86828>
```



Alright, the formatting looks better in terms of axis, but that graph is pretty wild! Could we settle it down a bit? We could smooth the data with a rolling average.

To do this, let's make a new column, and apply some smoothing:

```
albany_df["AveragePrice"].rolling(25).mean().plot()
<matplotlib.axes._subplots.AxesSubplot at
0x1223cc278>
```



Hmm, so what happened? Pandas understands that a date is a date, and to sort the X axis, but I am now wondering if the dataframe itself is sorted. If it's not, that would seriously screw up our moving average calculations. This data may be indexed by date, but is it sorted? Let's see.

```
albany_df.sort_index(inplace=True)
```

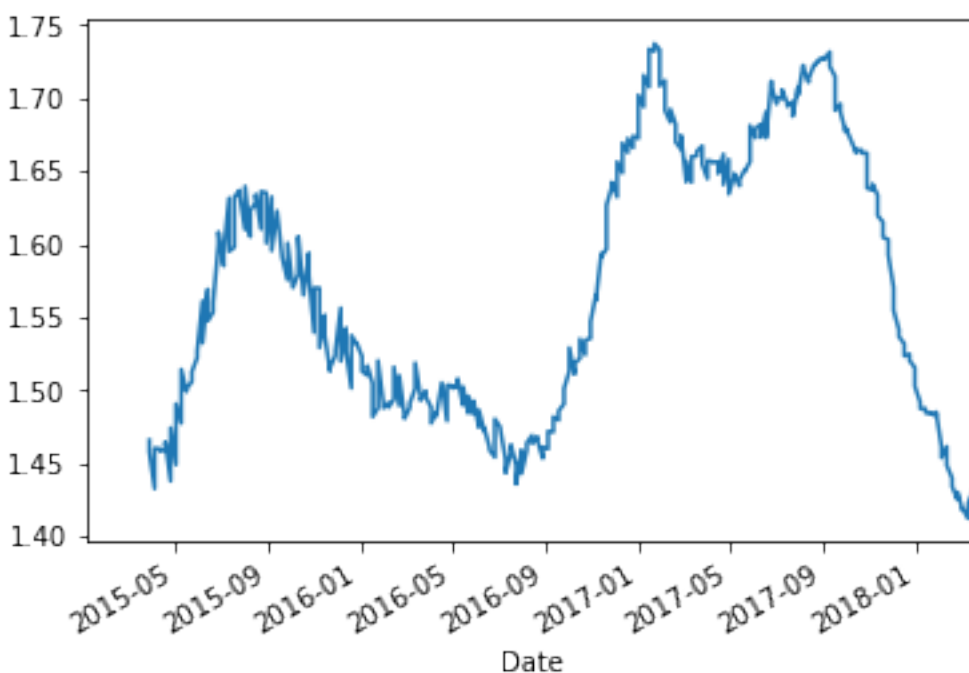
```
/Library/Frameworks/Python.framework/Versions/3.7/
lib/python3.7/site-packages/ipykernel_launcher.py:
1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice
from a DataFrame
```

See the caveats in the documentation: <http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy>

```
"""Entry point for launching an IPython kernel.
```

What's this warning above? Should we be worried? Basically, all it's telling us is that we're doing operations on a copy of a slice of a dataframe, and to watch out because we might not be modifying what we were hoping to modify (like the main df). In this case, we're not trying to work with the main dataframe. It's just a warning, not an error.

```
albany_df["AveragePrice"].rolling(25).mean().plot()  
<matplotlib.axes._subplots.AxesSubplot at  
0x1223ccf98>
```



And there we have it! A more useful summary of avocado prices for Albany over the years.

Visualizations are cool, but what if we want to save our new, smoother, data like above? We can give it a new column in our dataframe:

```
albany_df["price25ma"] =  
albany_df["AveragePrice"].rolling(25).mean()
```

```
/Library/Frameworks/Python.framework/Versions/3.7/
lib/python3.7/site-packages/ipykernel_launcher.py:
1: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice
from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value
instead
```

See the caveats in the documentation: <http://pandas.pydata.org/pandas-docs/stable/indexing.html#indexing-view-versus-copy>

```
"""Entry point for launching an IPython kernel.
albany_df.head()
```

That warning is annoying. What could we do? Let's be explicit and let Pandas know that we are making a copy...

```
albany_df = df.copy()[df['region']=="Albany"]
albany_df.set_index('Date', inplace=True)
albany_df["price25ma"] =
albany_df["AveragePrice"].rolling(25).mean()
```

Another subtle thing you might have glossed over is the requirement for us to sort things how we intend before we start performing operations. Many times, you won't be visualizing columns before you make them. You may actually never visualize them. Imagine if we wrote the above code before we sorted by date, basically just assuming things were ordered by date. We'd have produced bad data. It's very easy to make mistakes like this. Check your code early and check it often through printing it out and visualizing it where possible! Alright, let's make more graphs.

Let's graph prices in the different regions. We hard-coded the Albany region, but hmm, we don't know all of the regions. What do we do?! If

we just wanted to get a "list" from one of our columns, we could reference just that column, like:

```
df['region']
0          Albany
1          Albany
2          Albany
3          Albany
4          Albany
5          Albany
6          Albany
7          Albany
8          Albany
9          Albany
10         Albany
11         Albany
12         Albany
13         Albany
14         Albany
15         Albany
16         Albany
17         Albany
18         Albany
19         Albany
20         Albany
21         Albany
22         Albany
23         Albany
24         Albany
25         Albany
26         Albany
27         Albany
28         Albany
29         Albany
...
18219      TotalUS
```

```

18220          TotalUS
18221          TotalUS
18222          TotalUS
18223          TotalUS
18224          TotalUS
18225          West
18226          West
18227          West
18228          West
18229          West
18230          West
18231          West
18232          West
18233          West
18234          West
18235          West
18236          West
18237  WestTexNewMexico
18238  WestTexNewMexico
18239  WestTexNewMexico
18240  WestTexNewMexico
18241  WestTexNewMexico
18242  WestTexNewMexico
18243  WestTexNewMexico
18244  WestTexNewMexico
18245  WestTexNewMexico
18246  WestTexNewMexico
18247  WestTexNewMexico
18248  WestTexNewMexico
Name: region, Length: 18249, dtype: object

```

Then convert to array with:

```

df['region'].values
array(['Albany', 'Albany', 'Albany', ...,
      'WestTexNewMexico',

```



```
        'WestTexNewMexico', 'WestTexNewMexico'],  
dtype=object)
```

You could go to list like:

```
df['region'].values.tolist()
```

and then maybe get the uniques with:

```
print(set(df['region'].values.tolist()))
```

```
{'Orlando', 'RaleighGreensboro', 'Charlotte',  
'NewOrleansMobile', 'Syracuse', 'Nashville',  
'DallasFtWorth', 'Chicago', 'Columbus',  
'SanFrancisco', 'Southeast', 'Tampa',  
'Jacksonville', 'SanDiego', 'MiamiFtLauderdale',  
'Seattle', 'Philadelphia', 'California',  
'SouthCentral', 'Pittsburgh', 'GrandRapids',  
'Atlanta', 'Indianapolis', 'CincinnatiDayton',  
'RichmondNorfolk', 'Louisville', 'Roanoke',  
'LasVegas', 'Northeast', 'NorthernNewEngland',  
'Detroit', 'Portland', 'Plains', 'Spokane',  
'LosAngeles', 'HarrisburgScranton',  
'SouthCarolina', 'TotalUS', 'West', 'Albany',  
'NewYork', 'WestTexNewMexico', 'BuffaloRochester',  
'Sacramento', 'BaltimoreWashington', 'Boston',  
'Boise', 'Denver', 'HartfordSpringfield',  
'PhoenixTucson', 'Houston', 'GreatLakes',  
'Midsouth', 'StLouis'}
```

So, very quickly you can take your pandas data and get it out into array/list form and use your own knowledge of python. Or, you could also just use some Pandas method. Just know, if you're trying to do it, it probably has a method!

```
df['region'].unique()
```

```

array(['Albany', 'Atlanta', 'BaltimoreWashington',
      'Boise', 'Boston',
        'BuffaloRochester', 'California',
      'Charlotte', 'Chicago',
        'CincinnatiDayton', 'Columbus',
      'DallasFtWorth', 'Denver',
        'Detroit', 'GrandRapids', 'GreatLakes',
      'HarrisburgScranton',
        'HartfordSpringfield', 'Houston',
      'Indianapolis', 'Jacksonville',
        'LasVegas', 'LosAngeles', 'Louisville',
      'MiamiFtLauderdale',
        'Midsouth', 'Nashville', 'NewOrleansMobile',
      'NewYork',
        'Northeast', 'NorthernNewEngland',
      'Orlando', 'Philadelphia',
        'PhoenixTucson', 'Pittsburgh', 'Plains',
      'Portland',
        'RaleighGreensboro', 'RichmondNorfolk',
      'Roanoke', 'Sacramento',
        'SanDiego', 'SanFrancisco', 'Seattle',
      'SouthCarolina',
        'SouthCentral', 'Southeast', 'Spokane',
      'StLouis', 'Syracuse',
        'Tampa', 'TotalUS', 'West',
      'WestTexNewMexico'], dtype=object)

```

That was quick and painless!

```
graph_df = pd.DataFrame()
```

```

for region in df['region'].unique()[16]:
    print(region)
    region_df = df.copy()[df['region']==region]
    region_df.set_index('Date', inplace=True)
    region_df.sort_index(inplace=True)

```

```

region_df[f"{region}_price25ma"] =
region_df["AveragePrice"].rolling(25).mean()

if graph_df.empty:
    graph_df = region_df[[f"{region}
_price25ma"]] # note the double square brackets!
else:
    graph_df =
graph_df.join(region_df[f"{region}_price25ma"])

```

```

Albany
Atlanta
BaltimoreWashington
Boise
Boston
BuffaloRochester
California
Charlotte
Chicago
CincinnatiDayton
Columbus
DallasFtWorth
Denver
Detroit
GrandRapids
GreatLakes

```

I set the limit to 16 just to show we're getting bogged down. What is going on? Things are taking exponentially longer and longer. We are having memory issues and massive slow down. Let's look again at our dates. They are being duplicated. For example:

```
graph_df.tail()
```

Each row should be a separate date, but it's not. Through some debugging, we can discover what's happening, which actually informs us as to why our previous data looked so ugly.

Our avocados have multiple prices: Organic and Conventional! So, let's pick one. I'll go with organic. So, we'll just start over!

```
import pandas as pd

df = pd.read_csv("datasets/avocado.csv")
df = df.copy()[df['type']=='organic']

df["Date"] = pd.to_datetime(df["Date"])

df.sort_values(by="Date", ascending=True,
inplace=True)
df.head()
```

Now, let's just copy and paste the code from above, minus the print:

```
graph_df = pd.DataFrame()

for region in df['region'].unique():
    region_df = df.copy()[df['region']==region]
    region_df.set_index('Date', inplace=True)
    region_df.sort_index(inplace=True)
    region_df[f"{region}_price25ma"] =
region_df["AveragePrice"].rolling(25).mean()

    if graph_df.empty:
        graph_df = region_df[[f"{region}
_price25ma"]] # note the double square brackets!
(so df rather than series)
    else:
        graph_df =
graph_df.join(region_df[f"{region}_price25ma"])
```

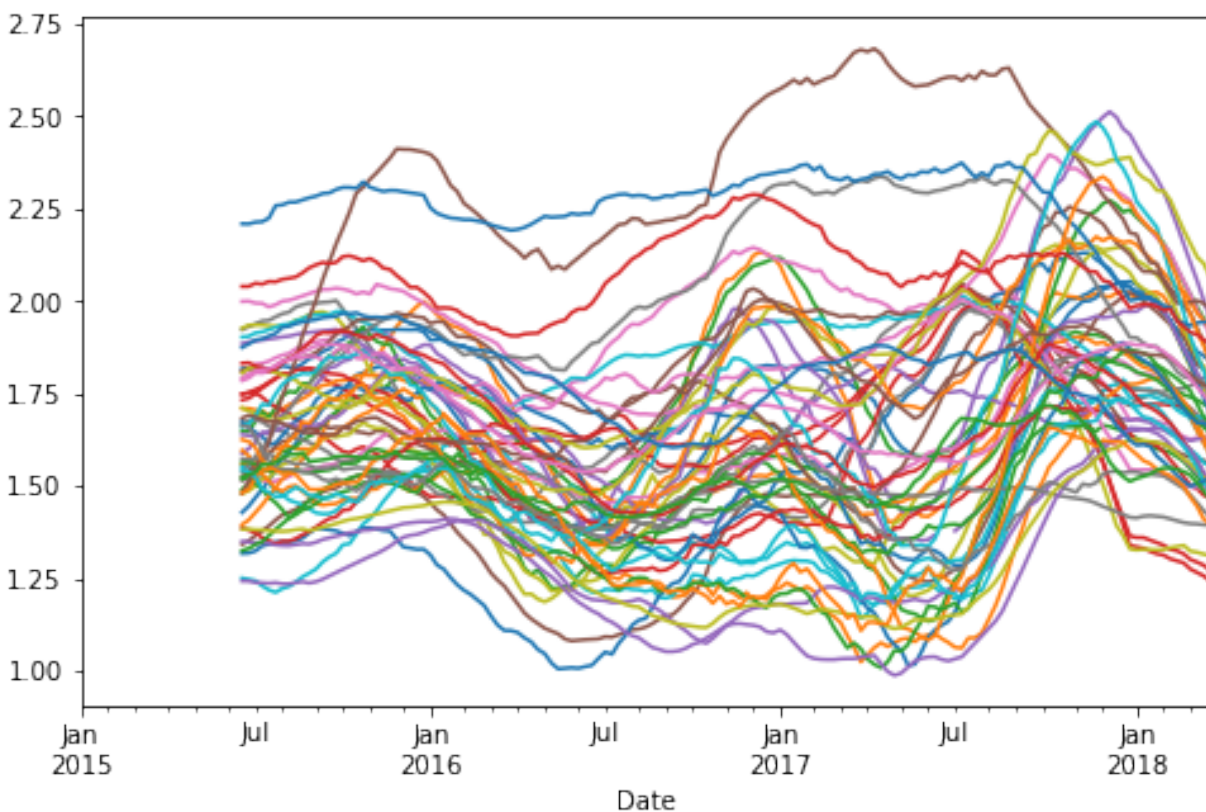
```
graph_df.tail()
```

5 rows A-- 54 columns

Now it's quick! Awesome!

Let's graph!

```
graph_df.plot(figsize=(8,5), legend=False)  
<matplotlib.axes._subplots.AxesSubplot at  
0x124046fd0>
```



Again, we could add a title, label our axes and add a legend, but this is an excellent point to break for your continued work and experimentation. Happy coding!