Synoptic Meteorology I: Assignment #5

Due: 23 October 2018

Learning Objectives: to learn how to use the matplotlib pyplot routines to plot data in Python; to learn how to create and execute a Python script.

Through the matplotlib module's pyplot routines, Python offers a robust way to create high-quality data visualizations. In this assignment, you will learn the basics of creating a simply plot in Python. The material found in this assignment largely follows that in matplotlib's own pyplot tutorial, and I recommend working through it on your own as time permits:

https://matplotlib.org/tutorials/introductory/pyplot.html

Beginning with this assignment, we will use the text editor in the left panel of the Spyder window. Begin by creating a new script by clicking on the left-most button on the Spyder toolbar. Save this blank script to your Desktop or home directory with File -> Save As…, then navigate to the desired folder (if not already there). Call your script something like assignment5.py, then save it. Spyder has a nice feature that automatically saves your script each time you run the code, so unless you've gone a long time without running your code, you need not worry about frequently saving it.

As with all Python modules, we must first load the necessary module:

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

Importing this module and renaming it to plt is a *de facto* Python convention that helps to shorten the code while not sacrificing the descriptive character of the module's name.

Python treats most things as objects, and in this example both a figure and its axes are no different. To begin to create our plot, we must first set up figure and axes objects. We do so using the pyplot `subplots` function:

```
fig, ax = plt.subplots(figsize=(16,9))
```

The `subplots` function is powerful in that it can be called to create a figure with only one panel or it can be called to create multipanel (1 x 2, 2 x 1, 2 x 2, and so on) figure. In this example, we are using it to create a one-panel figure. Inside of the function call, we pass it what is known as a *keyword argument*, here telling Python to create a plot that is 16" wide by 9" tall. The result of this function call is two returned objects, one for the figure as a whole (`fig`) and one for its axes (`ax`). Most of the time, we'll refer to the axes object when working with our data.

Now, we need some data to plot! There are many ways in which we can generate a 'fake' data set to plot. Perhaps the simplest is to use a random number generator to make up a specified number of data points. The Python numpy module has several such generators, the most flexible of which might be the `numpy.random.normal()` function. First, let's load the numpy module:

```
import numpy as np
```

Next, let's generate some data!

```
data = np.random.normal(loc=10,scale=2,size=50)
```

This function call generates a 1-D array of 50 elements, each of which is drawn randomly from a normal distribution (or bell curve) that has a mean value of 10 and a standard deviation (related to the distribution's width) of 2.

It's pretty easy to create a very basic plot of these data and show it:

```
ax.plot(data)
fig.show()
```

Here, we tell Python to plot the data in `data` on the axes defined before, then to show the figure. Because we don't pass in any information about the *x*-axis values associated with `data`, Python defaults to an *x*-axis of 0 to 49 (by 1) that corresponds to the indices of the `data` array. Go ahead and run your script once you've added the lines above.

You will find that your data are a bit noisy, as you'd expect for randomly generated data, but that the values are primarily 10 ± 2. Note that you'll get a different plot each time you run the script – a hallmark of using a random number generator! Regardless, we can use a couple of numpy math functions to check on the data:

```
print(np.mean(data), np.std(data))
```

This will print the mean of `data` followed by the standard deviation of `data`. You will find that they are ~10 and ~2, respectively, as we expect given the arguments we provided when generating the 50 random numbers in the first place.

(Now, don't modify or run your script until directed to do otherwise.)

Another way to generate data is to use an equation. One such equation relevant to the atmospheric sciences is that of a sine wave. In your script, remove all lines after the one in which you created the figure and axes objects. Next, we want to define some values at which we want to evaluate the sine function. A good range would be from -2π to $+2\pi$. We can do this pretty easily in Python:

```
xvals = np.linspace(-2*np.pi, 2*np.pi, 100)
```

Here, the numpy function `linspace` is used to generate a 100-element 1-D array with values ranging linearly from -2π to 2π (for which numpy has a built-in variable called pi to help with this calculation). Once we've set these values, we can evaluate the sine function at these locations:

```
yvals = np.sin(xvals)
```

And can go straight from there to plotting the data:

```
ax.plot(xvals, yvals)
fig.show()
```

Go ahead and run your script once you've added the lines above. The result should be the familiar sine wave, with values ranging from +1 at $\pi/2$ and $-3\pi/2$ to -1 at $-\pi/2$ and $3\pi/2$.

We can make this plot look even nicer. For instance, we can add axis labels by adding the following lines immediately before the `fig.show()` function call:

```
ax.set_ylabel('sin(x)')
ax.set_xlabel('angle (radians)')
```

We can make these axis labels larger by providing appropriate keyword arguments:

```
ax.set_ylabel('sin(x)', fontsize=18)
ax.set_xlabel('angle (radians)', fontsize=18)
```

We can make the *x*- and *y*-axis labels themselves larger, too:

```
ax.tick_params(labelsize=14)
```

Go ahead and add the most recent three lines to your code above the `fig.show()` function call and run the script.

(Now, don't modify or run your script until directed to do otherwise.)

We can even change the line style, color, and thickness used when plotting the data. To do so, the `ax.plot()` function call must have additional arguments passed to it, i.e.,

```
ax.plot(xvals, yvals, color='red', linewidth=4,
linestyle='dashed', marker='o', markerfacecolor='black',
                    markersize=8)
```

The above statement would plot the data with a red dashed line that is four pixels thick, as well as black markers that are 8 pixels wide at every *x*-axis data point. Naturally, you might not choose to have all of these elements on a given plot – the example above is only meant to be illustrative of the sorts of options you have when creating plots. In addition to line plots, pyplot can create bar charts, scatterplots, and many other plot types.

In addition to numbers, it is also possible to use text categories to label the *x*-axis. For instance, let us assume that we have a numpy array called `temps` that contains four temperatures for some city at four different times (0000, 0600, 1200, and 1800 UTC) on some day:

```
temps = np.array([76, 68, 66, 81])
```

We could set up a numpy array for the times that we can use to label the *x*-axis, i.e.,

```
times = np.array(['0000 UTC', '0600 UTC', '1200 UTC', '1800
                        UTC'])
```

And then we could plot the data as before:

```
ax.plot(times, temps)
```

If we had another set of temperature data in an array called `temps2`, we could display these data alongside the data in `temps` with another call to the plotting routine:

```
temps2 = np.array([55, 44, 38, 61])
ax.plot(times, temps2)
```

We can add a legend to help someone viewing the plot know what line corresponds to what city. The easiest way to do this is to modify our plot statements to include labels, then plot the legend:

```
ax.plot(times, temps, label='Milwaukee')
ax.plot(times, temps2, label='Seattle')
ax.legend(fontsize=20,loc='lower right')
```

The legend function call uses keyword arguments to control the font size used in its labels as well as the location of the legend within the plot window.

If desired, we can rotate the *x*-axis tick marks so that they are vertical instead of wide:

```
plt.xticks(rotation=-90)
```

This will rotate the *x*-axis tick marks on the entire plot by -90° so that they are vertical and can be read from left to right.

Let's add some axis labels and a title to this figure:

```
ax.set_ylabel('Temperature (°F)', fontsize=16)
ax.set_xlabel('Time (UTC)', fontsize=16)
ax.set_title('2-m Temperature for 1 January 2000')
```

Finally, instead of just having the image print to the Python window, we can also save it to a file:

```
fig.savefig('plot.png', bbox_inches='tight')
```

This will save the figure as plot.png in the same directory as your script. The `bbox_inches` call tells Python to reduce the white space around your plot when saving the image to file.

At this point, modify your script to define `temps`, `times`, and `temps2`; plot each with a label; add a legend; set the axis labels; title the figure; and save the figure to a file, then run the script.

For this assignment, you are to download the midnight, 6:00 am, noon, and 6:00 pm temperatures for two cities of your choosing from 6-12 August 2018. Use the Weather Underground interface from Assignment #3 to do so. Plot these data on a single plot using two different line colors, line styles, and line widths. Include a legend and labels for both axes, making sure to include units for the *y*-axis label. For the *x*-axis tick marks, include both the day and time, i.e., '6 Aug/12:00 am,' and rotate the tick marks for legibility. Finally, title the image with your name and the assignment number. Turn in your image *and* script in class or via e-mail by the start of class.