Data Visualization with Python

Session 3

Robert Palmere

2021

Topics:

- Generating Figures Using the Matplotlib Library
- Basic Plots Using the Seaborn Library
- Interacting with Data Using Python Plots

Python is a powerful tool for data visualization with access to extensive libraries for this purpose. In this session, we make use of the Matplotlib and Seaborn packages for creating static, and animated plots.

This document serves as a guide for its associated Jupyter Notebook.

Generating Figures with Matplotlib and Seaborn

First we import the necessary modules for this session.

```
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation
from matplotlib import colors
import numpy as np
```

We can now access the pyplot class of the matplotlib with the alias 'plt'. We generate a pseudo-random, normally distributed, data set using the 'random' class of numpy.

```
data = np.random.normal(0, 1, 1000)
```

The normal() function is accessed with the first two arguments specifying the span of the distribution ([0, 1]) and the last argument indicating the number of points to be generated within the span. Data is now a one-dimensional numpy

array.

To show a simple line plot of the data set we can first define the figure with use of the figure() class of pyplot. Subsequent statements for plotting will then default to the generated figure.

```
plt.figure()   # Generate a blank figure
plt.plot(data) # Draw a line plot on the figure
```

Outside of a Jupyter environment, the above code would not display a plot when ran. The show() method would be required in that case.

```
#!/usr/bin/env python3 # shebang
plt.figure()
plt.plot(data)
plt.show()
```

The 'shebang' at the top of the code specifies the Python interpreter.

The normality of the data is clearer using a histogram. We can generate a histogram using the hist() function of pyplot. The first argument of the histogram is a 1D iterable such as a list of numeric values or a 1D numpy array. Subsequent keyword arguments can be used to specify the binning of the data and plot display attributes.

```
data = np.random.normal(0, 1, 1000)      # NumPy 1D array
data2 = np.random.normal(0, 1, 1000) + 1 # NumPy 1D array shifted by 1 along x-axis
b1 = plt.hist(data, bins=20, edgecolor='k', facecolor='slateblue');
b2 = plt.hist(data2, bins=20, edgecolor='k', facecolor='orange');
plt.xlabel('Some Parameter')
plt.ylabel('Occurrence')
plt.show()
```

In the above code, adding 1 to the second data set propagates through the numpy array shifting the span to [1, 2]. By calling the hist() function on each of these arrays plots the two histograms to the same figure by default. Since there are no subplots, we can call xlabel() and ylabel() to display text along each axis.

Inspecting these plots shows that they overlay one another so that the histogram from 'data2' overlaps the histogram of 'data'. Perhaps we would like to preserve the binning of these histograms while still showing the entirety of each histogram so that none of the data is hidden from the viewer. One way to do this is by writting a function which displays a line plot using histogram bin centers as input.

```
def histlp(bins):
     heights, edges = bins[0], bins[1]
     points = []
     for i in range(len(edges)):
         if i > 0:
             point = edges[i-1] + ((edges[i] - edges[i-1]) / 2)
             points.append(point)
     return points, heights
```

The 'histlp' function takes in the bin heights and edges as the variable 'bins' which is returned by the hist() function. Within the function definition, since 'bins' contains both heights and the edges, we unpack this into two separate variables. The edges of each bin are then iterated over using a for loop. The current edge (i) and the previous edge (i-1) make up the two edges of a given bin in the histogram. Dividing the difference by 2 and adding it to the first edge gives the center of the bin which we assign as 'point'. We then return the original bin heights and all bin center points as a list to be plotted using plt.plot().

```
p1, h1 = histlp(b1)
p2, h2 = histlp(b2)
plt.plot(p1, h1)
plt.plot(p2, h2)
plt.show()
```

We can represent the data in these histograms as an estimated probability density function by using kernel density estimation. We can use the seaborn library to generate these plots.

```
import seasborn as sns
sns.kdeplot(data)
sns.kdeplot(data2)
```

After importing seaborn using the alias 'sns' we plot these data as kernel density estimates (KDEs) using the kdeplot() method. To avoid overlaying these plots we can generatetwo subplots using methods accessible from plt.figure() or plt.subplots() classes. Using the figure() class, we instantiate the figure and then add subplots using the add_subplot() method. This method takes an integer representing the dimensions and the position of the subplot.

```
figure = plt.figure()
figure.add_subplot(221) # 2x2 plot as the first subplot
figure.add_subplot(222) # 2x2 plot as the second subplot
sns.kdeplot(data, c='slateblue', lw=3, ax=figure.axes[0])
sns.kdeplot(data2, c='orange', lw=3, ax=figure.axes[1])
```

Each KDE plot can then be placed with the 'ax' keyword argument being set to

the particular subplot held in the axes attribute of the figure class. The other method is to use plt.subplots() as shown in the following example.

```
fig, ax = plt.subplots(1, 2)
sns.kdeplot(data, c='slateblue', lw=3, ax=ax[0])
sns.kdeplot(data, c='slateblue', lw=3, ax=ax[1])
plt.tight_layout()
plt.show()
```

Since there are multiple plots, the tick marks and labels may overlap one another. To avoid this, we can use the tight_layout() function to pad / adjust the subplots accordingly.

We can also use pyplot to plot scatter plots, error bars, vertical / horizontal lines, and to apply shadding between areas.

```
rx = np.arange(0, 1, 0.5)                       # range() does not support floats (hence np.
ry = rx
xd  = [random.rand() for i in range(50)]        # random data
yd = xd
plt.scatter(xd, yd) # scatter plot              # scatter plot
error = [random.random() for i in range(20)]    # random error
plt.errorbar(rx, ry, yerr=error) # line plot with error bars
plt.axvline(x=0.2) # veritcal line at x=0.2
plt.fill_between(rx, ry-error, ry+error) # Shade error
```

In order to add legends and format the text presented by the plot, we can use the legend() method and specify a variable for the 'handles' keyword. Calling plt.legend() alone may suffice if we used the 'label' keyword in our plots.

```
plt.plot(data, label='Data')
plt.legend()
plt.show()
```

However, if we want to indicate specific patches of color for our legends we will use the 'patches' module of matplotlib.

```
import matplotlib.patches as mpatches
line = mpatches.Patch(color='k', label='My Line', linestyle='-')
plt.legend(handles=[line])
plt.show()
```

In the above code, we instantiate a Patch object (class) using mpatches and pass an iterable containing this object to the 'handles' keyword of plt.legend. In addition to the legend, a simple way to adjust text-related plot properties is to call the rc() method of matplotlib before plt.show. "rc" stands for Runtime Configuration (like '.bashrc' or '.vimrc' for bash and VIM configurations, respectively).

```
import matplotlib as mpl
font = {'family' : 'Helvetica',
        'weight' : 'bold',}
mpl.rc('font', **font)
```

The rc() method accepts the target group (here its the 'font') and a keyword argument (dictionary) to specify the various parameters we would like to adjust for that group.

Going back to Seaborn plots, we are not limited to KDE plots. There are number of other plots we touch on in this session including loading data sets as a DataFrame from the pandas library, pairplot, regplot, which are demonstrated in the associated Jupyter notebook. It is important to note that Seaborn is rather good at working with pandas DataFrames. However, in a linear regression plot, seaborn does not have a way to show the underlying properities of the linear regression (i.e. R^2). To work around this, one can get the regression line from the matplotlib axis and use numpy to compute parameters.

```
ax = sns.regplot()   # returns mpl.axis
lines = ax.get_lines()   # return each line on the plot axis
reg_line = lines[0]      # regression line happens to be the first element
x = reg_line.get_xdata() # get the data along x for the regression line
y = reg_line.get_ydata() # get the data along y for the regression line
corr = np.corrcoef(x, y).astype(float) # get the correlation coefficients
rsquared = corr[0, 1]**2 # compute R-squared value
ax.text(3500, 230, str(rsquared)) # write the R-squared value on the plot
```

Animated Plots

In this section we will delve into displaying animated plots as well as some practical cases for the use of animated plots. My preferred way of displaying animated plots is to use a generator as the source of data for updates to the graph at each step of our animate function. Before we do this we should understand, generally, what a generator is.

```
def my_generator(num_list):
for i in num_list:
yield i*2

my_numbers = [1, 2, 3]

for i in my_generator(my_numbers):
print(i)
```

In the above code, we define a generator function called 'my_generator' which squares and *yields* each element of an iterable. Unlike a function that *returns* a list, a generator only generates the next element when it is required. This means that generators cannot be indexed / sliced. Another difference is that an iterable can be iterated over many times while a generator can only be iterated over once. The 'yield' keyword is what defines 'my_generator' as a generator function. the 'yield' keyword returns the function at an iteration without deallocating local variables which determine the iteration. This way the execution starts from the last yield statement. We use a loop, as shown above, to generate the squared values for printing. If we print 'my_generator' we will only display the generator object. Contrast this with a function which is not a generator:

```
def my_generator(num_list):
     for i in num_list:
         return i*2 # exits scope after at first ith element

my_generator(my_numbers)
```

Now that we understand what a generator is, let's move on to demonstrating the capabilities of animated plots and some use cases. To demonstrate an animated plot we will start by defining a class handling attributes of points we plan to plot on our graph called 'Atom'.

```
class Atom:
     def __init__(self, name, position, charge):
         self.name = name
         self.x = position[0]
         self.y = position[1]

     @property
     def position(self):
         return (self.x, self.y)

     @position.setter
     def position(self, pos):
         self.x = pos[0]
         self.y = pos[1]
```

The structure of this class and further details on classes in Python will be discussed in a future session. Now that we have defined our class, we can instantiate the class with parameters for name and position.

```
Na = Atom('Na+', (-3, 0))
```

```
Cl = Atom('Cl-', (3, 0))
```

Plotting of these objects can be done by accessing the x and y instance attributes of each using the . operator.

```
figure = plt.figure();
plt.scatter(Na.x, Na.y, s=10**2, edgecolor='k')
plt.scatter(Cl.x, Cl.y, s=10**2, edgecolor='k')
plt.xlim([-4, 4])
```

We now will write a generator function to move one of the points along the x-axis. This generator function will be used to produce values for updating the position of one of our points at each step of the animation.

```
def move(atom):
     sign = np.sign(atom.x)
     for step in range(1000):
         if sign > 0:
             atom.x = atom.x - 0.05
         elif sign < 0:
             atom.x = atom.x + 0.05
         yield atom.position
```

Now all we must do is import and use the 'FuncAnimation' class of the matplotlib.animation module. We will also import IPython.display.HTML to display the animated plot within the Jupyter notebook. We first generate a figure object to update, then define the 'animate' function which will update values depending on the generator function. The value produced by the generator function depends on the iteration or step that 'FuncAnimation' has arrived at.

```
from matplotlib import animation
from IPython.display import HTML # Required for Jupyter Notebook display

fig, ax = plt.subplots()
line = ax.scatter(Na.x, Na.y, s=10**2, edgecolor='k')
ax.scatter(Cl.x, Cl.y, s=10**2, edgecolor='k')

def animate(i):
     line.set_offsets((i[0], i[1]))
     line.axes.axis([-4, 4, -.4, .4])
     return line,

anim = animation.FuncAnimation(fig, animate, move(Na), interval=10)
HTML(anim.to_jshtml())
```

Notice that the 'animate' function we defined takes the argument 'i'. If we look

back at our generator ('move(Na)') we see that it yields the 'position' property of our 'Atom' class. The 'position' of each 'Atom' is a tuple for each objects x and y coordinates on the graph. Each time 'FuncAnimation' iterates forward, the next generated position is passed as a argument to 'animate' as the argument 'i'. We then update the scatter plot data by using the 'set_offsets' method of the scatter plot and return the updated scatter plot.

If you're interested, another annotated example, this time for generating 3D animated plots using molecular dynamics simulation data, is presented in the final section of the Jupyter notebook.