**Basic Data Visualization Through Matplotlib**

It might seem a bit tedious to create a matrix and then export it to excel to see any trends that your data shows, especially if you are constantly making tweaks to your data. Luckily for us, many packages have been made that can help us visualize data directly in Python. The module we will focus on is pyplot which is included in the matplotlib package.

**Importing Pyplot**

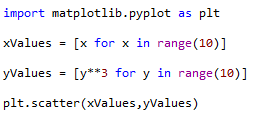
Like other packages we have used, we will need to first import the package before we use it. For now, we want to use the pyplot module from the matplotlib package, so we will import only that module and give it an alias of “plt” by doing the following.



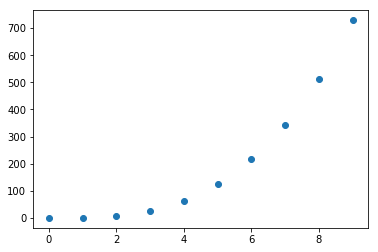
We now have access to the functions within pyplot. There are a lot of plots you can make with pyplot including line graphs, scatter plots, bar charts, and histograms. While the syntax varies from plot to plot, most have similar parameters. In this module, we will go over how to create scatter plots and surface plots. If there is another type of plot that you would like to use, you will need to look up the specific documentation at [http://www.scipy-lectures.org/intro/matplotlib/matplotlib.html](http://www.scipy-lectures.org/intro/matplotlib/matplotlib.html%20). However, once you know how to make one kind of plot, the others are easy to make.

**Scatter Plots**

First, we need some data to plot. Let’s create two lists for our x and y values that we want to plot using list comprehension. Our x values will go from 0 to 9, and our y values will go from 0 to 9 cubed. We will then pass our x and y lists to the plt.scatter() function, and a scatter plot will be generated.

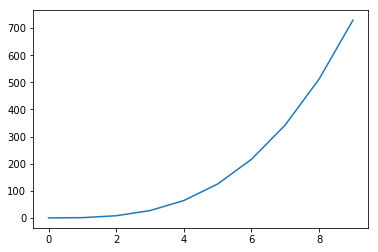


When you run your code, you should get a plot that shows up in the console that looks like what is shown below.

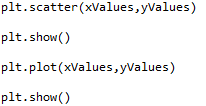


Alternatively if you wanted to make a plot with a line that connected the dots, you could use the plt.plot() function. From here on out we will just focus on scatter plots. All the information you learn from a scatter plot can be directly transferred over to line plots and probably other plots as well.





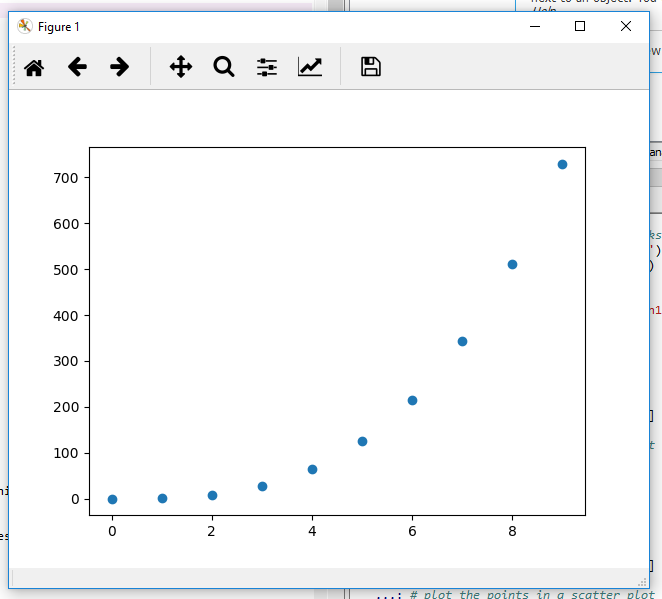
**You might have noticed that if you try to plot two plots consecutively it will plot your datasets on the same figure. To separate your figures use the plt.show() function.**



If you would like to **use these images in another document**, you can copy them by right clicking, copying it to your clipboard, and then pasting it in another document. If the resolution of the plot isn’t what you want, you can change Spyder’s settings to output the plot to another window. This can be done by following the path starting in the tools tab on the top of the Spyder application.

**Tools > preferences > IPython console > Graphics > Graphics backend > Backend: Automatic > Apply**

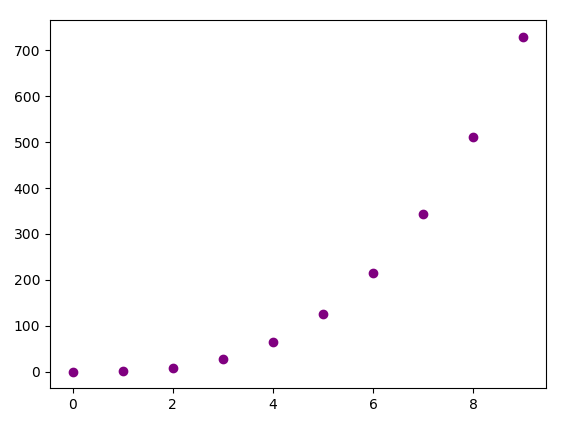
You will need to restart Spyder to make this change take effect. If you don’t see a plot pop up, try minimizing Spyder and looking for the plot behind Spyder.



Our scatter plot looks a little sad, so let’s see if we can make it a little more appealing. There are many things we can change about our plot, but for now let’s just focus on the color, size, and transparency of our points.

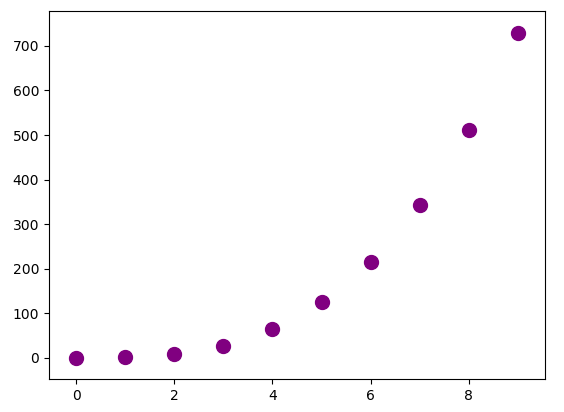
To change the color of our points, we will modify the “c” parameter in the plt.scatter() function, where “c” stand for “color”. We simply put the color we would like as a string for the c value. (Note: for a list of usable color arguments for the parameter “c”, see [this matplotlib documentation](https://matplotlib.org/3.1.0/gallery/color/named_colors.html).)





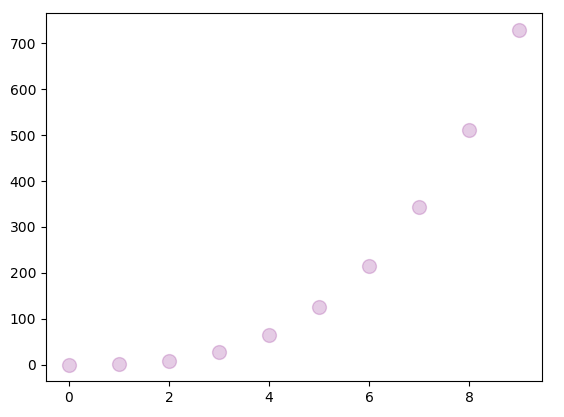
By setting the parameter “s” to a number in the plt.scatter() function, we can change the size of the points on our plot. Large values make the points bigger, and smaller values make the points smaller. Since this is all personal preference, you will need to see what values for “s” will make your graph look the best.



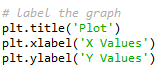


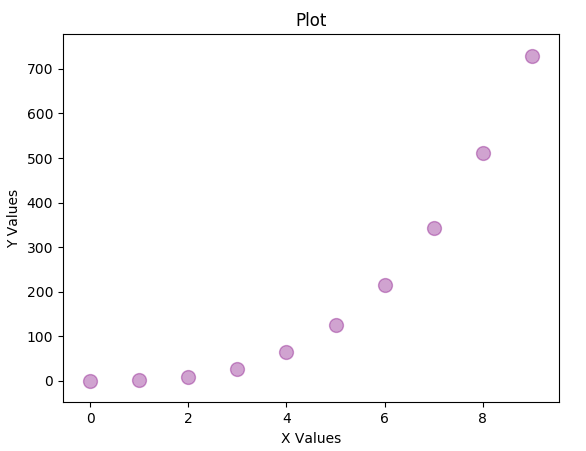
The last parameter of the plt.scatter() function we will mention is the alpha value. The alpha value can make the points translucent. Alpha values range from 0 to 1. The closer the alpha value is to 0, the more translucent it will be. The closer the alpha value is to 1, the more solid it will be. I’m going to give my points an alpha value of 0.2.





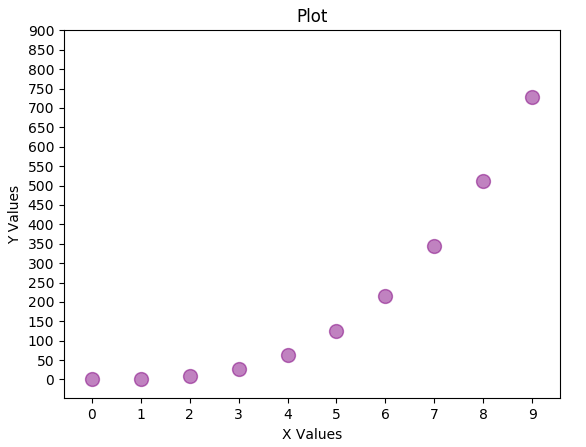
Now, we may want to add a title for our plot and labels for the x and y axes. To add these, we will use the plt.title(), plt.xlabel(), and plt.ylabel() functions. This is simply done by passing the words we would like to see as our title or label into these functions as a string.





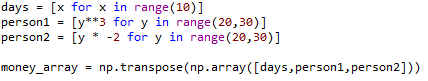
To change the tick marks on our x and y axes, we can use the plt.xticks() and plt.yticks() functions and put in a list of values we would like on the axes. To do this, we will use list comprehension to make a list of numbers from 0 to 9 with a step size of 1 for the x-axis and a list of numbers from 0 to 900 with a step size of 50 for the y-axis.





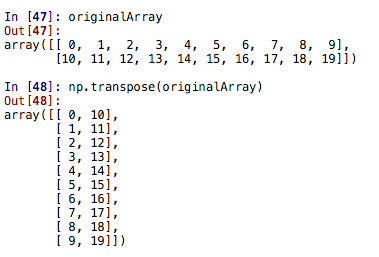
**Plotting Two Datasets on the Same Figure**

Let’s say we had a 10 x 3 numPy array with the first column being the time in days, the second column being the values of how much money person1 had on a given day, and the third column being how much money person2 had on a given day. We can make this data with the following code.



NumPy’s transpose function is used here, because when using “np.array()” each list will be read in as the next row instead of the next column. Transposing an array will switch the columns with the rows. For example: We want a 10 x 3 matrix instead of a 3 x 10 matrix.

Example:



Since the person1 and person2 columns are associated with the same x value, it makes sense to plot person1 and person2’s data on the same plot. To do this, we can use the plt.scatter() function twice using the same x values but different y values.

***The Colon Operator***

To access slices of an arrayinstead of a single element, we can use the colon operator ( : ) to say what values we want to grab. The colon operator is the syntax to ‘slice out’ sub-parts in sequences (of lists or matrices). For instance, if we wanted to access/grab every row of the first column we could reference them with this [ : , 0], This would be implemented like so:

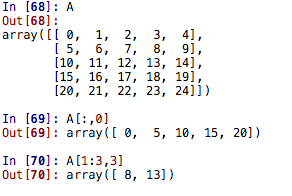


See how we accessed all the values of the first column of our “money\_array”?

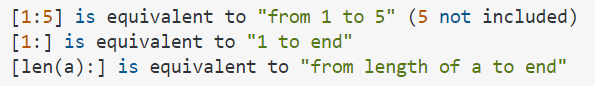
If we wanted only the 3rd – 6th elements of the first column, we could place the starting and ending value (non-inclusive) before and after the colon, respectively.



A way to think about it is, the first two numbers, separated by the “:”, represent the range of rows and the last number represents the column you want. In the following example you get all the rows but only the first column (column zero). Next, we get the rows 1 and 2 but only with the fourth column. (Remember python starts counting from 0).

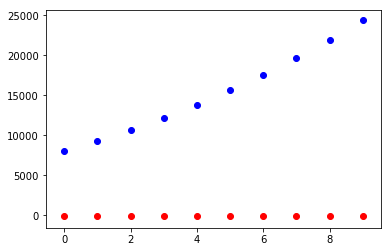
****

To summarize, “:” can be used to easily slice as in the following:



Going back to our example, now we can grab the slices of the matrix we want by using the colon operator. This can be done within the plt.scatter() function to use specific columns of the matrix as values for x or y in our plot. We can easily call the scatter function twice, with both person1 and person2’s money per day with the x values being the same.





**Plotting Two Datasets with Different Scales on the Same Figure**

While we figured out how to plot the data on the same plot, person2’s data points (red) seem really insignificant to person1’s data points (blue). As you might have already guessed, the scales are the same for both datasets. This can make it hard to tell exactly what is going on with the data set in red. To remedy this, we will create a separate y-axis for both sets of data. To do this, we will need to implement subplots and the “.twinx()” function.

Typically, a subplot is used to have multiple plots within the same figure. You can think of figure as being in the same window. The following line of code is how you create a subplot (explanation will follow):

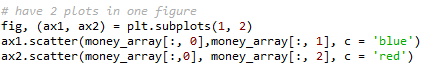


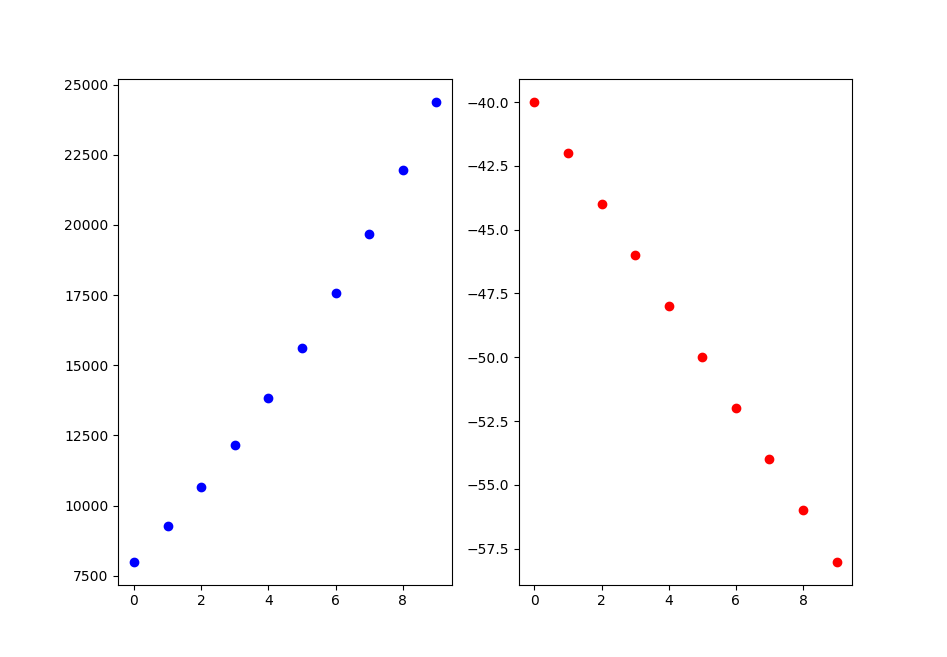
“fig,” – this is just syntactical. You can put whatever you want as long as you have the comma.

“(ax1, ax2)” – these are the different graphs you want to plot. Again you can call them whatever you like.

“plt.subplots(1,2)” – this is describing how you want the plots laid out. The first number indicates how many rows of graphs you want and the second column is how many columns. For this example, you will have 1 row with 2 columns. If this explanation doesn’t make sense, try different numbers and look at the output.

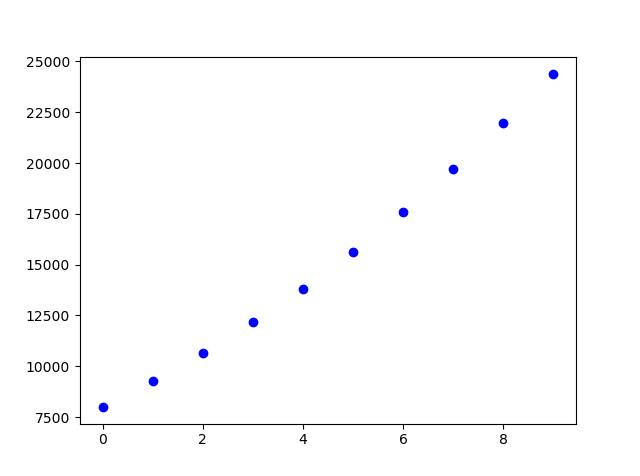
We can then plot our data to a specific set of axes by calling scatter on our ax1 and ax2 variables with whatever x and y values we want for each plot. In our case, we will use the money person1 and person2 have each day.





While we now can see the different trends for both person1 and person2, what we really wanted is to have one plot with two datasets sharing the x-axis but scaled differently in the y direction. This is where “.twinx()” comes in. To get the datasets in one figure, we will plot our first subplot. If we don’t pass in any arguments into the plt.subplots() function, it will default to giving us one figure and one set of axes. Then, like before, we can call the scatter function on that axis (or those axes).

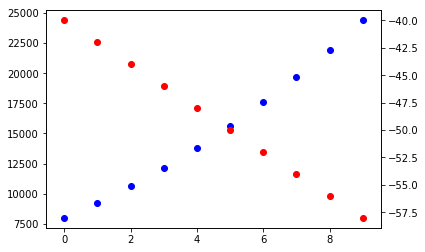




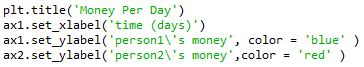
Now, to add another set of data to this plot we need to signify that our next axes will be sharing the x- axis with the axis “ax1”. To signify that the x-axis is going to be shared by both datasets, we will set the second axis (ax2) equal to “ax1.twinx()” and then call scatter on our second axis (ax2).

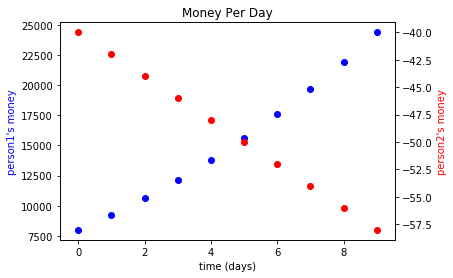


When we print this out, we should get what is shown below.



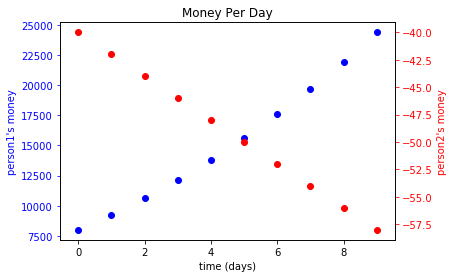
While, we have both the date sets on one graph, it still doesn’t look great. To make it easier to read, we can make a title for the plot and labels for our axes. Notice that if we want to label the x axis we need to use “ax1”, since the x axis used in the graph belongs to this subplot.



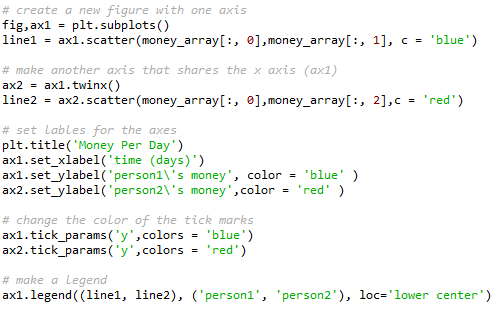


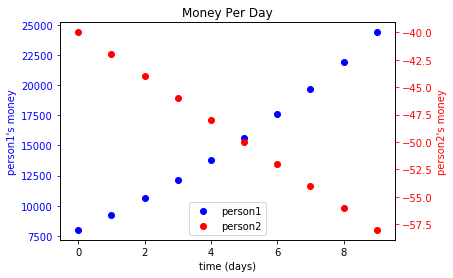
If we wanted to make it even clearer to others that the left y-axis belongs to the blue points and the right y-axis belongs to the red points, we could color the tick marks on the y-axis by calling the “.tick\_params()” function on ax1 and ax2 and explicitly tell it to color the y-axis the same color as the data points they are associated to. We do this by specifying we want to change the y-axis by passing y in as the first argument, and then passing in a string as an argument for the “colors” parameter.





Last of all we could add a legend to be very clear about what each of the different colored data points mean. To do this, we need to save the lines of our scatter plots in variables (in this case line1 and line2) and then pass them into the legend function as a tuple (list with parenthesis instead of square brackets) for the first parameter of the function, the labels of the legend as a tuple (‘person1’ and ‘person2’) as the second parameter, and then the location of the legend on the graph ( ‘lower center’ ) for the third parameter. For a list of the different locations you can place the legend on your plot please consult [https://matplotlib.org/api/pyplot\_api.html#matplotlib.pyplot.legend](https://matplotlib.org/api/pyplot_api.html%23matplotlib.pyplot.legend).





**Surface Plots**

Surface plots are a good way to visualize how two independent variables effect a dependent variable. To make a surface plot, we will need to import the mplot3d module from the mpl\_toolkits package.



Now, we will need some data to plot. Inside the folder for this module, you should see a csv file called “surface\_plot\_practice.csv”. This data has the first row as the x values, the first column as the y values, and the values that correspond to each are a function of both x and y. Import this data using the np.loadtxt() function that you learned from the last module. We will then need to set a new figure to a variable.



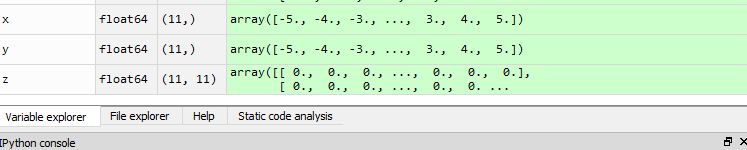
We will then set the axes of the figure to 3D axes.



Now, we need to set the x and y values for the plot. For this dataset, the x and y values both go from 5 to -5 by increments of 1, inclusive, and as mentioned above, are the first row for the x values and the first column for the y values, and the rest of the values are a function of x and y, which we will call our z values. To grab the data we want for these values, we will need to slice the data as we did before



We can see that we sliced the matrix correctly by looking at the variables x, y, and z in the variable explorer.



Since we are plotting in 3D, we need a grid that consists of every combination of our x and y values. Only then can we plot the z values within the 3D grid. To make our x and y arrays become a grid, we can use the np.meshgrid() function to make both the x and y arrays into 2D matrices.



We can now see that our x and y values have been turned into 2D arrays in the variable explorer.

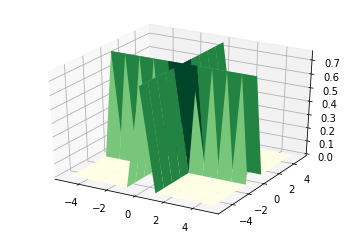


The last thing we need is a color scheme for our surface plot. To add a color scheme to the plot we will need to import cm, which stands for color map, from the matplotlib package.

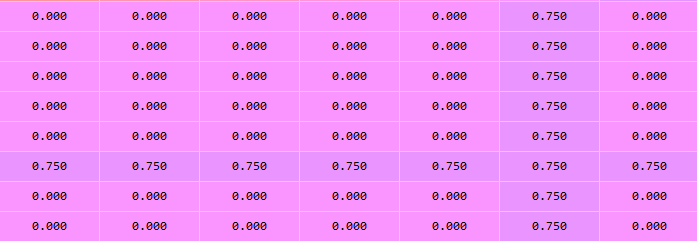


Now we can make our surface plot by calling plot\_surface() on the axes that we made (ax) and passing in our x, y, and z values (the matrix we read in) along with a color map. I used the “YlGn” color gradient, but to see all the possible color schemes see the [matplotlib colormap reference](https://matplotlib.org/examples/color/colormaps_reference.html).

****



It might be a good idea to check if the surface plot corresponds to the data. We can go to the variable explorer again to make sure that they match up.



From the variable explorer, the plot looks plausible.

To finish this graph, we should add a color bar to our map to correlate with the different color gradients we see in our graph. To do this we will need to call colorbar() on the figure that we made (fig), and pass in our surface plot as a parameter.



For surface plots, I would suggest changing the graphics settings in Spyder so it will open in a new window. You can then rotate it until you get it to a desirable position. When we run our code, we get the graph shown below.

