Day6-Intro_plotting

July 30, 2021

Day 6: Introduction to Plotting

We've had a busy week last week! We went over data types and storage, automating your code, and an introduction to `NumPy`!

Today, we're going to go over an introduction to plotting in Python (**yay**!). Data visualization is arguably one of the most important components of research and science communication. Always visualize your data early on in your research process because it helps you identify errors in your data!

Goals for the day:

- Get used to plotting basic plots in `Matplotlib`
- Make multiple subplots in one figure
- Plot multiple lines on one figure
- Make scatterplots and histograms
- Learn to plot using a function
- Go through some extra resources for cool plots using `Altair`

Functions Learned:

- Show plot: `plt.show()`
- Create subplots: `plt.subplots()`
- Create a scatterplot: `ax.scatter()`
- Create histogram: `hist()`
- Add colorbar to plot: `plt.colorbar()`
- Add legend to plot: `ax.legend()`

Set Directory

[1]:
```
#this is the specific directory where the data we want to use is stored
datadirectory = '../data/'

#this is the directory where we want to store the data we finish analyzing
data_out_directory='../output/'
```

1 Matplotlib

First, we're going to import an important package `matplotlib`, which is a great base for plotting and is highly customizable. We will give a subsection of `matplotlib.pyplot` a nickname, `plt`, the same as we did for numpy last week.

We are importing only a section of `matplotlib`, which is intended for simple and interactive plots. There is a huge amount of customization functions, which can be found here.

[2]:
```
import matplotlib.pyplot as plt
import numpy as np
```

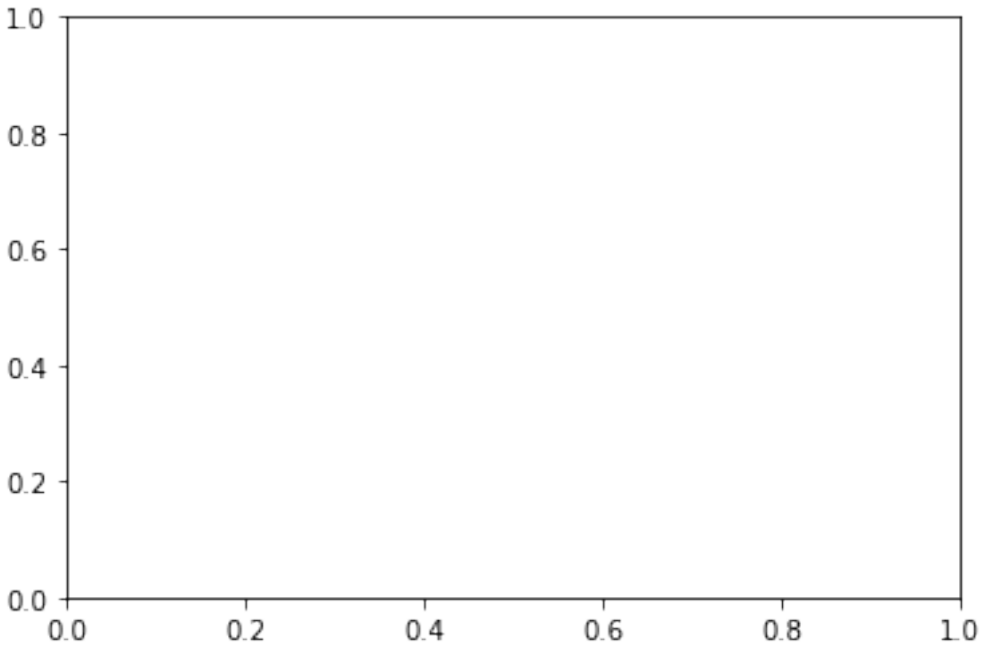
1.1 Let's do some plots

The mechanics of what is happening here is in the first line, we use a common `Matplotlib` convention by calling `subplots`, which allows you to plot multiple plots at once. The variable `fig` represents the entire set of figures and the variable `ax` represents a single plot within `fig` and is what we will then be using.

Remember that `plt` is our shortcut to access `matplotlib` and we are trying to save ourselves some keystrokes down the line!

First, we can set up an empty figure using `subplots`. This is the default that we will then build upon.

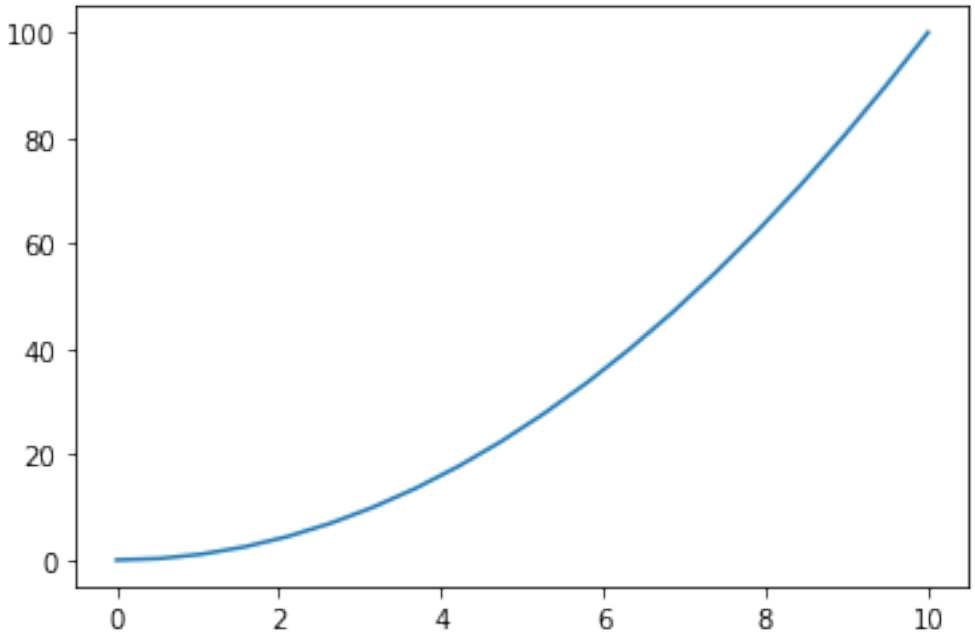
[3]:
```
fig, ax = plt.subplots() # building our figure.

plt.show() # use the show function to show the figure we've made
```



We can generate some data using `np.linspace` as we did in the `numpy` lecture. We will create a array of 20 numbers from 0 to 10 as `input_values` and then calculate the square of those values as an array called `squares`.

Then we use the function `plot` on our subplot that we called `ax`, which will generate a line plot with `input_values` on our x axis and `squares` on our y axis. Then we use the function `show` to print the plot to the command line.

[4]:
```
input_values = x = np.linspace(0,10,20) # array of x values
squares = x**2 # array of y values

fig, ax = plt.subplots() # setting up our empty figure space
ax.plot(input_values, squares) #ax.plot(x,y)

plt.show() # shows our plot for us
```



2 Axis labels, types of plots, shapes, and colors

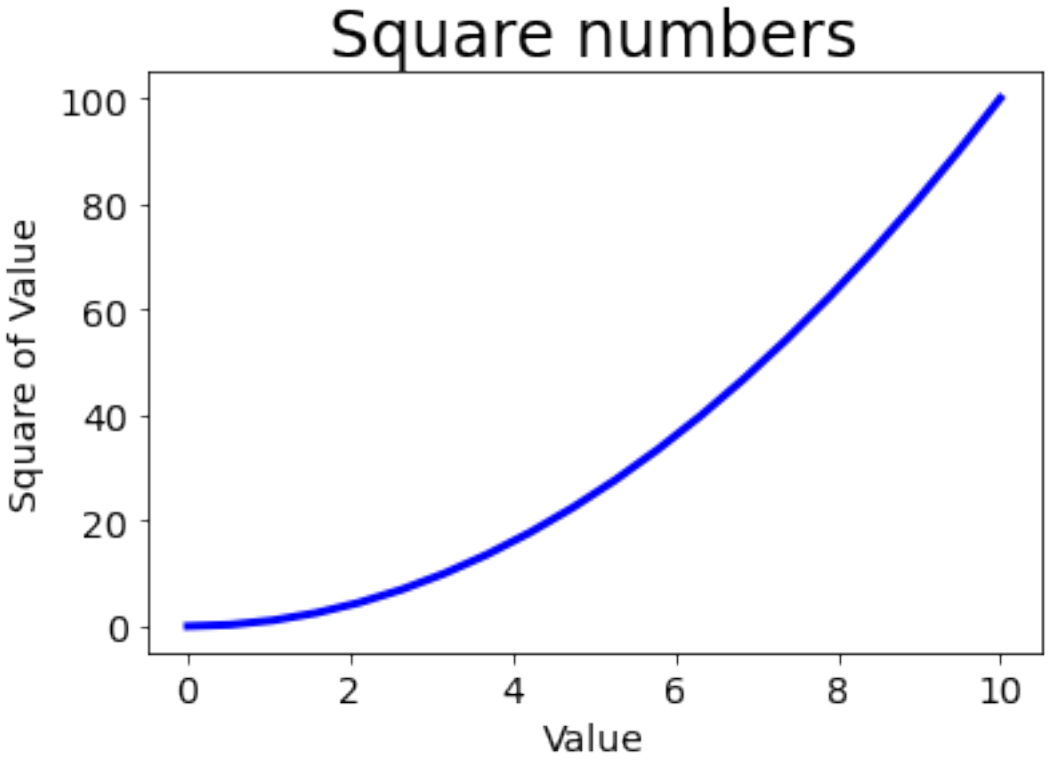
The plot is correct, but it's a little boring and not very informative. Also note that Python assumed that I wanted a line plot, when I didn't really tell them what I wanted. If you try to share this information, the first question will be "What are the axes??" (My PI would also say "I'm old, I can't see them!!")

So let's try again and add some information and change the linewidth and color. We will do that by building on our plot variable `ax`.

The documentation for `Matplotlib` will show you the different color and symbol options.

We will set our title, xlabel, ylabel and make the labels on our axes bigger

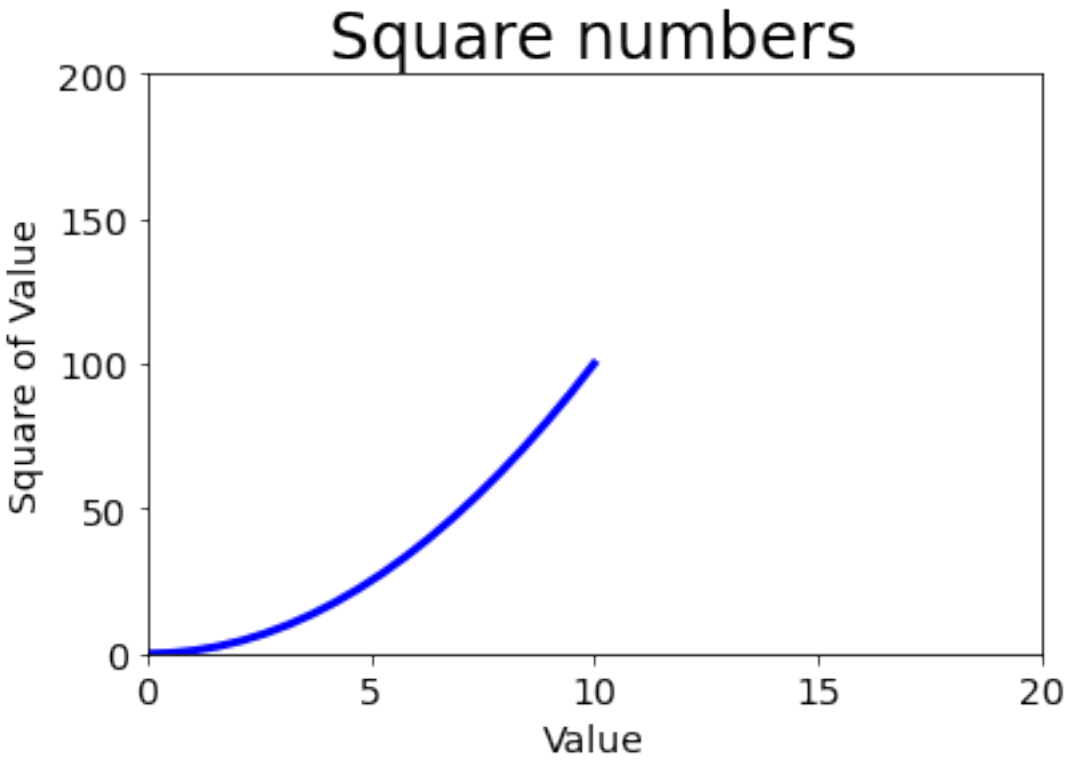
[5]:
```
input_values = x = np.linspace(0,10,20) # array of x values
squares = x**2 # array of y values

# set up our basic plot
fig, ax = plt.subplots()
ax.plot(input_values, squares,  'blue', linewidth = 3)

# set up our axes and title
ax.set_title("Square numbers", fontsize = 24) # adds a title
ax.set_xlabel("Value", fontsize = 14) # adds x label
ax.set_ylabel("Square of Value", fontsize = 14) # adds y label

ax.tick_params(axis = "both", labelsize = 14) # increases font size for both axes

plt.show()
```



2.1 Changing the axis

It comes up a lot that you want to change your axis. The documentation for some of that is found here. We can use `set` to manually set our axis labels. There are also other ways to do more specific things, like setting the limits to be equal to each other.

[6]:
```
input_values = x = np.linspace(0,10,20) # array of x values
squares = x**2 # array of y values

# set up our basic plot
fig, ax = plt.subplots()
ax.plot(input_values, squares,  'blue', linewidth = 3)

# set up our axes and title
ax.set_title("Square numbers", fontsize = 24) # adds a title
ax.set_xlabel("Value", fontsize = 14) # adds x label
ax.set_ylabel("Square of Value", fontsize = 14) # adds y label

ax.set(xlim=(0,20), ylim=(0, 200)) # manually setting our axis labels


ax.tick_params(axis = "both", labelsize = 14) # increases font size for both axes

plt.show()
```

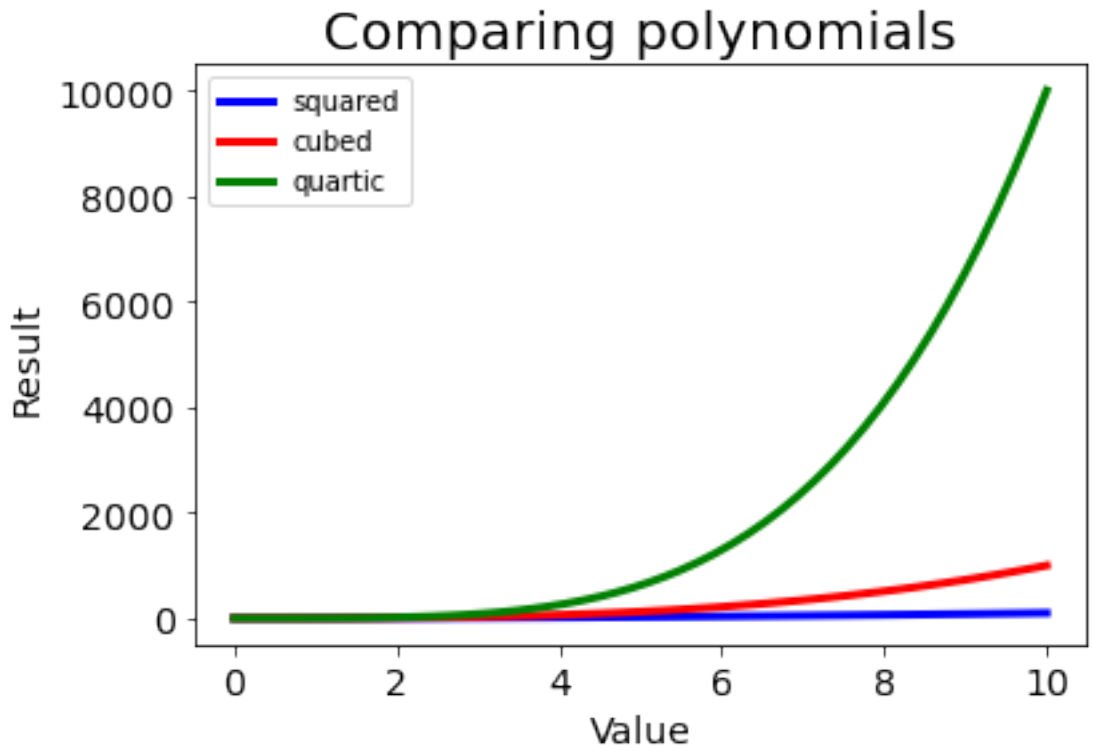


3 Multiple lines on one plot

Sometimes, you need to show multiple lines or points at once. We can do so by building on our `ax` variable with multiple plots.

We create different y values for a set of x values by raising x to different polynomials. We then use `ax.plot` multiple times with the new y values and this time we include a `label`. This will allow us to use `ax.legend()` to put a legend on the plot, so we can tell which line is which.

[7]:
```
x = np.linspace(0,10)
y1 = x**2
y2 = x**3
y3 = x**4

# set up our basic plot
fig, ax = plt.subplots()
ax.plot(x, y1,  'blue', linewidth = 3, label = "squared") # first line
ax.plot(x, y2,  'red', linewidth = 3, label = "cubed") # second line
ax.plot(x, y3,  'green', linewidth = 3, label = "quartic") # third line

# set up our axes and title
ax.set_title("Comparing polynomials", fontsize = 20) # adds a title
ax.set_xlabel("Value", fontsize = 14) # adds x label
ax.set_ylabel("Result", fontsize = 14) # adds y label

ax.tick_params(axis = "both", labelsize = 14) # increases font size for both axes
ax.legend() # adding the legend

plt.show()
```



4 Coding check in

Using `np.linspace`, make a list of x values called `x`. Using `np.sin(x)`, make a sine curve of those x values called `y`. Plot these values. Mess around with the color and linewidth to make it however you like, but make sure to add axis labels (with something that might follow a sine curve) and make them readable.

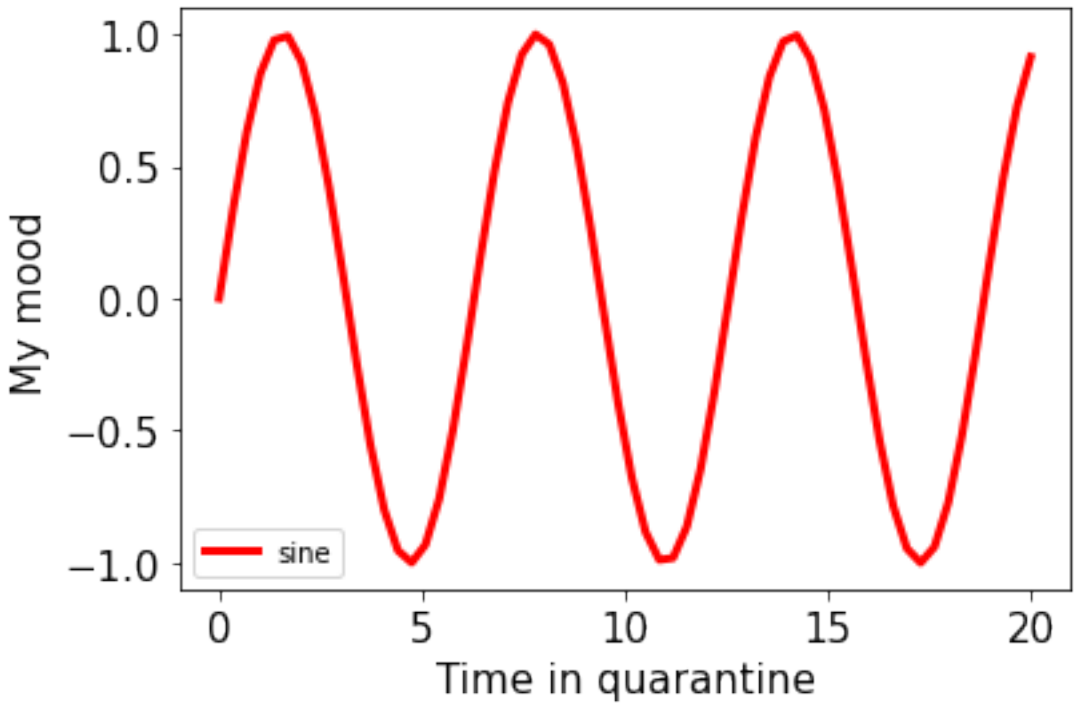
Advanced: create a second set of of y values (`y2`) using `np.cos(x)` and then plot it on the same graph.

[8]:
```
### coding check in here

# set your x and y values



# set up the base for the plot
```

```
# set axis labels, tick sizes and add the legend




# show the plot
```

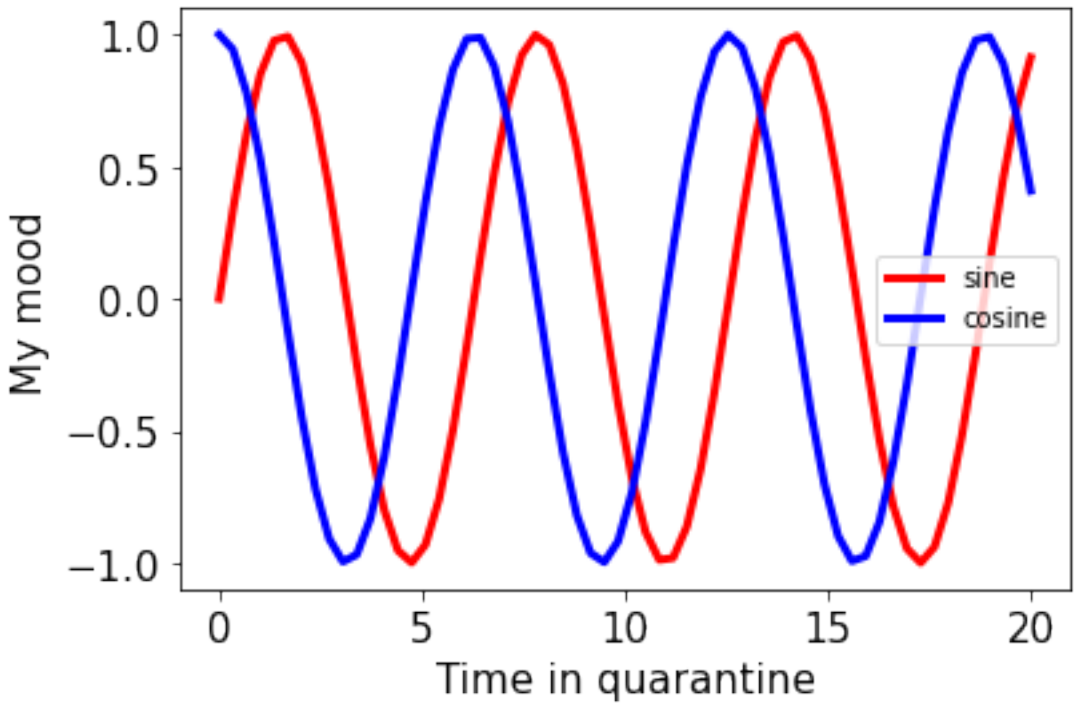
4.1 Coding check in answer

[9]:
```
x = np.linspace(0,20,60)
y = np.sin(x)

fig, ax = plt.subplots()
ax.plot(x,y, 'red', linewidth = 3, label = "sine")

ax.set_xlabel("Time in quarantine", fontsize = 15)
ax.set_ylabel("My mood", fontsize = 15)

ax.tick_params(axis = "both", labelsize = 15) # increases font size for both axes
ax.legend() # adding the legend


plt.show()
```



4.2 Advanced coding check in answer

[10]:
```
x = np.linspace(0,20,60)
y1 = np.sin(x)
y2 = np.cos(x)

fig, ax = plt.subplots()
ax.plot(x,y1, 'red', linewidth = 3, label = "sine")
ax.plot(x,y2, 'blue', linewidth = 3, label = "cosine")

ax.set_xlabel("Time in quarantine", fontsize = 15)
ax.set_ylabel("My mood", fontsize = 15)

ax.tick_params(axis = "both", labelsize = 15) # increases font size for both axes
ax.legend()

plt.show()
```



5 Multiple plots in one figure using `subplots`

We've done our first informative plot! This is so great, but what if we want to show multiple plots at once? In our research, we commonly show multiple plots at once for different relationships of

our regressors. First, we will begin by using `np.linspace` to create an array of values.

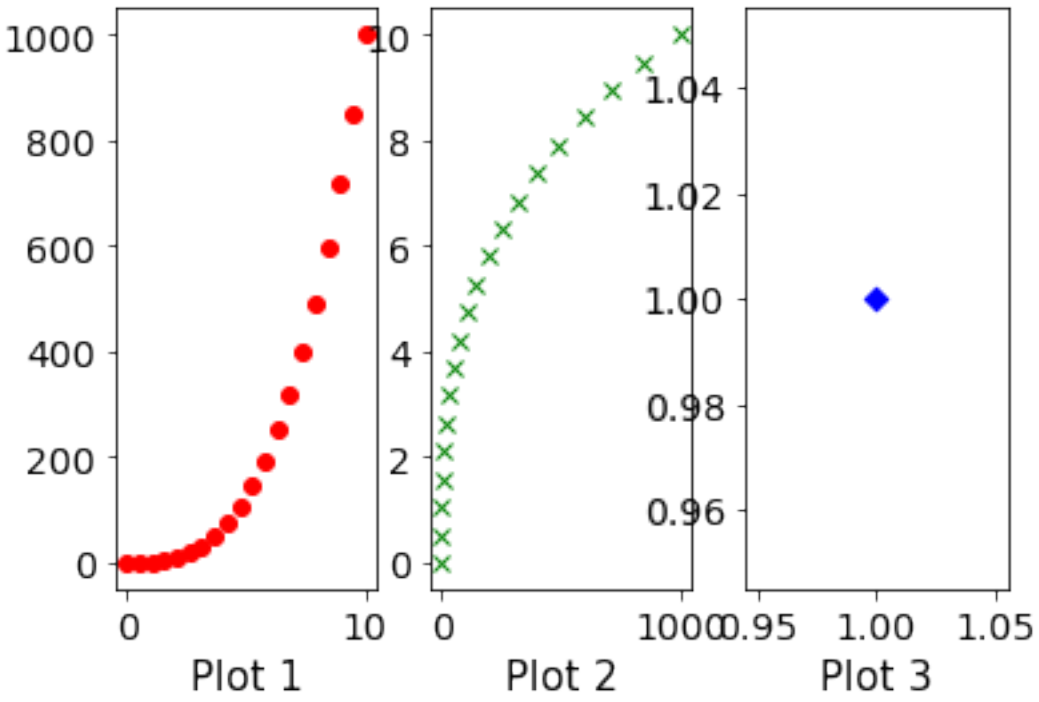
Using `subplots`, we set up three subplots in a row. We can then reference each subplot individually by indexing `ax`.

The first subplot we will make with red circles, the second with green xs, and the third will be a blue diamond.

The third plot will be a `scatter` plot, which makes it easy to plot just one point. Sometimes you'll want to highlight a data point, so this is how.

We then set the axis labels and sizes individually for each of the plots.

[11]:
```
x = np.linspace(0,10,20)
y = x**3 # now we are cubing

#plt.subplot(nrows,ncols,plot_number)

fig, ax = plt.subplots(nrows=1,ncols=3) # setting up our three subplots

ax[0].plot(x,y,'ro') # the first subplot
ax[1].plot(y,x,'gx') # the second subplot
ax[2].scatter(1,1,c = "blue", marker = "D") # the third subplot

ax[0].set_xlabel('Plot 1', fontsize = 15) # xlabel for first subplot
ax[1].set_xlabel('Plot 2', fontsize = 15)
ax[2].set_xlabel('Plot 3', fontsize = 15)

ax[0].tick_params(axis = "both", labelsize = 14) # increases font size for both␣
 ↪axes
ax[1].tick_params(axis = "both", labelsize = 14)
ax[2].tick_params(axis = "both", labelsize = 14)
```

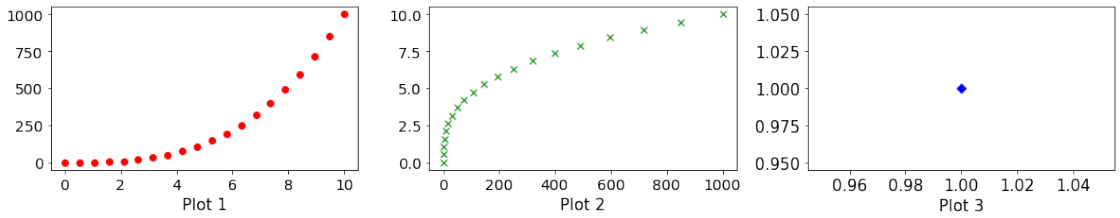


5.1 Tight Layout and changing the figure size

This figure looks pretty mushed together, so we will change the figure size when setting up the subplots, and we will use the command `tight_layout` to make sure the plots aren't overlapping

[12]:
```
x = np.linspace(0,10,20)
y = x**3 # now we are cubing

#plt.subplot(nrows,ncols,figsize)

fig, ax = plt.subplots(nrows=1,ncols=3, figsize = (15,3))

ax[0].plot(x,y,'ro') # the first subplot
ax[1].plot(y,x,'gx') # the second subplot
ax[2].scatter(1,1,c = "blue", marker = "D") # the third subplot

ax[0].set_xlabel('Plot 1', fontsize = 15)
ax[1].set_xlabel('Plot 2', fontsize = 15)
ax[2].set_xlabel('Plot 3', fontsize = 15)

ax[0].tick_params(axis = "both", labelsize = 14) # increases font size for both␣
 ↪axes
ax[1].tick_params(axis = "both", labelsize = 14) # increases font size for both␣
 ↪axes
```

```
ax[2].tick_params(axis = "both", labelsize = 15) # increases font size for both␣
 ↪axes

plt.tight_layout()
```

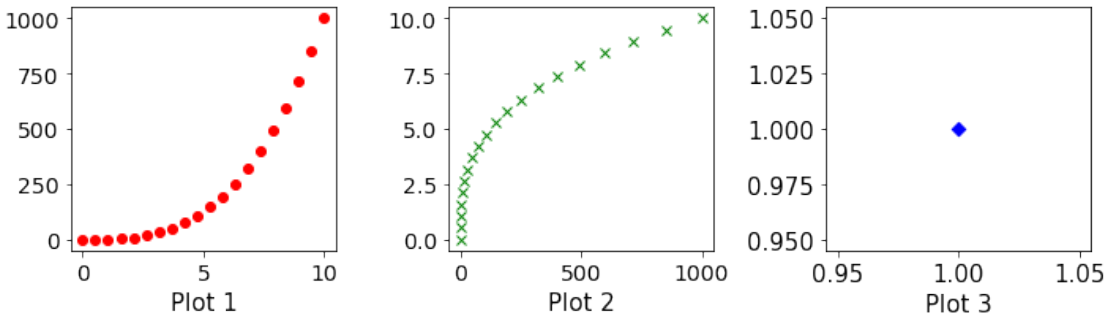


6 Saving your figures

Outputting your figures is very important to all aspects of science. We can use the function `savefig` with a file path to output our figure.

If you can't find your figure, check on your `data_out_directory` from earlier in the lesson.

[13]:
```
x = np.linspace(0,10,20)
y = x**3 # now we are cubing

#plt.subplot(nrows,ncols,figsize)

fig, ax = plt.subplots(nrows=1,ncols=3, figsize = (10,3))

ax[0].plot(x,y,'ro') # the first subplot
ax[1].plot(y,x,'gx') # the second subplot
ax[2].scatter(1,1,c = "blue", marker = "D") # the third subplot

ax[0].set_xlabel('Plot 1', fontsize = 15)
ax[1].set_xlabel('Plot 2', fontsize = 15)
ax[2].set_xlabel('Plot 3', fontsize = 15)

ax[0].tick_params(axis = "both", labelsize = 14) # increases font size for both␣
 ↪axes
ax[1].tick_params(axis = "both", labelsize = 14) # increases font size for both␣
 ↪axes
ax[2].tick_params(axis = "both", labelsize = 15) # increases font size for both␣
 ↪axes

plt.tight_layout()

#plt.savefig(data_out_directory+'katie_figure.png') # change it to be your name
```



[ ]:
```
print(data_out_directory) # to check to see where the plot is printing
```

```
/content/drive/My Drive/my_bootcamp/
```

7 Coding check in

Building off our sine and cosine plots from before, create a series of three subplots. On the first subplot, plot the sine function. On the second subplot, plot the cosine function. On the third sine plot, plot both the sine and cosine functions.

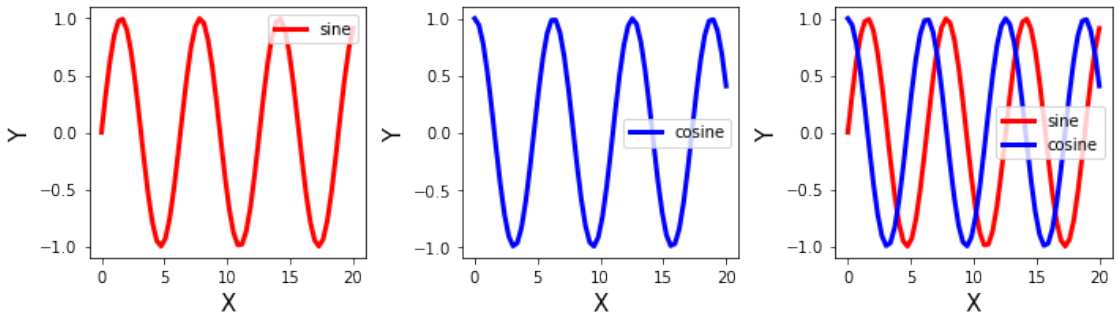
Make sure to use `tight_layout` and `figsize` to make your plot look nice. If you have time, make sure to add correct legends to each plot.

[14]:
```
### coding check in here
```

7.1 Coding check in answer

[15]:
```
x = np.linspace(0,20,60)
y1 = np.sin(x)
y2 = np.cos(x)

fig, ax = plt.subplots(nrows = 1, ncols = 3, figsize = (10,3))

ax[0].plot(x,y1, 'red', linewidth = 3, label = "sine") # plotting sin curve on␣
 ↪first plot
ax[1].plot(x,y2, 'blue', linewidth = 3, label = "cosine") # plotting cos curve␣
 ↪on second plot

ax[2].plot(x,y1, 'red', linewidth = 3, label = "sine") # sin curve on third plot
ax[2].plot(x,y2, 'blue', linewidth = 3, label = "cosine") # cos curve on third␣
 ↪plot


ax[0].set_xlabel("X", fontsize = 15)
ax[0].set_ylabel("Y", fontsize = 15)
```

```
ax[1].set_xlabel("X", fontsize = 15)
ax[1].set_ylabel("Y", fontsize = 15)

ax[2].set_xlabel("X", fontsize = 15)
ax[2].set_ylabel("Y", fontsize = 15)


ax[0].legend(loc = "upper right")
ax[1].legend()
ax[2].legend()

plt.tight_layout()

plt.savefig(data_out_directory+'sin_and_cos.png')
```



8 Histograms

Histograms are great for when you're trying to look at the distribution of your data. This could be important for what type of statistical test you want to do and it helps you identify outliers in your data.

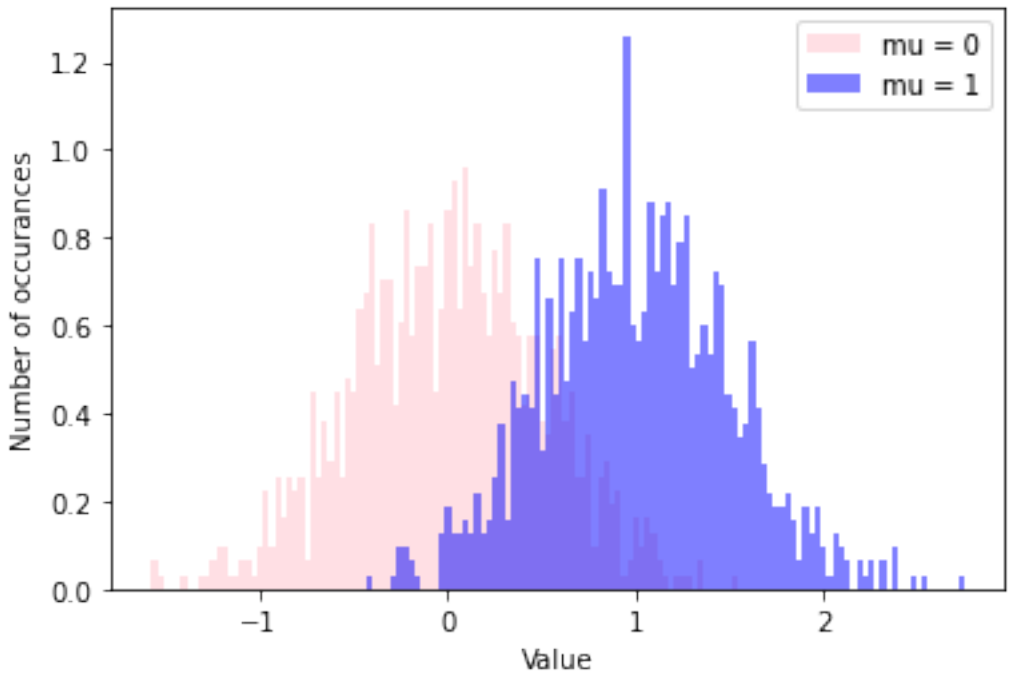
Here, we can generate two sets of a thousand random numbers from a normal distribution using our mean `mu` and standard deviation `sigma`. We will use two different means to compare the distributions. We then set the number of bins we want for our histogram and then call up our figure using `subplots`.

On the same graph, we will plot our two distributions. We can adjust the transparency by setting `alpha` to be some value less than 1, greater than 0.

We can also set our title to include the values for `mu` in our legend using an `f string` **Your graph will look different than the example!**

[16]:
```
mu1, sigma = 0, 0.5 # mean and standard deviation
mu2, sigma = 1, 0.5 #
sample_data_1 = np.random.normal(mu1, sigma, 1000) # generating sample data from␣
 ↪normal dist
```

```
sample_data_2 = np.random.normal(mu2, sigma, 1000) # generating sample data from␣
 ↪normal dist

num_bins = 100 # setting the number of bins for our histogram

fig, ax = plt.subplots()
ax.hist(sample_data_1, num_bins, density = True, color = "pink", label = f"mu =␣
 ↪{mu1}", alpha = 0.5)
ax.hist(sample_data_2, num_bins, density = True, color = "blue", label = f"mu =␣
 ↪{mu2}", alpha = 0.5)

ax.set_xlabel('Value')
ax.set_ylabel('Number of occurances')
ax.legend()

plt.show()
```



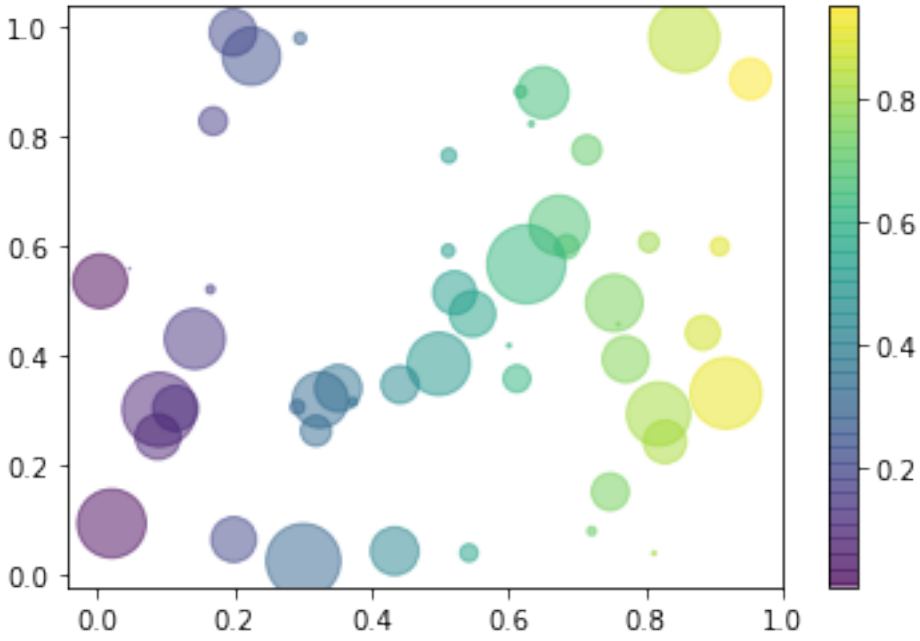
9 Colors and sizes in scatterplot

We can also manipulate the colors and sizes in scatterplots. We will draw random values from a uniform distribution between 0 and 1 using `np.random.rand()` for x and y. We will also draw random values between 0 and 15 for the sizes of the points.

For fun, we will color the points based on their y value, so as the points are laid out on the y axis, they will be colored.

We are also using `alpha = 0.5` here, which controls the opacity of the points, and can be between 0 and 1. This allows us to see the overlapping points. We can adjust this as well, where lower numbers are more faint and higher numbers are darker.

[17]:
```
# Fixing random state for reproducibility
# your plot will then look exactly like mine
np.random.seed(10)

N = 50
x = np.random.rand(N)
y = np.random.rand(N)
area = (30 * np.random.rand(N))**2  # 0 to 15 point radii for the areas

plt.scatter(x, y, s = area, c = x, alpha = 0.5) # c is what it will be colored␣
 ↪by, s is for the shape
plt.colorbar() # adding colorbar to our plots

plt.show()
```



10 Coding check in

First, create two arrays of 1000 random values, the x array from a `lognormal` distribution, and y from a `normal` distribution. Here are the setups for those two functions from the `numpy` package.

`np.random.lognormal(mean,sigma,size)`

`np.random.normal(loc,scale,size)`

Next, create four subplots in a two by two setup. Set the top two to be scatterplots of x and y and vice-versa. The bottom two will be one histrogram of x and one histogram of y.

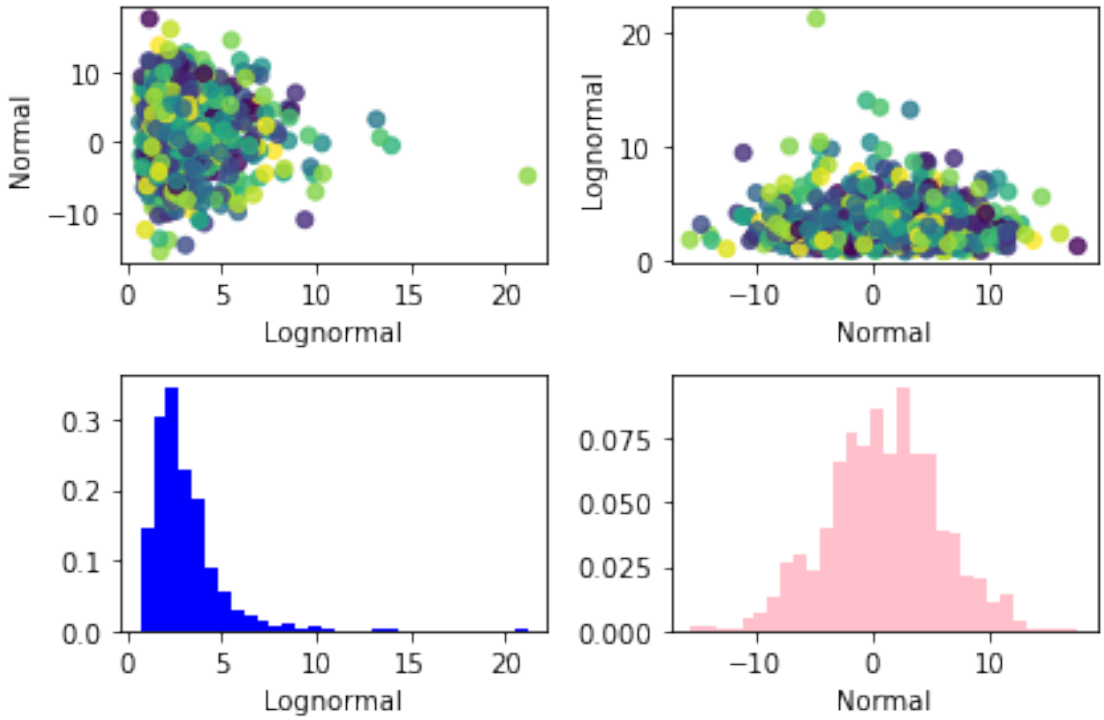
Color all of the values by their position on y and and correctly label all the axis.

[ ]:
```
### Space for coding check in code

# set x and y values



# set up your four subplots







# build on your four subplots

ax[0,0] # top left plot
ax[0,1] # top right plot
ax[1,0] # bottom left plot
ax[1,1] # bottom right plot




# set your x labels




# show your plot
```

10.1 Coding check in answer

[19]:
```
np.random.seed(2)

N = 1000

x = np.random.lognormal(mean = 1, sigma = 0.5, size = N)
y = np.random.normal(loc = 1, scale = 5, size = N)
colors = np.random.rand(N)

fig, ax = plt.subplots(nrows=2,ncols=2)

ax[0,0].scatter(x,y, c = colors, alpha = 0.8)
```

```
ax[0,1].scatter(y,x, c = colors, alpha = 0.8)

ax[1,0].hist(x, 30, color = "blue", density=True)
ax[1,1].hist(y, 30, color = "pink", density=True)


ax[0,0].set_xlabel('Lognormal')
ax[0,0].set_ylabel('Normal')

ax[0,1].set_xlabel('Normal')
ax[0,1].set_ylabel('Lognormal')

ax[1,0].set_xlabel('Lognormal')
ax[1,1].set_xlabel('Normal')

plt.tight_layout()
```

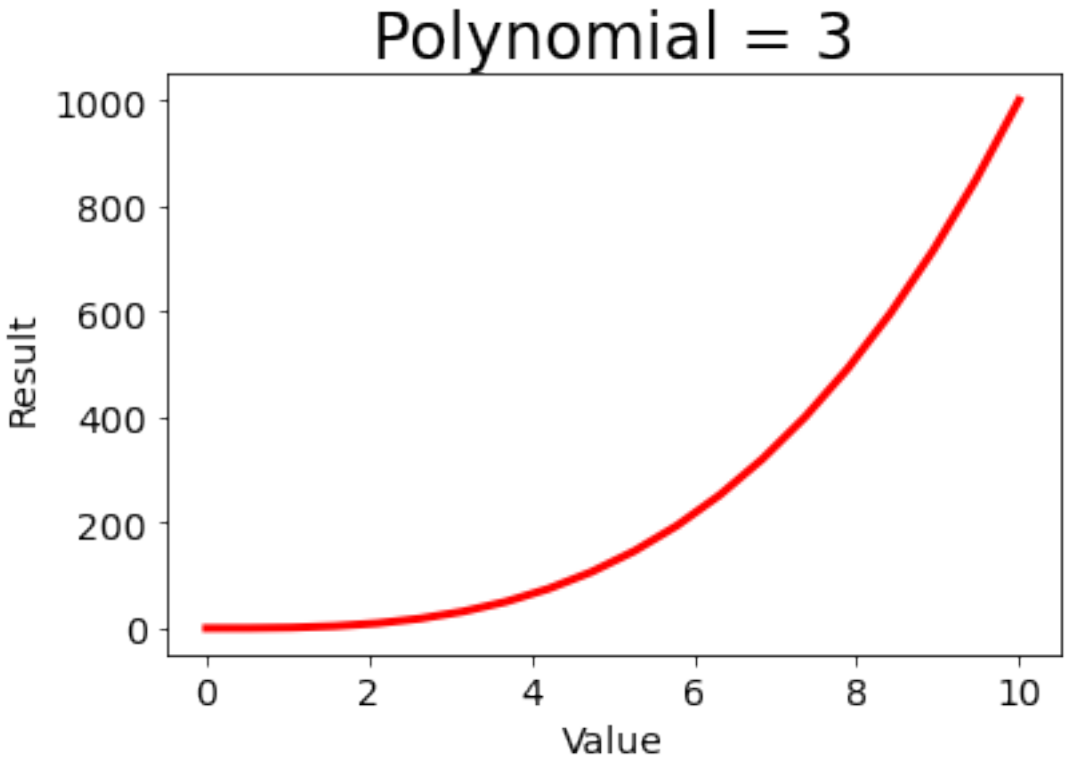


11 Plotting using a function

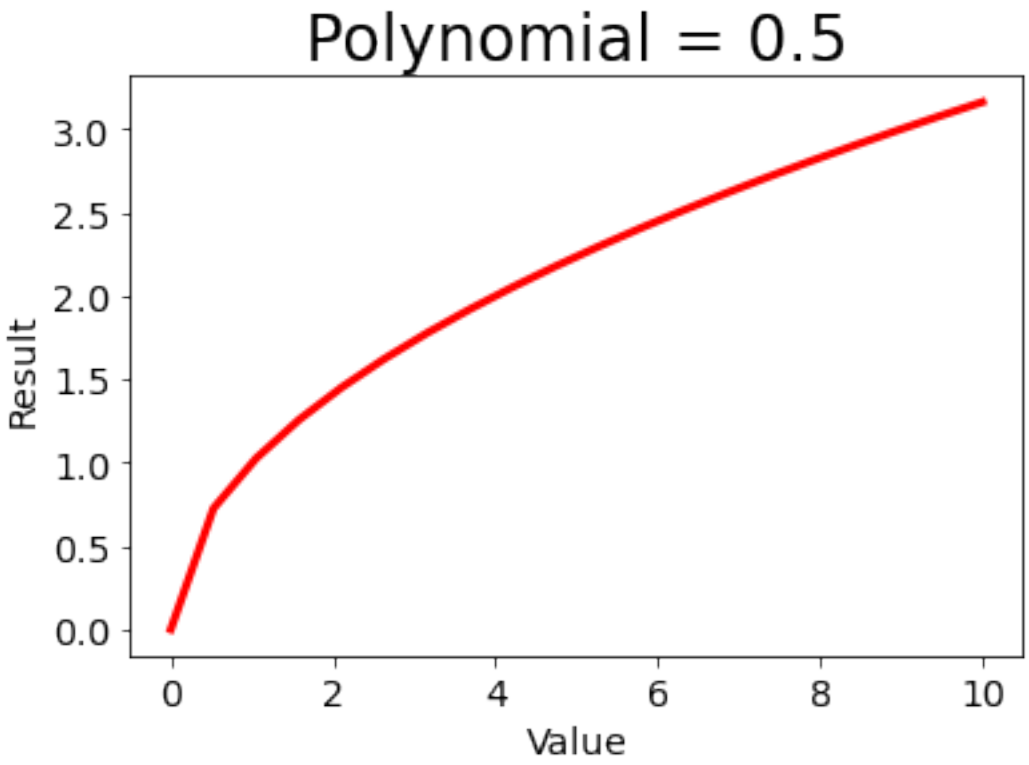
Very often in science we have to create a lot of plots, we might want to give different inputs, change datasets, or change features about the plot. It could be very helpful to create a function that outputs a plot, which is one of the examples of a plot that doesn't return anything, but prints a plot either in your notebook or as a file.

[20]:
```
# creating a plot that plots a polynomial of our choosing

def plot_polynomial(value, color_val):

  x = np.linspace(0,10,20)
  y = x**value # use the arguement for our function to create y


  # set up our basic plot
  fig, ax = plt.subplots()
  ax.plot(x, y,  color_val, linewidth = 3)

  # set up our axes and title
  ax.set_title(f"Polynomial = {value}", fontsize = 24) # adds a title
  ax.set_xlabel("Value", fontsize = 14) # adds x label
  ax.set_ylabel("Result", fontsize = 14) # adds y label

  ax.tick_params(axis = "both", labelsize = 14) # increases font size for both␣
 ↪axes

  plt.show()
```

[21]:
```
plot_polynomial(value = 3, color_val = "red")
```



[22]:
```
plot_polynomial(0.5, color_val="red")
```



12 Lotka Volterra plotting function (real life example)

Now I'll show you a real world example of a function to plot that an Ecologist would use.

I will be showing you how to plot the output of a classic set of equations showing a predator prey relationship. You may have learned about the Lynx-Hare population cycles in your high school biology class. Essentially, the Lynx will eat the Hare, the Hare's population will decrease and the Lynx population will suffer because there isn't as much food. Then the Hare will rebound because the predators have decreased, and then the Lynx will rebound as well. It's a little more complicated than this, but this is just the basics.

This results in a population cycle, where the population size in each year depends on the year before and the Lynx and Hare are intertwined. The discrete equations for the prey (V) and predator (P) are known as the **Lotka Volterra** equations. V_t is the time step the year after V_{t-1}.

$$V_t = rV_{t-1}(1 + R - \frac{RV_{t-1}}{K}) - aV_{t-1}P_{t-1}$$
$$P_t = P_{t-1}(1 - d + bV_{t-1})$$

The details of which aren't that important to us right now, but you can read more about them here.

- K is the carrying capacity
- r is the growth rate of the prety
- a is the attack rate of the predators
- d is the death rate of the predators

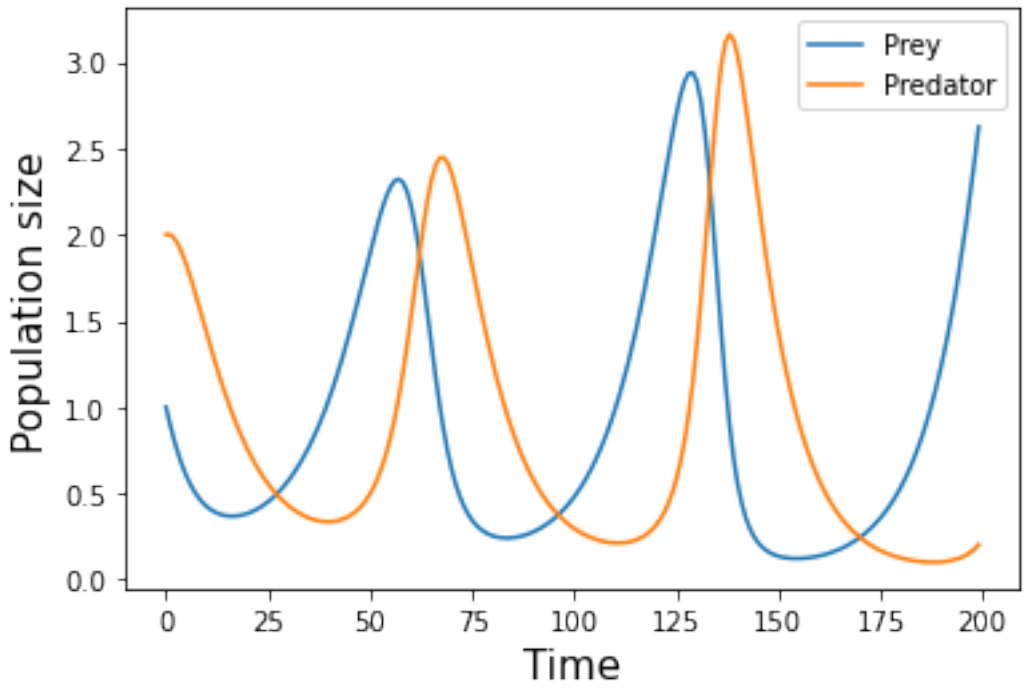
Inside the function, we have a lot of components.

- Creating an empty array of zeros for as long as we want to simulate (`MaxTime`)
- Initializing our prey (V) and predator (P) populations with `Init_V` and `Init_P`
- for loop that starts at time 2 and goes through `MaxTime` and simulates the predator and prey populations using the parameters
- plot the populations and then format the plot

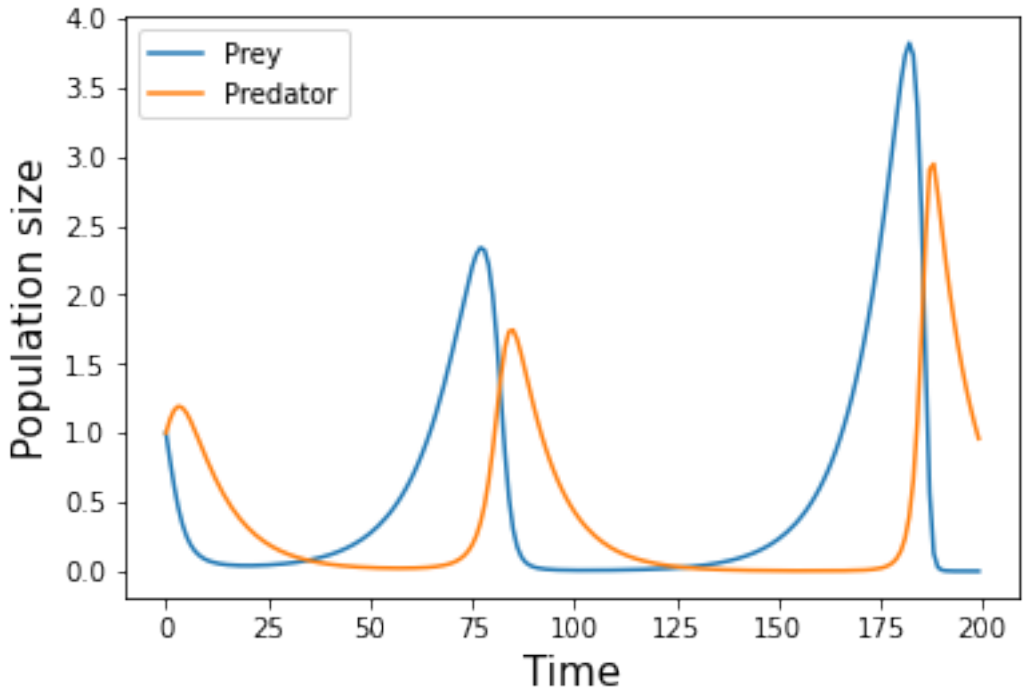
[23]:
```
def plot_lotka_volterra(Init_V, Init_P, R, K, a, b, d, MaxTime):

  V_T = [0]*MaxTime # initialize a list that is MaxTime elements long of zeros
  P_T = [0]*MaxTime

  V_T[0] = Init_V # setting the first value to the initial conditions
  P_T[0] = Init_P

  for i in range(1,MaxTime,1):
    V_T[i] =  V_T[i-1] * (1 + R - (R*V_T[i-1]/K)) - a*V_T[i-1]*P_T[i-1] # prey␣
↪equation
    P_T[i] = P_T[i-1]*(1 - d + b*V_T[i-1]) # predator equation

  fig, ax = plt.subplots()
  ax.plot(V_T, label = "Prey")
  ax.plot(P_T, label = "Predator")
  ax.set_xlabel("Time", fontsize = 15)
  ax.set_ylabel("Population size", fontsize = 15)
  ax.legend()

  plt.show()
```

Now, we will create a dictionary with the parameters for our function, which include the initial conditions and the parameters.

[24]:
```
params_set1 = {"Init_V": 1, "Init_P": 2, "R":0.1, "K":100, "a":0.1, "b":0.1, "d"␣
↪:0.1, "MaxTime": 200}

plot_lotka_volterra(**params_set1)
```



[25]:
```
# we can then test different parameter sets to see our output
# try changing some values in the dictionary
# as long as they are positive, it should give a reasonable outcome (there are␣
 ↪some other limitations)

params_set2 = {"Init_V": 1, "Init_P": 1, "R":0.1, "K":100, "a":0.3, "b":0.2, "d"␣
 ↪:0.1, "MaxTime": 200}
plot_lotka_volterra(**params_set2)
```



13 Extra coding check in

The most basic set of differential euqations used by disease ecologists (aka Katie) are SIR (susceptible, infected, recovered) equations. Most of the population starts out susceptible to infection, and can move to the infected class by interacting with an infected person. The infected class then recovers at some rate, and moves to the recovered class. In this case, there is no disease or population mortality.

The discrete set of equations are as follows, where β is the transmission rate, and γ is the recovery rate:

$$S_t = S_{t-1} - \beta S_{t-1} I_{t-1}$$
$$I_t = I_{t-1} + \beta S_{t-1} I_{t-1} - \gamma I_{t-1}$$
$$I_t = R_{t-1} + \gamma I_{t-1}$$

In a simliar fashion to the Lotka-Volterra function above, simulate an epidemicusing the SIR equations and then plot the output of that function. I will write down the equations for you to get you started, since this is the first way to run into trouble.

Initial conditions: * S_init = 0.99 * I_init = 0.01 * R_init = 0

Parameters: * beta = 0.5 * gamma = 0.2 * MaxTime = 100

Advanced: Try it out by writing the equations by yourself

[26]:
```
### This is where you can write your answer to the check in
```

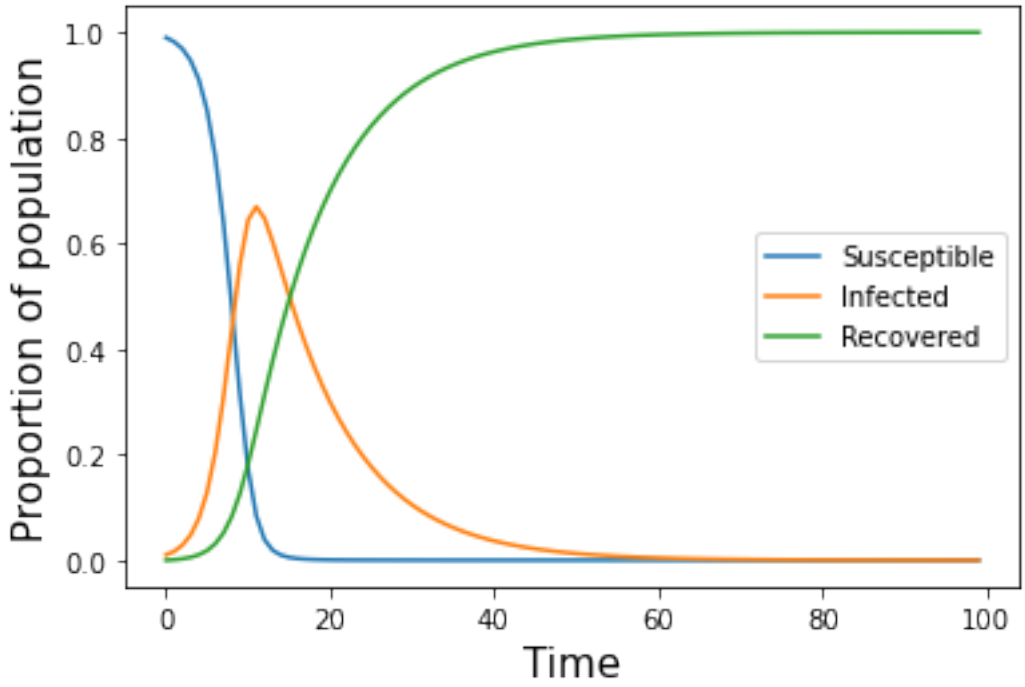
13.1 Check in equations

[ ]:
```
for i in range(1,MaxTime,1):
  S[i] = S[i-1] - beta*S[i-1]*I[i-1] # susceptible equation
  I[i] = I[i-1] + beta*S[i-1]*I[i-1] - gamma*I[i-1] # infected equation
  R[i] = R[i-1] + gamma*I[i-1]
```

13.2 Check in answer

[27]:
```
def plot_SIR(Init_S, Init_I, Init_R, beta, gamma, MaxTime):

  S = [0]*MaxTime # initialize a vector that is 200 elements long of zeros
  I = [0]*MaxTime
  R = [0]*MaxTime

  S[0] = Init_S # setting the first value to the initial conditions
  I[0] = Init_I
  R[0] = Init_R

  for i in range(1,MaxTime,1):
    S[i] = S[i-1] - beta*S[i-1]*I[i-1] # susceptible equation
    I[i] = I[i-1] + beta*S[i-1]*I[i-1] - gamma*I[i-1] # infected equation
    R[i] = R[i-1] + gamma*I[i-1]

  fig, ax = plt.subplots()
  ax.plot(S, label = "Susceptible")
  ax.plot(I, label = "Infected")
  ax.plot(R, label = "Recovered")

  ax.set_xlabel("Time", fontsize = 15)
  ax.set_ylabel("Proportion of population", fontsize = 15)
  ax.legend()

  plt.show()
```

[28]:
```
params_set1 = {"Init_S": 0.99, "Init_I": 0.01, "Init_R":0, "beta":0.8, "gamma":0.
 ↪1, "MaxTime":100}
plot_SIR(**params_set1)
```



Appendix 1: Resources- things that are not Matplotlib

There are lots of other types of plots you can do in `Matplotlib`, here is a sample of other options for you to look at.

Altair for cool, fancy plots

One of the other common plotting packages is `Altair`. We can import simple datasets from `vega_datasets`, just to get an idea of what is possible. Later in the course, we will be going over more complex plotting with `pandas` dataframes.

[ ]:
```
pip install vega_datasets
```

[ ]:
```
pip install altair vega_datasets
```

[29]:
```
from vega_datasets import data as vgd
import altair as alt
```

This is an example that takes stock information from `vega_datasets` and plots it on a simple interactive plot.

[30]:
```
stocks = vgd.stocks()
### Time series plot
alt.Chart(stocks).mark_line().encode(
```

```
    x = 'date:T',   # :T encodes this as a 'temporal' data. :N == discrete␣
↪unordered, :Q == continous, :O == discrete ordered !!! Encodings affect plot␣
↪aesthetics !!!
    y = 'price',
    color = 'symbol'
).interactive()
```

[30]: `alt.Chart(...)`

Here's an example of plotting seattle weather data.

[31]:
```
### complicated example ###
source = vgd.seattle_weather()

scale = alt.Scale(domain=['sun', 'fog', 'drizzle', 'rain', 'snow'],
                  range=['#e7ba52', '#a7a7a7', '#aec7e8', '#1f77b4', '#9467bd'])
color = alt.Color('weather:N', scale=scale)

# We create two selections:
# - a brush that is active on the top panel
# - a multi-click that is active on the bottom panel
brush = alt.selection_interval(encodings=['x'])
click = alt.selection_multi(encodings=['color'])

# Top panel is scatter plot of temperature vs time
points = alt.Chart().mark_point().encode(
    alt.X('monthdate(date):T', title='Date'),
    alt.Y('temp_max:Q',
        title='Maximum Daily Temperature (C)',
        scale=alt.Scale(domain=[-5, 40])
    ),
    color=alt.condition(brush, color, alt.value('lightgray')),
    size=alt.Size('precipitation:Q', scale=alt.Scale(range=[5, 200]))
).properties(
    width=550,
    height=300
).add_selection(
    brush
).transform_filter(
    click
).interactive()

# Bottom panel is a bar chart of weather type
bars = alt.Chart().mark_bar().encode(
    x='count()',
    y='weather:N',
    color=alt.condition(click, color, alt.value('lightgray')),
).transform_filter(
```

```
    brush
).properties(
    width=550,
).add_selection(
    click
)

alt.vconcat(
    points,
    bars,
    data=source,
    title="Seattle Weather: 2012-2015"
)
```

[31]:
```
alt.VConcatChart(...)
```

[32]:
```
stocks
```

[32]:
```
    symbol       date   price
0     MSFT 2000-01-01   39.81
1     MSFT 2000-02-01   36.35
2     MSFT 2000-03-01   43.22
3     MSFT 2000-04-01   28.37
4     MSFT 2000-05-01   25.45
..     ...        ...     ...
555   AAPL 2009-11-01  199.91
556   AAPL 2009-12-01  210.73
557   AAPL 2010-01-01  192.06
558   AAPL 2010-02-01  204.62
559   AAPL 2010-03-01  223.02

[560 rows x 3 columns]
```