CS 237 Homework 01 -- Data Display in Python

Due Thursday at 11:59 PM in Gradescope (with grace period of 6 hours)

In this first homework, you will become familiar with various methods of displaying the results of probability experiments graphically in Python using Jupyter notebooks and Matplotlib. This will be a fundamental way of understand the results of experiments throughout the course. We will cover:

- Basic introduction to Python and Jupyter Notebook.
- Basic introduction to Matplotlib and graphing
- Basic introduction to Monte Carlo (probability) simulation.

Please follow the HW Submission Instructions (http://www.cs.bu.edu/fac/snyder/cs237/HWSubmissionInstructions.html); any homeworks not following these requirements may be penalized.

Anaconda Distribution of Python

- Download from https://www.continuum.io/downloads (https://www.continuum.io/downloads)
  - Make sure it's Python 3!
  - Be sure to pick the correct Operating System (i.e., Windows, MAC OS, Linux)

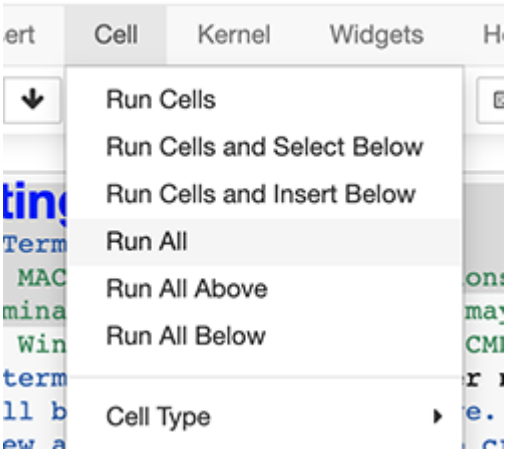
Starting Jupyter

- Open a Terminal Window
  - For MAC OS, open your Applications folder, then open the Utilities folder. Open the Terminal application. (Or, type "terminal" into Spotlight.) You may want to add this to your dock.
  - For Windows, simply search for CMD and run the result
- In the terminal run `jupyter notebook`
- This will bring you to the home page.
- Click new and then click python3 to create a new Ipython3 notebook.
  - Alternatively, use the Anaconda Navigator that will be installed in your Applications folder.

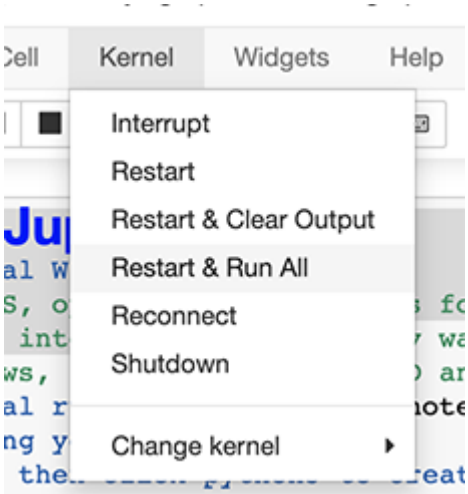
Downloading the Assignments

- From the class web page download HW01 as an IPYNB file (NOT a .txt file):

- As soon as you download it, Run All the cells:



- Normally, you select one or more cells and then start execution using Cell -> Run Cells, but it is easier to use the keyboard shortcut: Click inside the cell to select it, then type Control-Return.
- Use the Kernel menu to kill a runaway piece of code, and if weird stuff is happening and you want to reset everything, select Kernel -> Restart and Run All:



In [1]:

```
# Here are some imports which will be used in the code in the rest of the lab

# Imports used for the code in CS 237

import numpy as np                # arrays and functions which operate on array
import matplotlib.pyplot as plt   # normal plotting
import seaborn as sns             # Fancy plotting
import pandas as pd               # Data input and manipulation

from numpy.random import seed, randint, random
from collections import Counter

%matplotlib inline
```

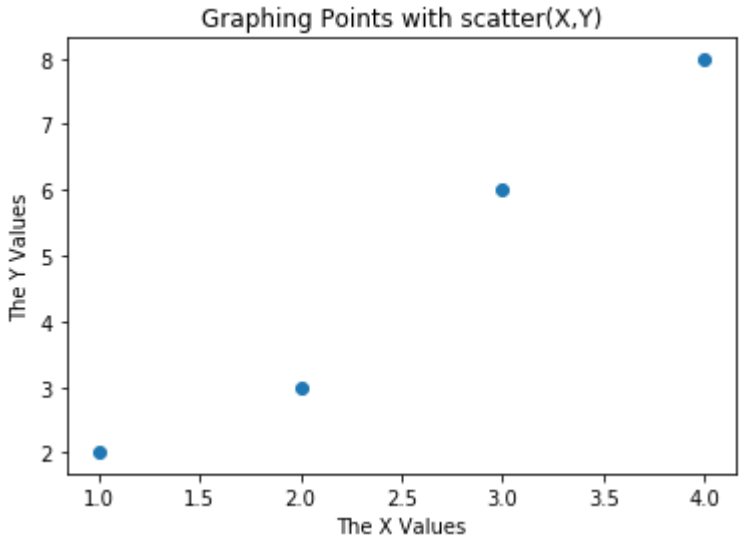
Plotting Points

The `scatter(...)` function is used to plot points from a list of x values and the associated y values.

In [2]:

```
# To plot the points (1,2), (2,3), (3,6), (4,8) we would list the x values and t
he corresponding y values:
X = [1,2,3,4]
Y = [2,3,6,8]

print("\nThis is the list of points:",list(zip(X,Y)))
print("They must be input to the function as separate lists:")
print("\tX =",X)
print("\tY =",Y,"\n")
plt.scatter(X,Y)
plt.title('Graphing Points with scatter(X,Y)')
plt.xlabel("The X Values")
plt.ylabel("The Y Values")
plt.show()
```

```
This is the list of points: [(1, 2), (2, 3), (3, 6), (4, 8)]
They must be input to the function as separate lists:
        X = [1, 2, 3, 4]
        Y = [2, 3, 6, 8]
```

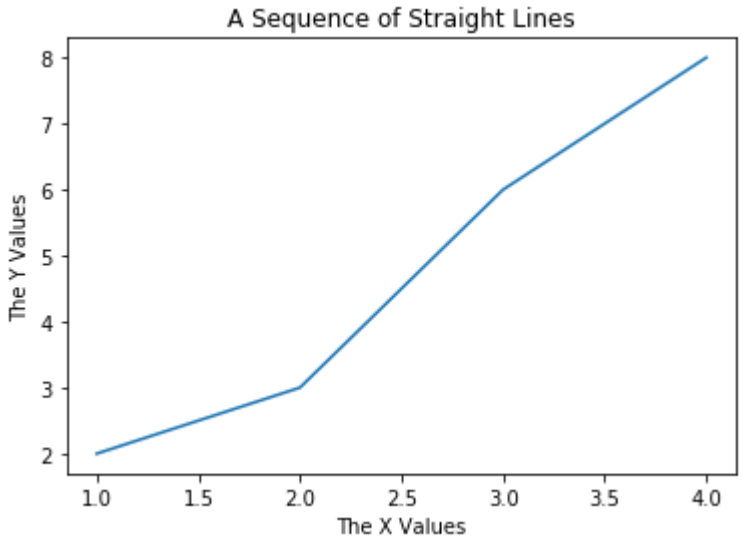


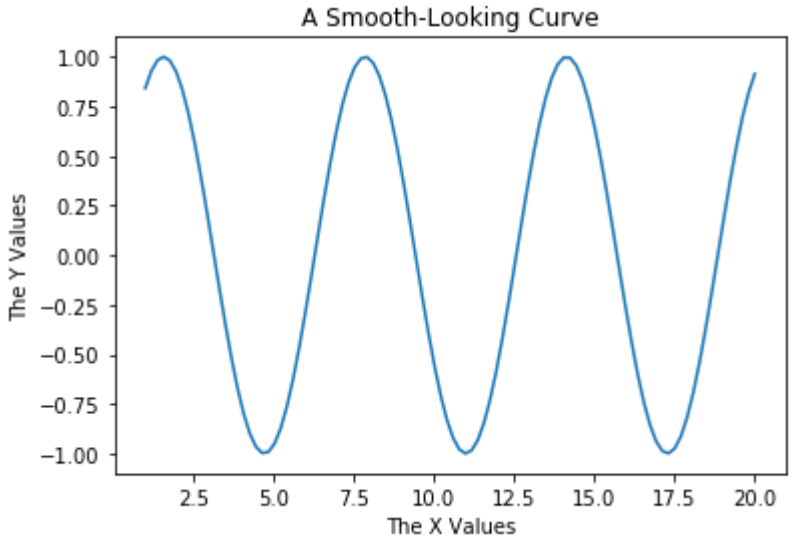
Plotting Lines and Curves

If you call `plot(...)` instead of `scatter(...)` you will display a curve created by connecting the points with straight lines. Essentially you can only plot straight lines between points, but if the points are close together, you will not notice, and it will look like a smooth curve.

In [3]:

```
# To plot a curve through the points (1,2), (2,3), (3,6), (4,8) we would use:
plt.plot([1,2,3,4], [2,3,6,8])
plt.title('A Sequence of Straight Lines')
plt.xlabel("The X Values")
plt.ylabel("The Y Values")
plt.show()

X = np.linspace(1,20,100)           # returns a list of 100 equally-spaced valu
es in the range [1..20]
Y = [np.sin(x) for x in X]
plt.plot(X,Y)
plt.title('A Smooth-Looking Curve')
plt.xlabel("The X Values")
plt.ylabel("The Y Values")
plt.show()
```

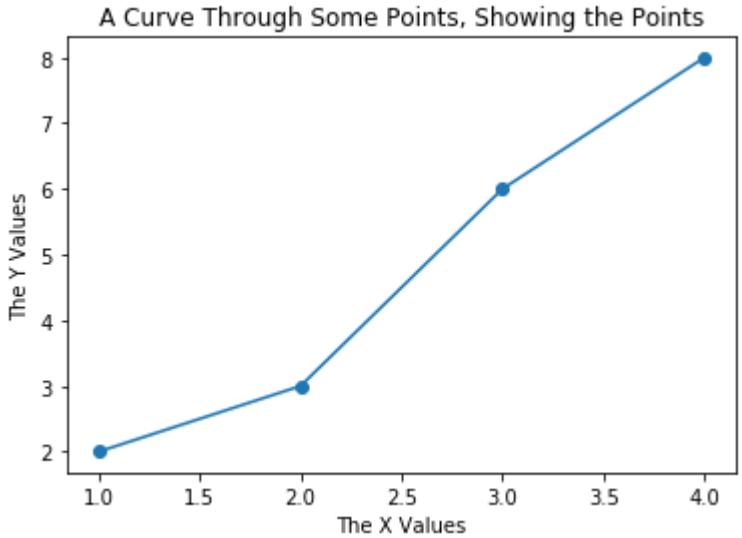




If you want to do both, you can simply call both functions before you call show().

In [4]:

```
plt.scatter([1,2,3,4], [2,3,6,8])
plt.plot([1,2,3,4], [2,3,6,8])
plt.title('A Curve Through Some Points, Showing the Points')
plt.xlabel("The X Values")
plt.ylabel("The Y Values")
plt.show()
```

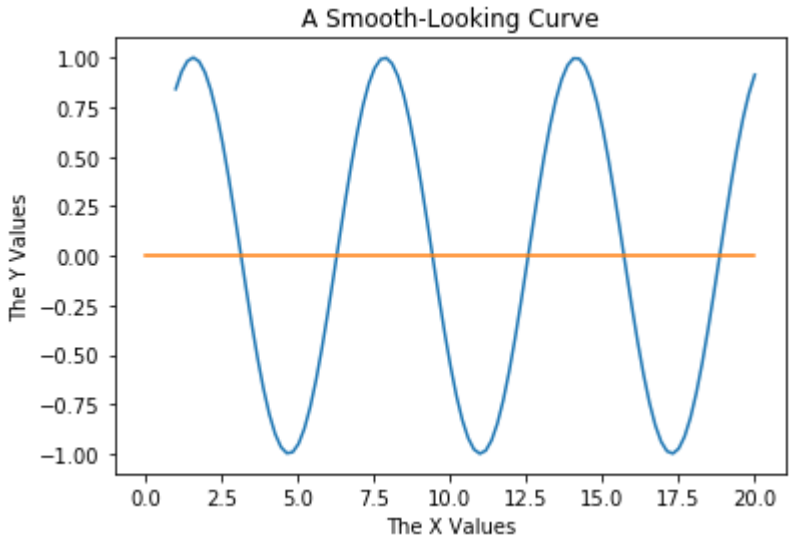


If you want to draw a single line from $(\mathscr{X}_1, \exists_1)$ to $(\mathscr{X}_2, \exists_2)$ you can plot $[\mathscr{X}_1, \mathscr{X}_2]$ and $[\exists_1, \exists_2]$.

Here we have added a zero line to our sin curve:

In [5]:

```
X = np.linspace(1,20,100)           # returns a list of 100 equally-spaced valu
es in the range [1..20]
Y = [np.sin(x) for x in X]
plt.plot(X,Y)
plt.plot([0,20],[0,0])
plt.title('A Smooth-Looking Curve')
plt.xlabel("The X Values")
plt.ylabel("The Y Values")
plt.show()
```



For further details on drawing plots, particularly on color and format, see the Appendix at the end of this document

Problem Zero: Using the Numpy Random Library

We have imported a number of functions from the Numpy Random library, which you can read about here (https://docs.scipy.org/doc/numpy/user/quickstart.html).

In [6]:

```
# Run this cell several times and and see what happens

random()
```

Out[6]:

```
0.1343104923008508
```

In [7]:

```
# Run this cell several times and and see what happens
# List comprehensions are your best friend, learn how to use them!!!

size = 10
rands = [ random() for k in range(size) ]
rands
```

Out[7]:

```
[0.32236802911024376,
 0.1938382268897476,
 0.8664186176922196,
 0.39480920772432304,
 0.8536048306540535,
 0.5559868320205608,
 0.09767311322489713,
 0.5433670169791758,
 0.17376218611282102,
 0.9255498358389846]
```

In [8]:

```
# Run this cell several times and and see what happens
# This produces a Numpy 1D vector, which is
# another way to store a sequence.

size = 5
random(size)
```

Out[8]:

```
array([0.6125342 , 0.91621057, 0.78499746, 0.14862263, 0.59883973])
```

In [9]:

```
# In most cases, arrays are interchangeable with lists, but if you need a Python
# list, just do this:

list(random(size))
```

Out[9]:

```
[0.44987236572493017,
 0.9724258325249178,
 0.6519244820167338,
 0.9014209389647188,
 0.6747711903087755]
```

In order to make grading easier, we will "seed" the random number generation so that it always produces the same pseudo-random sequence. However, you should generally try running your code several times without the seed, trying it on various random sequences.

Just be sure to include the seed before running your program to submit, so that the graders can see your correct results.

In [10]:

```
# Run this cell several times, and see what happens
# How is it different from the result in cell [210]?

seed(0)
random(5)
```

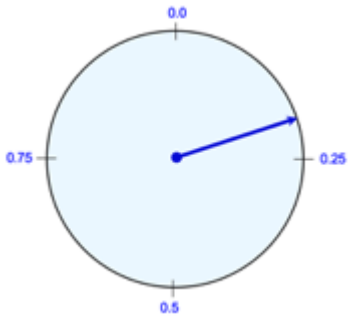
Out[10]:

```
array([0.5488135 , 0.71518937, 0.60276338, 0.54488318, 0.4236548 ])
```

Problem 1 (Plotting Points)

Part A

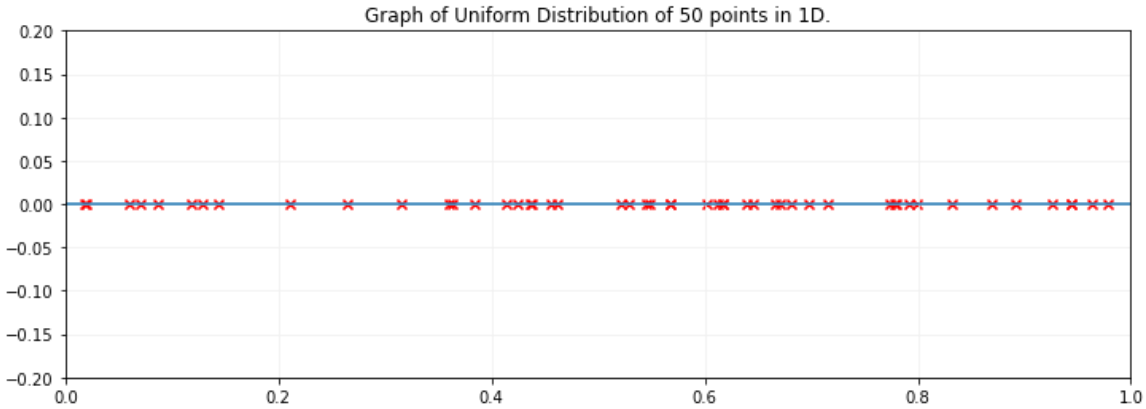
One of our standard examples in lecture will be a "spinner" which can be set in motion to randomly chose a real number in the half-open interval [0..1):



We will simulate this using the Python function `random()` from the Numpy.random library we imported above.

In [11]:

```
def random_line_plot(num_trials):
    x_vals =  [random() for k in range(num_trials)]
    y_vals =  [0 for k in range(num_trials)]
    plt.figure(figsize=(12, 4))
    plt.title('Graph of Uniform Distribution of '+str(num_trials)+' points in 1
D.',fontsize=12)
    plt.grid(color='0.95')
    plt.ylim(-0.2, 0.2)
    plt.xlim(-0.0,1.0)
    plt.plot([0,1.0],[0,0])
    plt.scatter(x_vals, y_vals, marker="x",color="r")
    plt.show()

seed(0)
random_line_plot(50)
```

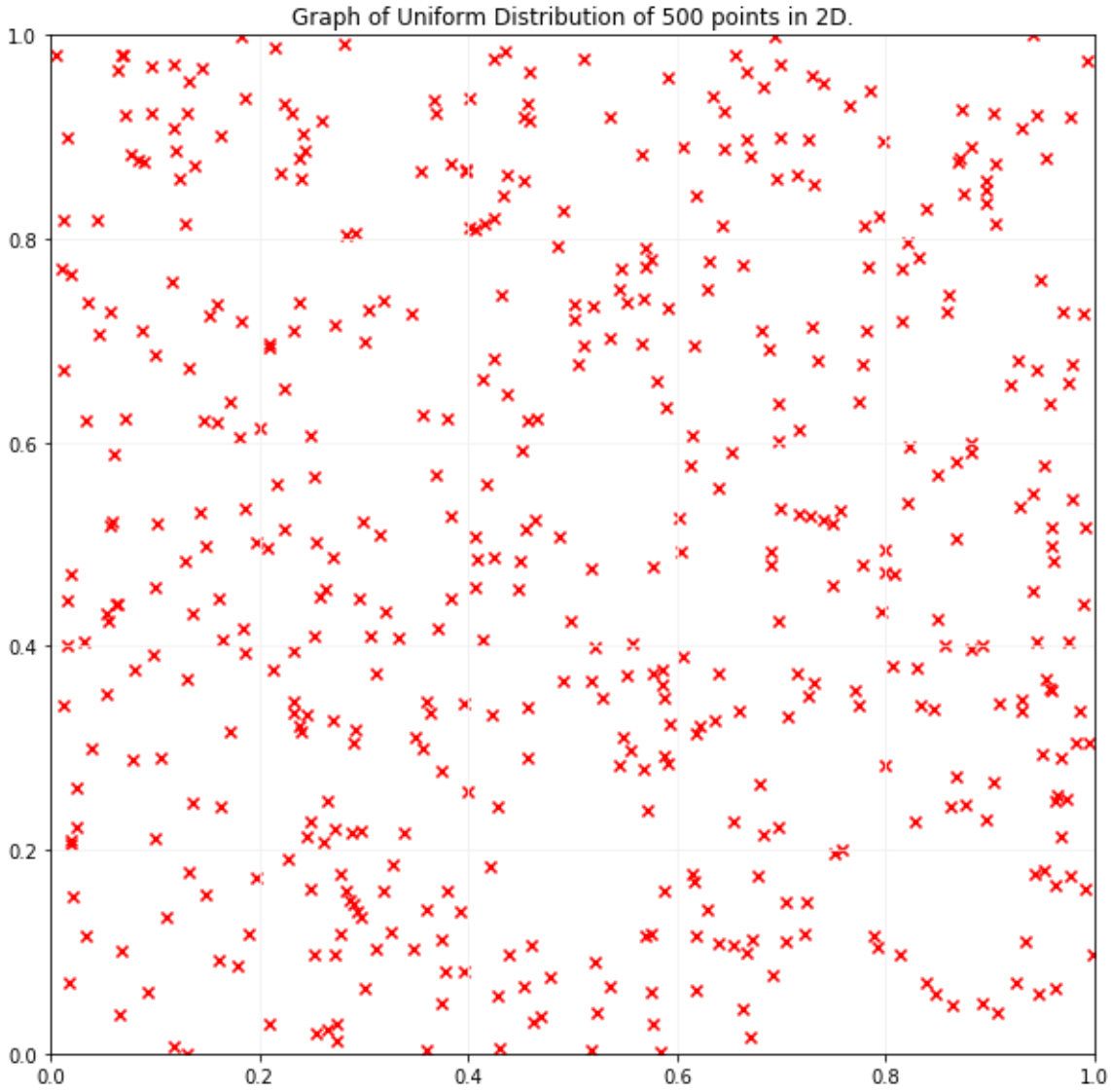


Part B

Now we will simulate the experiment of throwing a dart at a unit square, which will produce a scatter plot of random points in a 2D grid.

In [12]:

```
def random_plane_plot(num_trials):
    """Uses numpy's random function to build a list of x and y values with the f
ollowing properties:
    - x_vals should have len num_trials
    - y_vals should have len num_trials
    - Each value in x and y should be between 0 and 1
    HINT: use the same code to build x_vals and y_vals
    """
    x_vals = [random() for k in range(num_trials)]
    y_vals = [random() for k in range(num_trials)]
    plt.figure(num=None, figsize=(10, 10))
    plt.title('Graph of Uniform Distribution of '+str(num_trials)+' points in 2
D.',fontsize=12)
    plt.grid(color='0.95')
    plt.ylim(0, 1)
    plt.xlim(0,1)
    plt.scatter(x_vals, y_vals,marker="x",color="r")
    plt.show()

seed(0)
random_plane_plot(500)
```



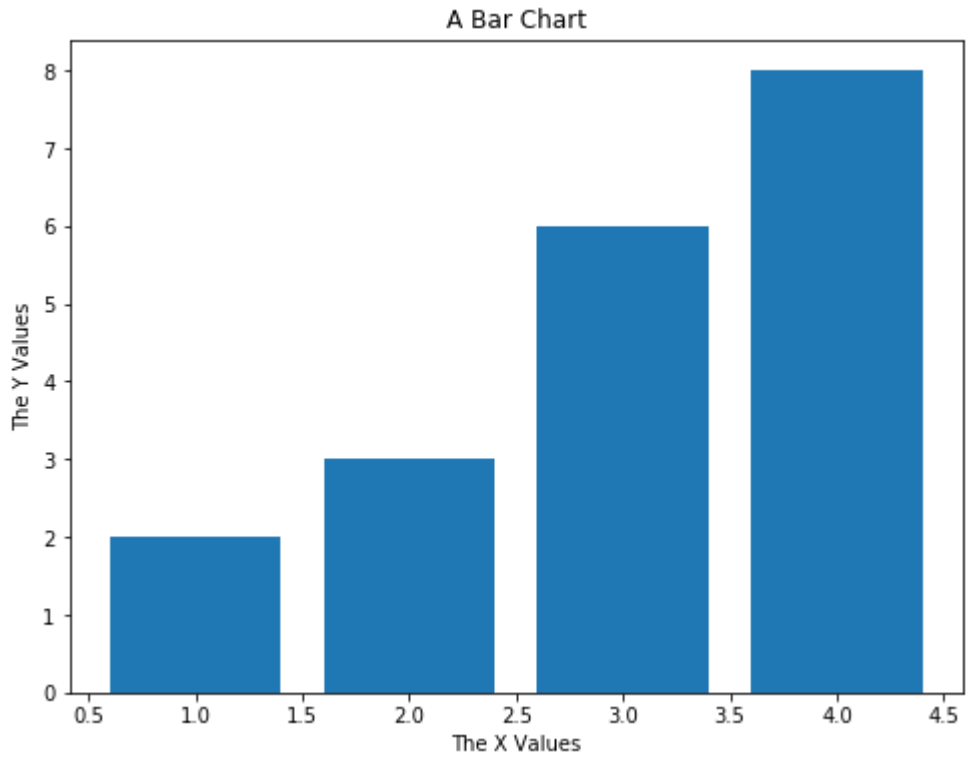
[End of Problem One]

Bar Charts

If we do the exact same thing as we did with a simple plot, but use the function `bar(...)` we get a bar chart:

In [13]:

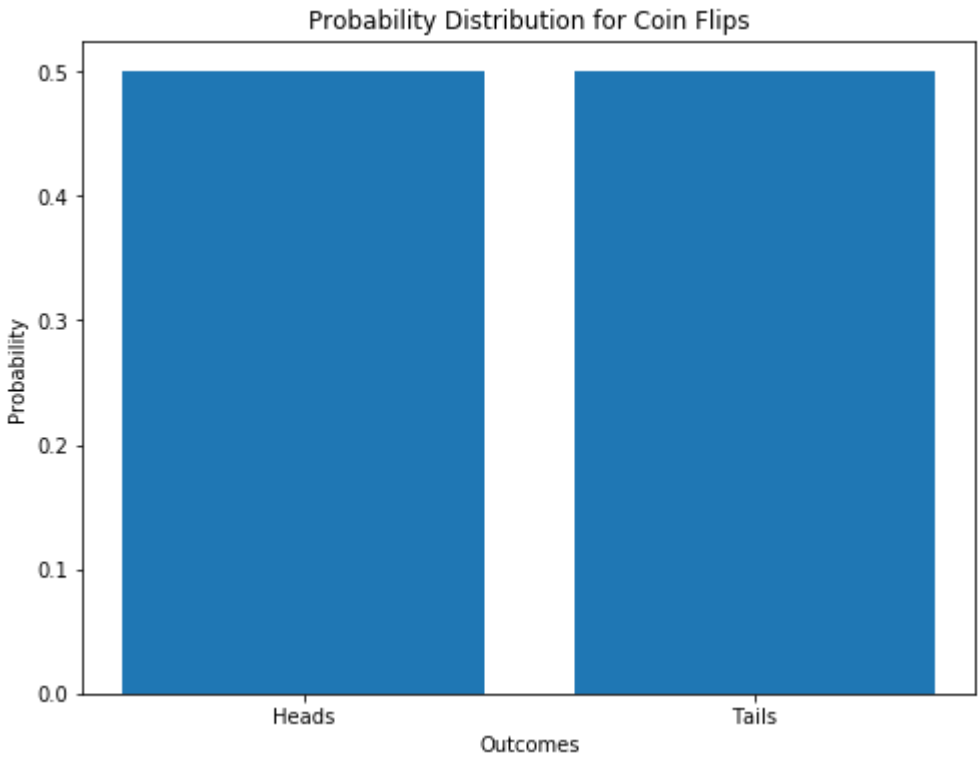
```
# To plot the points (1,2), (2,3), (3,6), (4,8) we would list the x values and t
he corresponding y values:
plt.figure(num=None, figsize=(8, 6))
plt.bar([1,2,3,4], [2,3,6,8])
plt.title('A Bar Chart')
plt.xlabel("The X Values")
plt.ylabel("The Y Values")
plt.show()
```



If the Y axis is probabilities (in the range 0 .. 1), we get a distribution of the probabilities among the outcomes of an experiment:

In [14]:

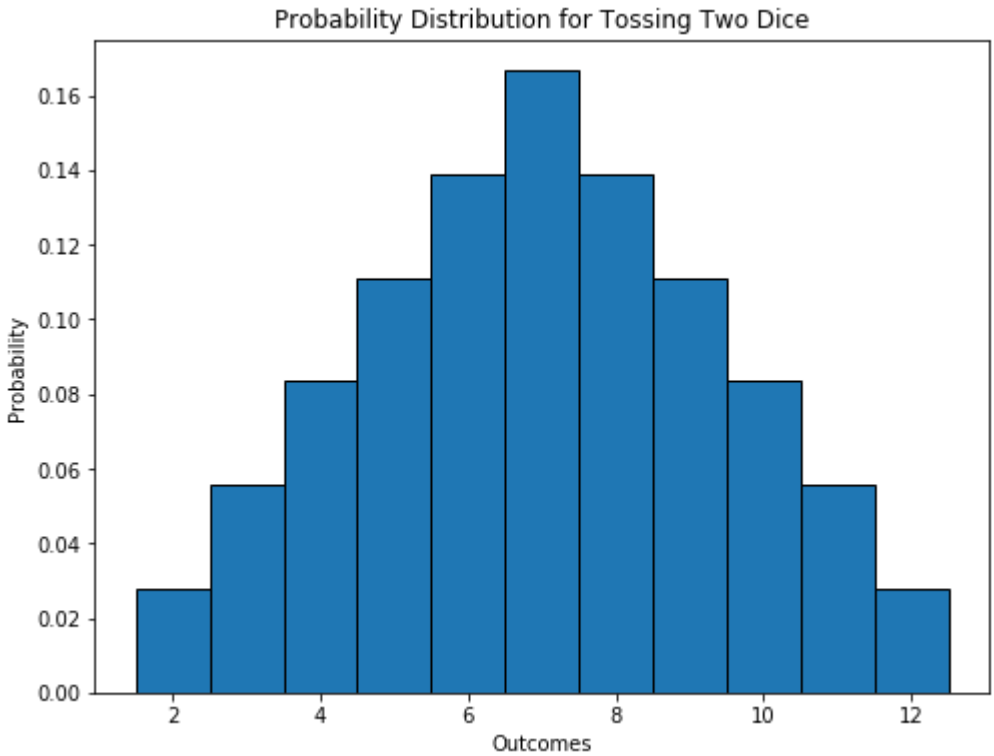
```
# Show the distribution of probabilities for a coin flip:
x = [0,1]
y = [0.5, 0.5]
labels = ['Heads', 'Tails']

plt.figure(num=None, figsize=(8, 6))
plt.xticks(x, labels)
plt.bar(x,y)
plt.title('Probability Distribution for Coin Flips')
plt.ylabel("Probability")
plt.xlabel("Outcomes")
plt.show()
```



With a few tweaks, you can create an attractive bar chart for arbitrary probability distributions (we will cover the topic of distributions in a few weeks).

In [15]:

```
# Show the distribution of probabilities for flipping two dice
# We will consider this example in lecture

x = [k for k in range(2,13)]
y = [1/36,2/36,3/36,4/36,5/36,6/36,5/36,4/36,3/36,2/36,1/36]

plt.figure(num=None, figsize=(8, 6))
plt.bar(x,y, width=1.0,edgecolor='black')    # <--- Note how we set the width an
d edge color
plt.title('Probability Distribution for Tossing Two Dice')
plt.ylabel("Probability")
plt.xlabel("Outcomes")
plt.show()
```

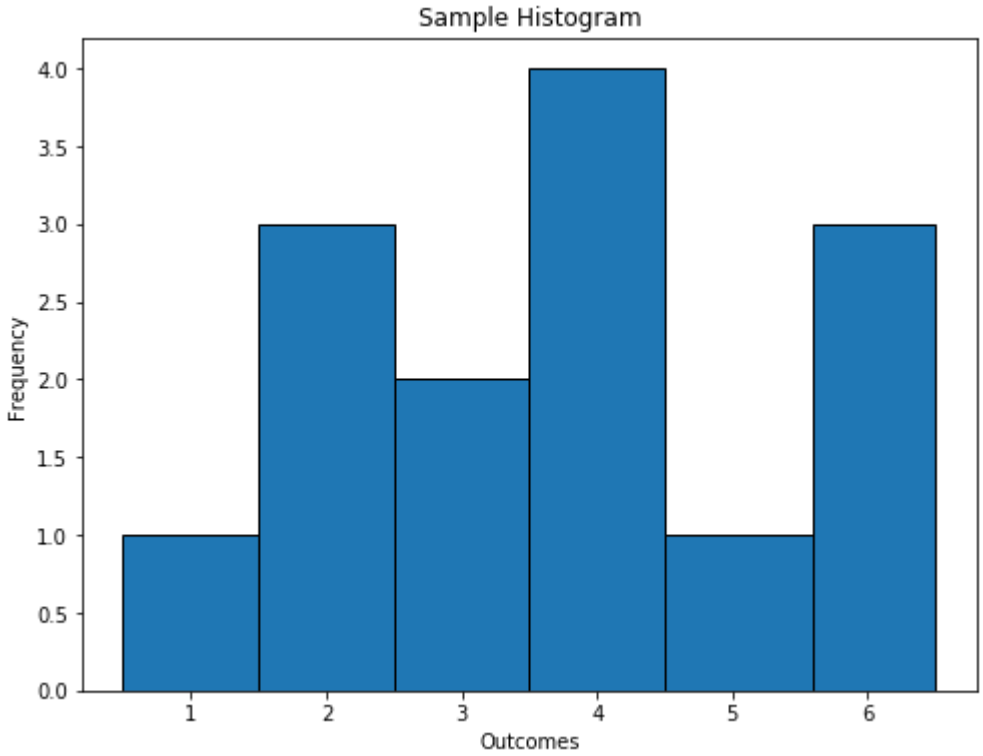


Histograms

- If you give a list of values to `hist(...)` it will create a histogram counting how many of each value occur; this list can be unordered;
- You will get a cleaner display if you specify where the edges of the bins are, and make sure the edges of the bins are visible, as shown in this example:

In [16]:

```
plt.figure(num=None, figsize=(8, 6))
plt.hist([1,2,4,2,6,2,4,5,6,4,6,3,4,3],bins=[0.5,1.5,2.5,3.5,4.5,5.6,6.5],edgeco
lor='black')
plt.title('Sample Histogram')
plt.xlabel("Outcomes")
plt.ylabel("Frequency")
plt.show()
```



Problem 2

Read and understand the function `role_die(...)` below, which simulates the experiment of rolling a fair, six-sided die `num_trials` times. Note carefully:

- The sample space is $\{1, 2, 3, 4, 5, 6\}$.
- The experiment is equi-probable, i.e., the probability of any particular outcome is $\frac{1}{6}$.
- If we record the outcome for a large number of experiments, we would expect the number of outcomes to be "evenly distributed." In other words, for a large number of trials, we would expect the probability of each outcome ™$\in \{1, 2, 3, 4, 5, 6\}$ to be

$$\frac{\text{number of times we observed a value ™}}{\texttt{num\_trials}} \approx \frac{1}{6}$$

TO DO: For this first problem, simply provide the Python code which would display a **histogram** of the results of the experiment for 100,000 trials with appropriate labels. You should use the Numpy function `randint(...)`, as shown in the next cell.

In [17]:

```
# Demo of randint(lo,hi), which generates a random integer from the sequence
# [lo,lo+1,...,hi-1].

# In other words the upper bound is exclusive, to be consistent with the indices
in lists,
# the Python range function, and so on.

randint(0,4)
```

Out[17]:

```
1
```

In [18]:

```
# You can also ask it for an array, since it is a Numpy function:

randint(0,4,10)
```

Out[18]:

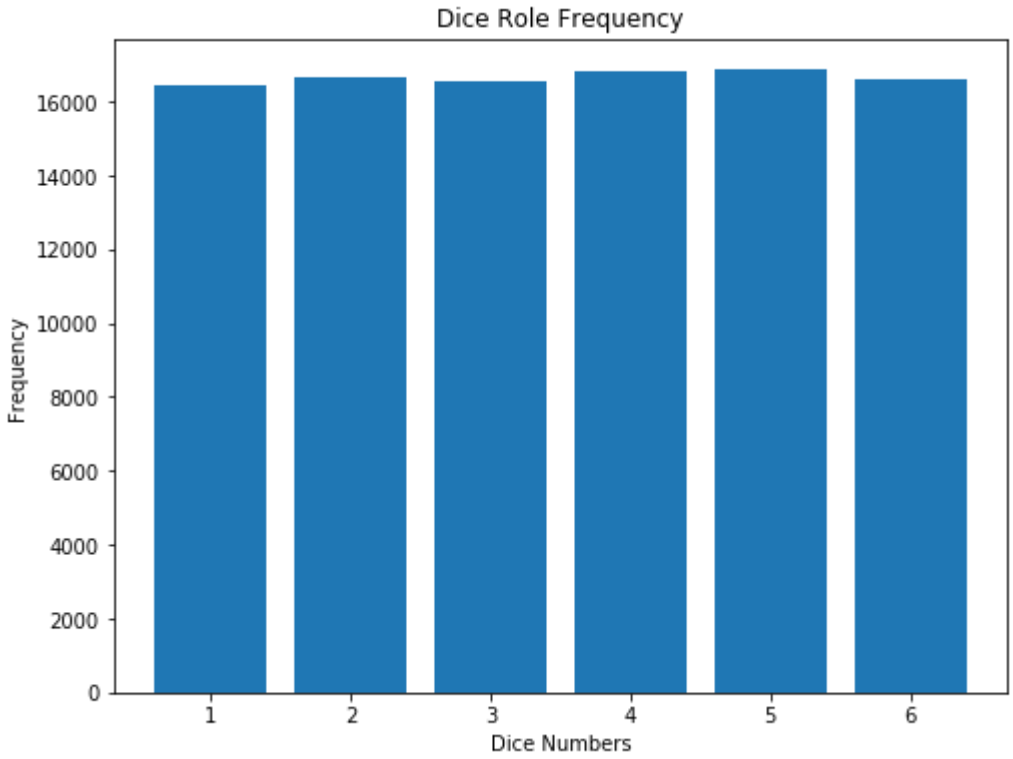
```
array([1, 2, 0, 2, 1, 0, 0, 0, 3, 2])
```

In [19]:

```
seed(0)

def roll_die(num_trials = 100000):         #  <-- note the use of the default pa
rameter value
    trials_and_results = randint(1,7,num_trials) # this creates a 1D array of le
ngth num_trials of random integers 1..6
    return trials_and_results

example_trials = roll_die()                    # <-- now we don't have to specify
unless different
```

In [20]:

```
plt.figure(num=None, figsize=(8, 6))
one = 0
two = 0
three = 0
four = 0
five = 0
six = 0
for x in example_trials:
    if x == 1:
        one += 1
    if x == 2:
        two += 1
    if x == 3:
        three += 1
    if x == 4:
        four += 1
    if x == 5:
        five += 1
    if x == 6:
        six += 1
plt.bar([1,2,3,4,5,6], [one,two,three,four,five,six])
plt.title('Dice Role Frequency')
plt.xlabel("Dice Numbers")
plt.ylabel("Frequency")
plt.show()
```



Problem 3

Now we will display the same results showing the distribution of probabilities, instead of an explicit histogram; since the experiment is equi-probable, if we record the outcome for a large number of experiments, we would expect the number of outcomes to be "evenly distributed." In other words, for a large number of trials, we would expect the probability of each outcome ™∈ {1, 2, 3, 4, 5, 6} to be

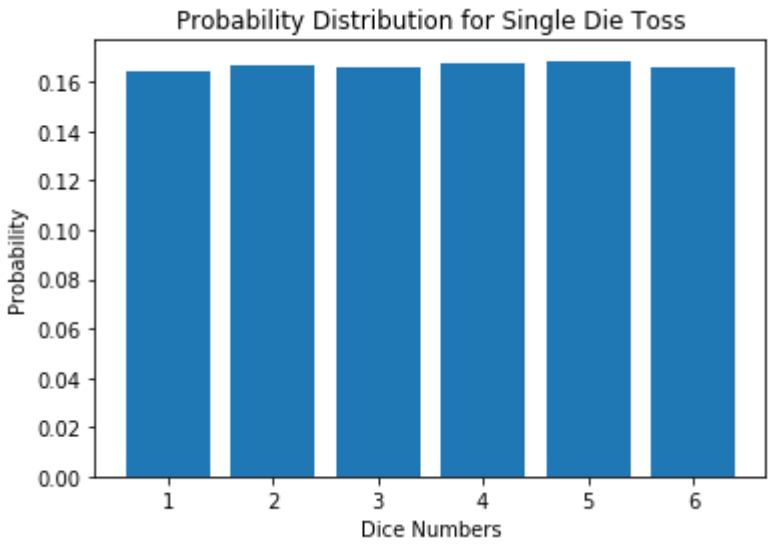
$$\frac{\text{number of times we observed a value ™}}{\texttt{num\_trials}} \approx \frac{1}{6}$$

- To calculate the probabilities, you will need to count the number of occurrences of each of the outcomes, you may find the function `Counter(...)` useful for this (Google "Numpy Counter" to find out how this works);
- Once you have the frequency of each outcome, divide by the total number of trials to get the probability for each.

TO DO: Complete the function stub below which takes the list returned by `roll_die(...)`, or any other experiment returning numerical results, and produces a frequency distribution; this should have the same shape as the histogram, but the Y axis will be probabilities instead of the frequency. Again, create appropriate labels. Demonstrate your function, again, on the list `example_trials` produced in Problem 1.

In [21]:

```
# Solution

seed(0)

def show_distribution(outcomes, title='Probability Distribution'):
    lst = []
    lengthT = 0
    for x in outcomes:
        lengthT += 1
        if x not in lst:
            lst.append(x)
    lst.sort()
    size = 0
    for x in lst:
        size += 1
    lstTwo = [0] * size
    for x in outcomes:
        len = 0
        for y in lst:
            if x == y:
                lstTwo[len] += 1
                break
            else:
                len += 1
    val = 0
    for x in lstTwo:
        lstTwo[val] = x/lengthT
        val = val + 1

    plt.bar(lst, lstTwo)
    plt.title(title)
    plt.xlabel("Dice Numbers")
    plt.ylabel("Probability")
    plt.show()

show_distribution(example_trials,title='Probability Distribution for Single Die
 Toss')
```



Motivation for Monte Carlo Simulation

For the case of a fair die, the distribution is very easily computed by hand. But in general, it may be difficult to write down an analytical solution for the distribution produced by a random experiment. This is where simulation comes into play: instead of mathematically computing the distribution explicitly, you can use this method of repeating experiments, and recording outcomes to understand the probabilistic rules governing some real world event. When you can come up with an analytical result, this is a nice way of confirming its correctness!

Problem 4

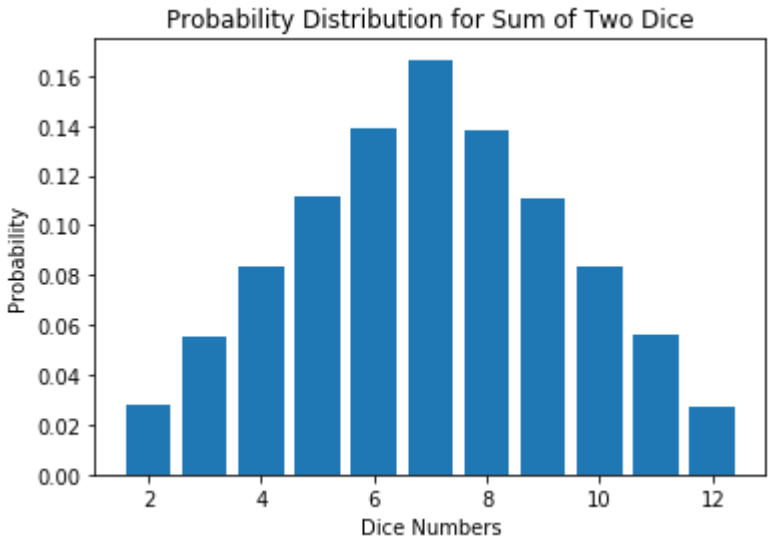
You will now do the same thing you did in the previous problems, but with a new experiment: instead of rolling one die and recording the value, you will simulate rolling Ωdies and recording their sum. For example, if $\Omega = 2$ and the first die shows up as a 3, and the second die shows up as a 1, the sum (and the value we record) would be 4.

TO DO: Complete the two function stubs below and then demonstrate by providing code which would print out the probability distribution for rolling 2 dice 100,000 times.

Hint: Not required, but think about how you might do this in one line using Numpy and list comprehensions.

In [22]:

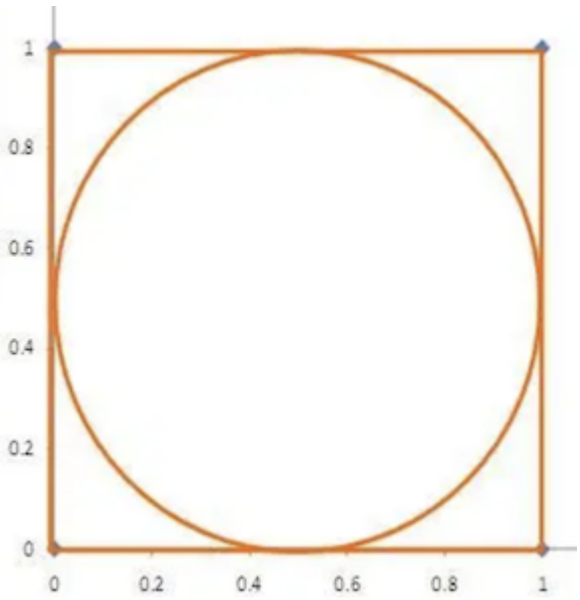
```
# Solution

seed(0)


def roll_and_add_dice(num_dice, num_trials = 10**6):
    lst = []
    sum = 0
    for x in range(num_trials):
        trial = randint(1,7,num_dice)
        for x in trial:
            sum += x
        lst.append(sum)
        sum = 0
    return lst


show_distribution(roll_and_add_dice(2),title='Probability Distribution for Sum o
f Two Dice')
```



Problem 5 (Monte Carlo Calculation of ∀)

This final problem is also a Monte Carlo simulation, but this time in the continuous domain: we will calculate the value of ∀ using a variation of Problem 1B.

We will leave this one up to you as it the exact details, but you must use the following fact: a circle inscribed in a unit square (i.e., with side of length and area 1.0):



has as radius of 0.5 and an area of $\forall * (0.5^2) = \frac{\forall}{4}$.

Therefore, if you generate `num_trials` random points in the unit square, as in Problem 1B, and count how many land inside the circle, you can calculate an approximation of ∀.

For this problem, you must

(A) Draw the diagram of the unit square with inscribed circle and 500 random points, and calculate the value of ∀.

(B) Without drawing the diagram, calculate the value of ∀ you would get from 10^5 trials.

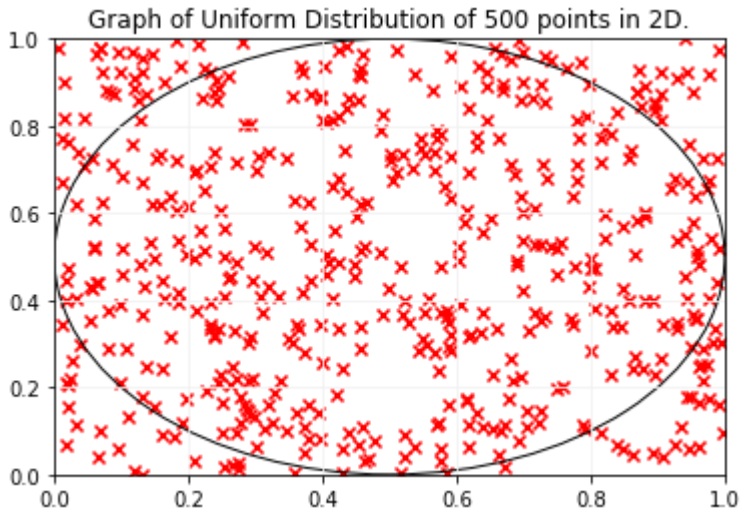
(C) After completing (B), try to get a more accurate value for ∀ by increasing the number of trials. Your results will depend on your machine, but for comparison, with my new Macbook Pro, I ran it with 10^8 trials while I got a cup of coffee, and it had the answer correct to 3 decimal places when I came back. Sometimes I run big experiments overnight! The key here is to try increasingly large numbers and see how the time increases.

</blockquote>
Hint: Start by copying your code from Problem 1B. You might find Mr. Pythagoras's formula useful.

In [23]:

```
def random_plane_plot(num_trials):
    x_vals = [random() for k in range(num_trials)]
    y_vals = [random() for k in range(num_trials)]
    plt.figure(num=None, figsize=(10, 10))
    circle = plt.Circle((0.5, 0.5), 0.5, color = 'black', fill = False)
    fig, ax = plt.subplots()
    ax.add_artist(circle)
    plt.title('Graph of Uniform Distribution of '+str(num_trials)+' points in 2
D.',fontsize=12)
    plt.grid(color='0.95')
    plt.ylim(0, 1)
    plt.xlim(0,1)
    plt.scatter(x_vals, y_vals,marker="x",color="r")
    plt.show()
    number = 0
    for x in range(num_trials):
        if (np.sqrt(x_vals[x]**2 + y_vals[x]**2)<= 1):
            number += 1
    value = number * 4 /num_trials
    return value


seed(0)
random_plane_plot(500)
```

```
<Figure size 720x720 with 0 Axes>
```



Out[23]:

```
3.128
```

In [24]:

```
def piValue(num_trials):
    x_vals = [random() for k in range(num_trials)]
    y_vals = [random() for k in range(num_trials)]
    number = 0
    for x in range(num_trials):
        if (np.sqrt(x_vals[x]**2 + y_vals[x]**2)<= 1):
            number += 1
    value = number * 4 /num_trials
    return value


seed(0)
piValue(10**5)
```

Out[24]:

```
3.13364
```

In [25]:

```
def piValue(num_trials):
    x_vals = [random() for k in range(num_trials)]
    y_vals = [random() for k in range(num_trials)]
    number = 0
    for x in range(num_trials):
        if (np.sqrt(x_vals[x]**2 + y_vals[x]**2)<= 1):
            number += 1
    value = number * 4 /num_trials
    return value


seed(0)
piValue(10**8)
```

Out[25]:

```
3.14178916
```

Appendix: Customizing Your Plots

One thing you have probably noticed is that when you write "bare-bones" code such as we have above, certain defaults are used for the size and layout of the figure and the style of the drawing. One of the most noticable is that when you draw multiple lines, Matplotlib will change the color each time you call the same function (notice that this doesn't happen when calling a different function, e.g., plot followed by scatter).

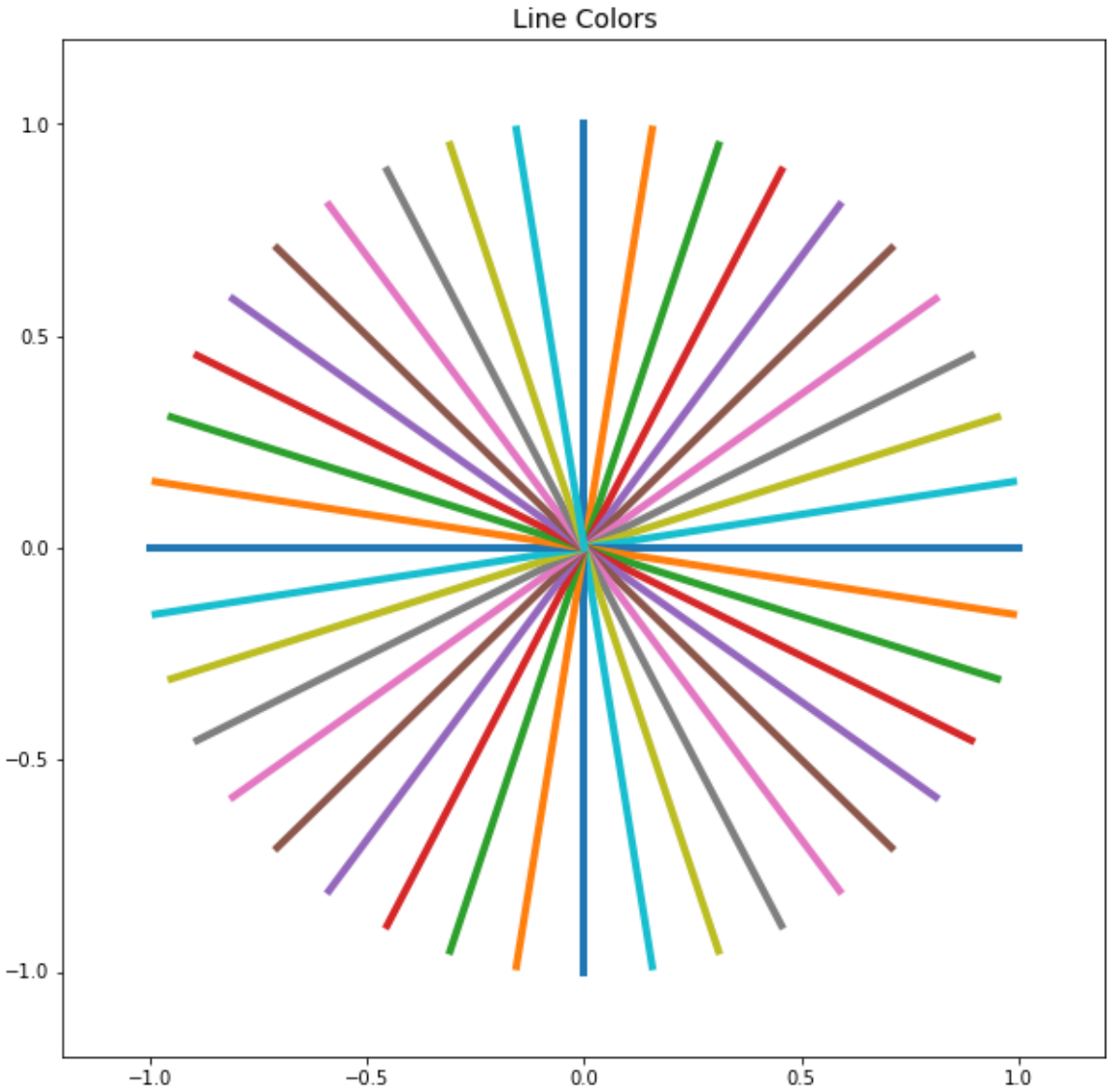
Using Colors

Matplotlib cycles through a sequence of 10 colors, which is fine if that is what you want. For my taste, they are pretty ugly, and in the next section we will show you how to use the colors you want.

In [26]:

```
print("\nThe 10 Matplotlib color sequence, starting at 12 o\'clock and going clo
ckwise:")

plt.figure(figsize=(10,10))
for k in np.arange(0,2*np.pi,np.pi/20):                    # arange is like range,
 except it allows you to use floats
    plt.plot([0,np.sin(k)],[0,np.cos(k)],lw=4)
plt.title('Line Colors',fontsize=14)
plt.xlim([-1.2,1.2])
plt.ylim([-1.2,1.2])
plt.show()
```

```
The 10 Matplotlib color sequence, starting at 12 o'clock and going c
lockwise:
```



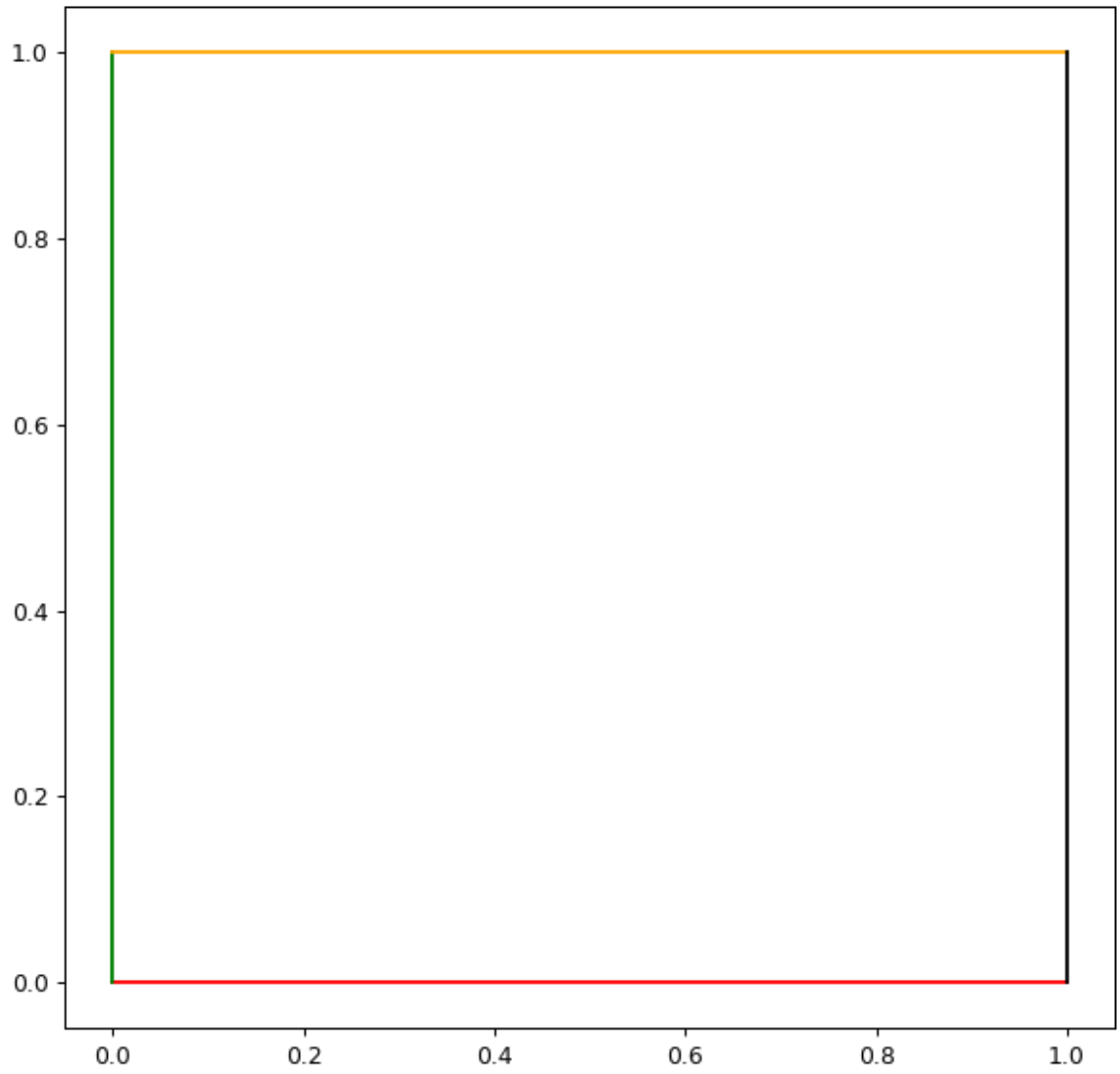
Here is an example where we simply change the colors of the plot using the appropriate parameter; see a complete list of colors here: https://matplotlib.org/2.0.2/api/colors_api.html (https://matplotlib.org/2.0.2/api/colors_api.html)

In [27]:

```
# EXAMPLE: Plotting a square with lines of different colors
plt.figure(num=None, figsize=(8, 8), dpi=89)
plt.plot([0,1],[0,0],color='red') # Line connecting (0,0) to (1,0)
plt.plot([0,0],[0,1],color='green') # Line connecting (0,0) to (0,1)
plt.plot([0,1],[1,1],color='orange') # Line connecting (0,1) to (1,1)
plt.plot([1,1],[0,1],color='black') # Line connecting (1,0) to (1,1)
```

Out[27]:

```
[<matplotlib.lines.Line2D at 0x1a198ac128>]
```



Changing the Style of Plots

Here is an example showing how to

- change the size of the whole figure
- change the color of lines or points
- change the style of lines or points

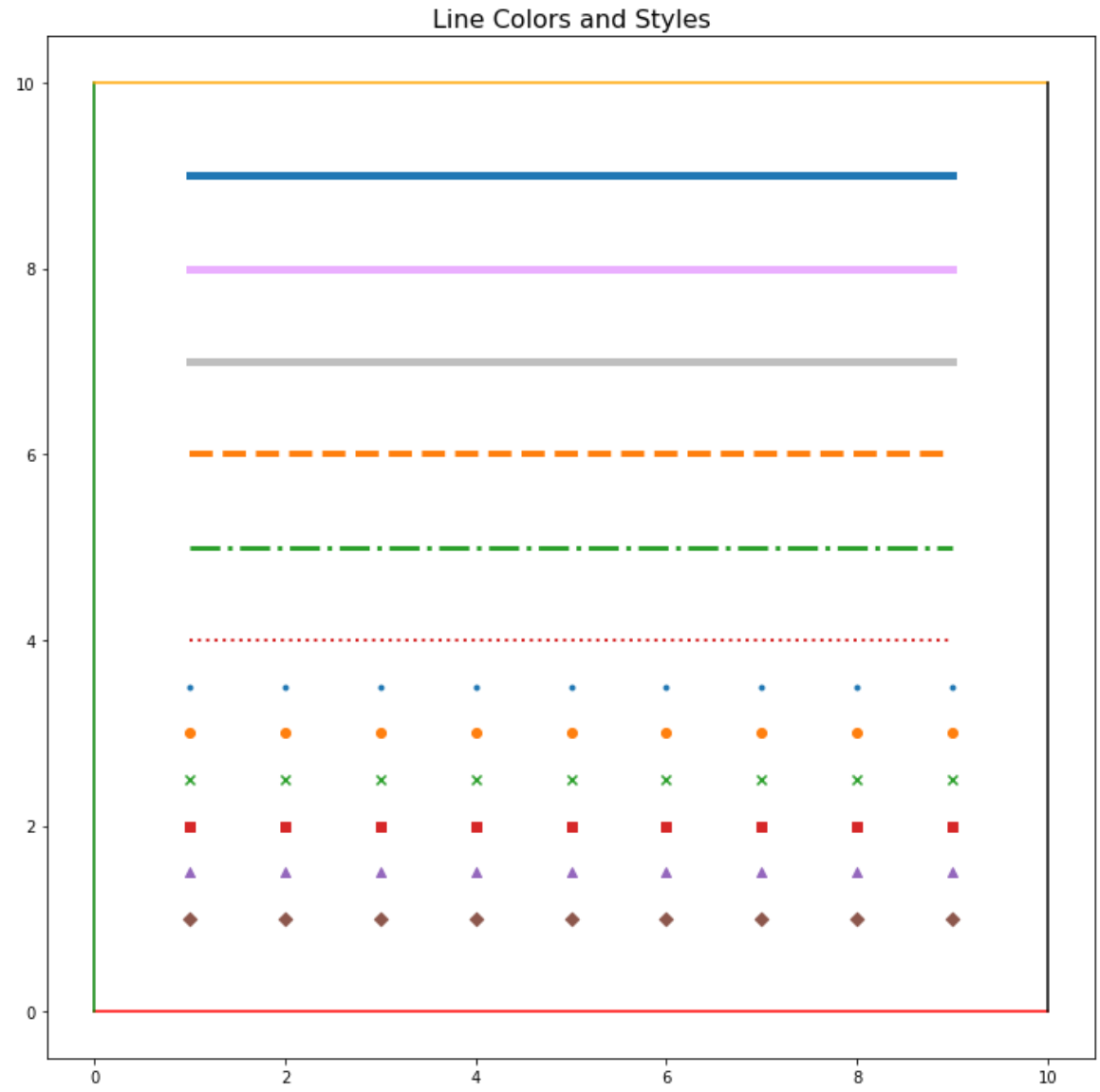
To see a complete list of lines styles see: https://matplotlib.org/2.0.2/api/lines_api.html (https://matplotlib.org/2.0.2/api/lines_api.html)

To see a complete list of colors see: https://matplotlib.org/2.0.2/api/colors_api.html (https://matplotlib.org/2.0.2/api/colors_api.html)

To see a complete list of marker (point) styles see: https://matplotlib.org/2.0.2/api/markers_api.html#module-matplotlib.markers (https://matplotlib.org/2.0.2/api/markers_api.html#module-matplotlib.markers)

In [28]:

```
# EXAMPLE: Plotting a square via lines
plt.figure(figsize=(12, 12))             # the size is (horizontal, vertical)
plt.title("Line Colors and Styles",fontsize=16)
plt.plot([0,10],[0,0],  color='red') # Line connecting (0,0) to (1,0)
plt.plot([0,0],[0,10],  color='green') # Line connecting (0,0) to (0,1)
plt.plot([0,10],[10,10],color='orange') # Line connecting (0,1) to (1,1)
plt.plot([10,10],[0,10],color='black') # Line connecting (1,0) to (1,1)
plt.plot([1,9],[9,9], linewidth=5)    # give a linewidth in points, default is
 1.0
plt.plot([1,9],[8,8], linewidth=5,color = '#eaafff')    # for custom color give
 the RGB value in hex
plt.plot([1,9],[7,7], linewidth=5,color='0.75') # for grey give the percentage o
f white in quotes
plt.plot([1,9],[6,6], lw=4,linestyle='--') # Linestyles
plt.plot([1,9],[5,5], lw=3,linestyle='-.') # Linestyles
plt.plot([1,9],[4,4], lw=2,linestyle=':') # Linestyles

plt.scatter(range(1,10),[3.5]*9,marker='.')  # various markers, if you don't spe
cify the colors it will cycle
plt.scatter(range(1,10),[3]*9,marker='o')    # through a bunch of colors, starti
ng with blue, orange, green, etc.
plt.scatter(range(1,10),[2.5]*9,marker='x')
plt.scatter(range(1,10),[2]*9,marker='s')
plt.scatter(range(1,10),[1.5]*9,marker='^')
plt.scatter(range(1,10),[1]*9,marker='D')
print()
```



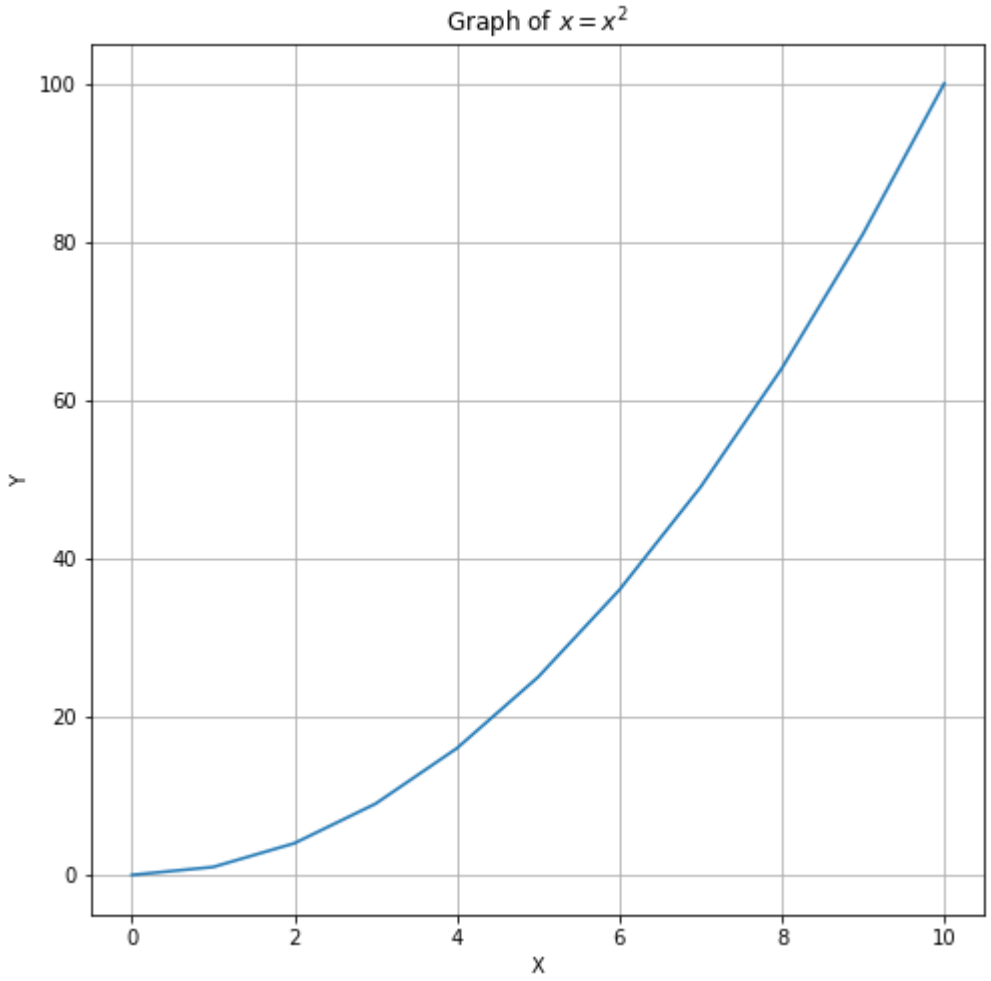
Et Cetera

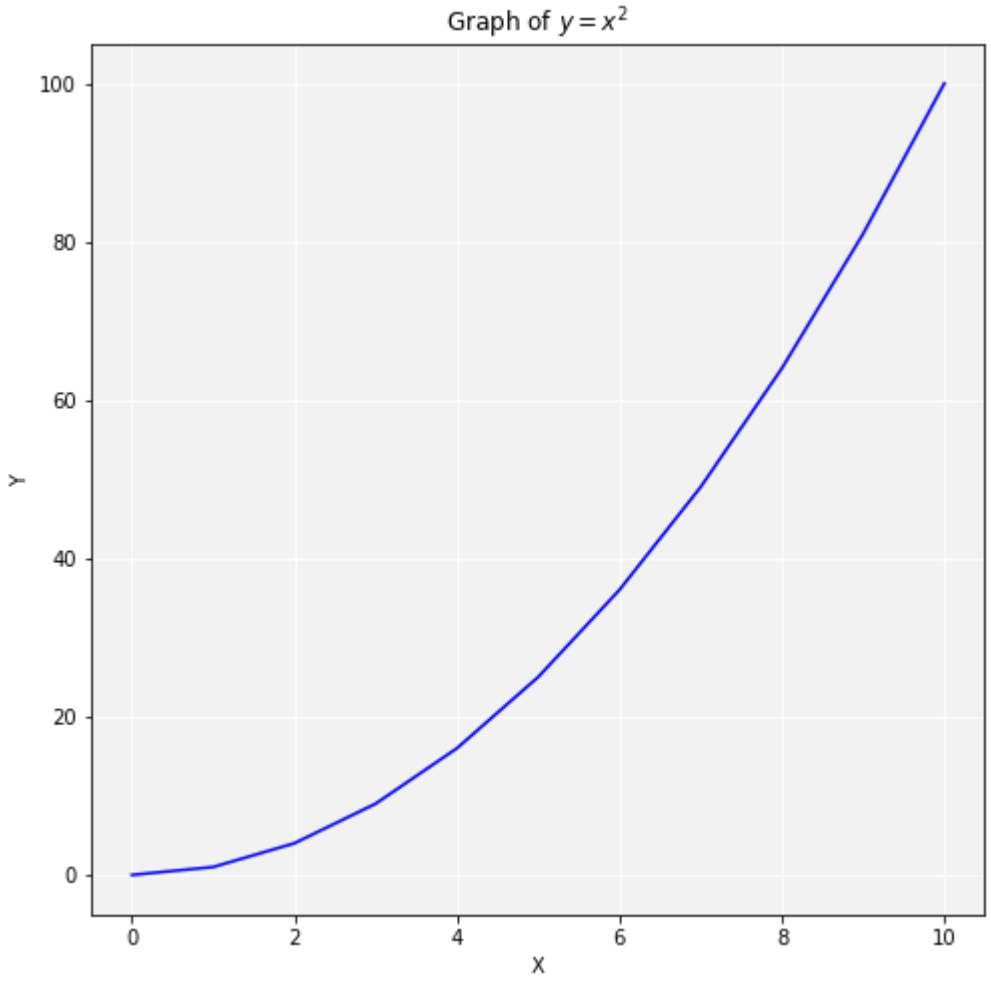
Then you can start getting obsessive, adding gridlines, changing the background color, adding legends, text, and so on.

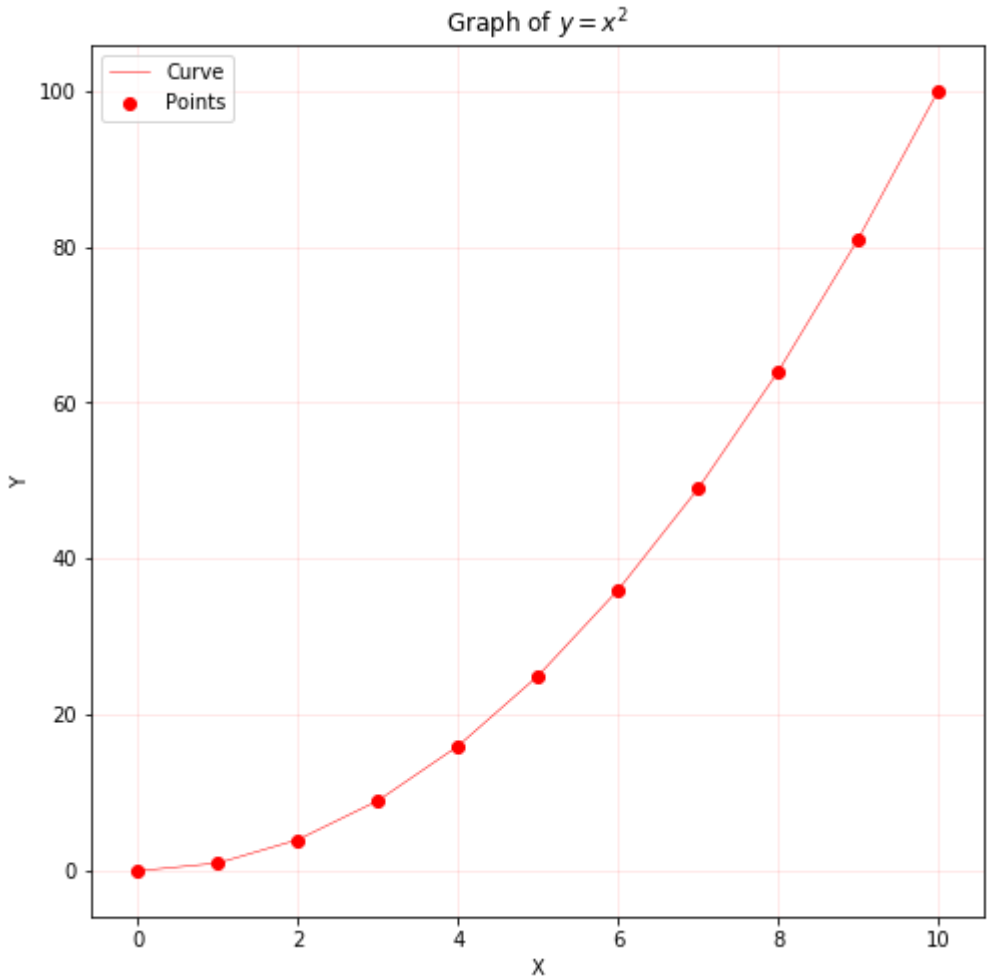
Another nice feature of matplotlib is that you can insert simple Latex commands into titles and text.....

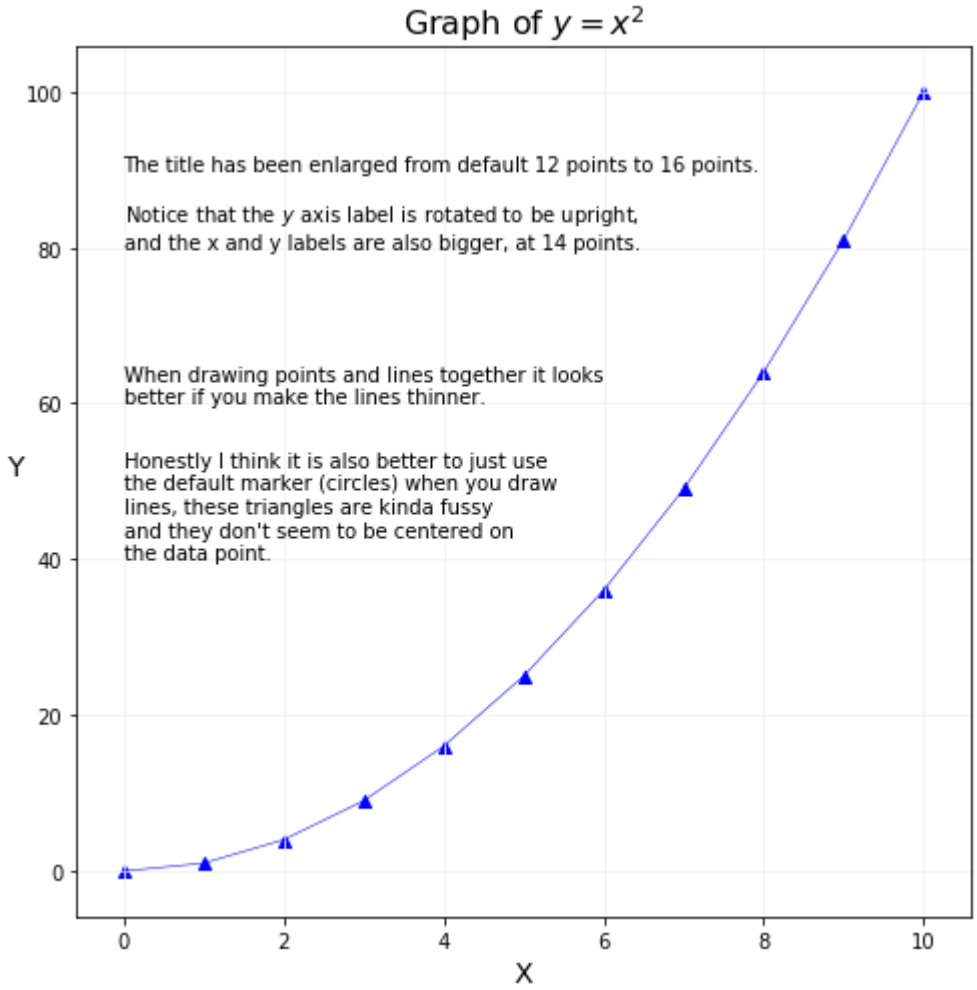
In [29]:

```
# Plotting a smooth curve for the function x^2
x = [i for i in range(11)]
y = [i**2 for i in x]


plt.figure(figsize=(8, 8))
plt.title('Graph of $x = x^2$')
plt.xlabel("X")
plt.ylabel("Y")
plt.grid()
plt.plot(x,y)
plt.show()


plt.figure(figsize=(8, 8))
plt.title('Graph of $y = x^2$')
plt.grid(color='w')                # grid of white lines -- don't use points wit
h this, they look funny
plt.gca().set_facecolor('0.95')    # background of light grey
plt.plot(x,y,color='b')
plt.xlabel("X")
plt.ylabel("Y")
plt.show()


plt.figure(figsize=(8, 8))
plt.title('Graph of $y = x^2$')
plt.grid(color='r',alpha=0.1)       # alpha sets the transparency, 0 = invisible
and 1 = normal
plt.plot(x,y,color='r',lw=0.5,label='Curve')
plt.scatter(x,y,color='r',marker='o',label='Points')
plt.legend()
plt.xlabel("X")
plt.ylabel("Y")
plt.show()


plt.figure(figsize=(8, 8))
plt.title('Graph of $y = x^2$',fontsize=16)
plt.xlabel("X",fontsize=14)
plt.ylabel("Y",rotation=0,fontsize=14)
plt.grid(color='0.95')
plt.text(0,90,"The title has been enlarged from default 12 points to 16 points."
)
plt.text(0,80,"Notice that the $y$ axis label is rotated to be upright, \nand th
e x and y labels are also bigger, at 14 points.")   # lower left corner of text
 string is at point (0,60)
plt.text(0,60,"When drawing points and lines together it looks \nbetter if you m
ake the lines thinner.")
plt.text(0,40,"Honestly I think it is also better to just use\nthe default marke
r (circles) when you draw \nlines, these triangles are kinda fussy\nand they do
n't seem to be centered on \nthe data point.")
plt.plot(x,y,color='b',lw=0.5)
plt.scatter(x,y,color='b',marker='^')
plt.show()
```


Graph of $x = x^2$
100
80
60
40
20
0
Y
0
2
4
6
8
10
X


Graph of $y = x^2$
100
80
60
40
20
0
Y
0
2
4
6
8
10
X


Graph of $y = x^2$
Curve
Points
100
80
60
40
20
0
Y
0
2
4
6
8
10
X


Graph of $y = x^2$
The title has been enlarged from default 12 points to 16 points.
Notice that the y axis label is rotated to be upright,
and the x and y labels are also bigger, at 14 points.
When drawing points and lines together it looks
better if you make the lines thinner.
Honestly I think it is also better to just use
the default marker (circles) when you draw
lines, these triangles are kinda fussy
and they don't seem to be centered on
the data point.
Y
X
0
20
40
60
80
100
0
2
4
6
8
10