

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) (<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.

```
In [36]: # Here are some imports which will be used in the code in the rest of the 1

# Imports used for the code in CS 237

import numpy as np          # arrays and functions which operate on a
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 [37]: # To plot the points (1,2), (2,3), (3,6), (4,8) we would list the x values
X = [1,2,3,4]
Y = [2,3,6,8]

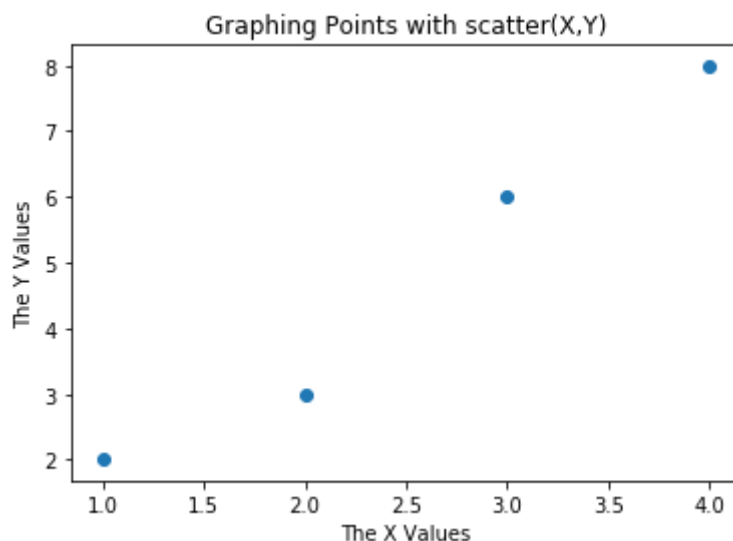
print("\nThis is the list of points:",list(zip(X,Y)))
print("They must be input to the function as separate lists:")
print("\tX =",X)
print("\tY =",Y,"\n")
plt.scatter(X,Y)
plt.title('Graphing Points with scatter(X,Y)')
plt.xlabel("The X Values")
plt.ylabel("The Y Values")
plt.show()
```

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]

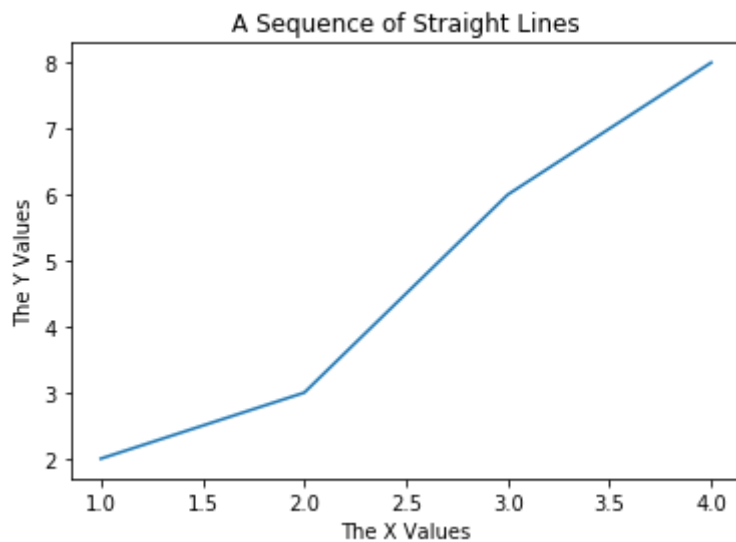


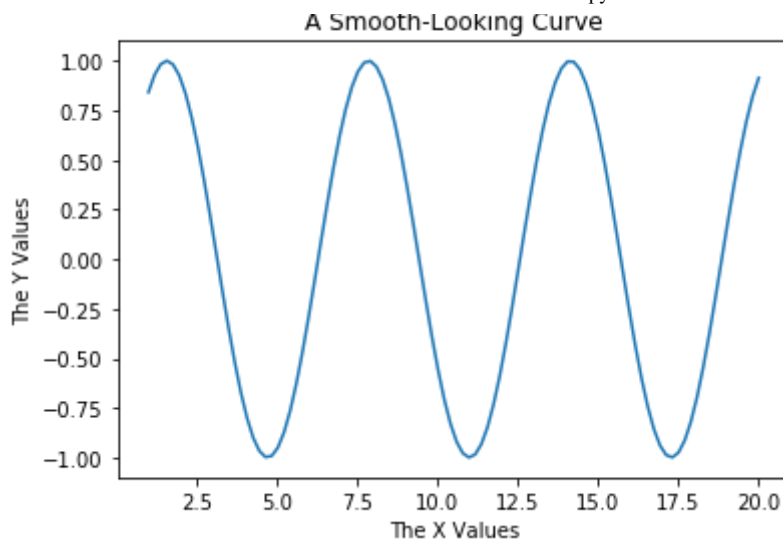
Plotting Lines and Curves

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 [38]: # 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()

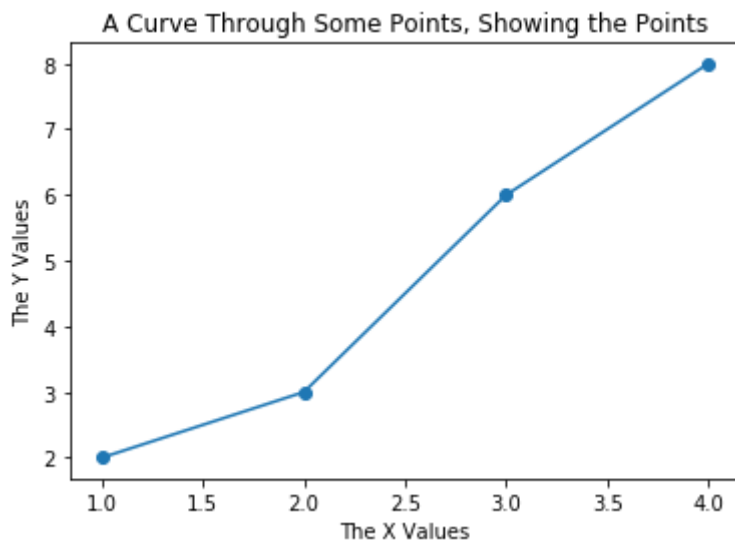
X = np.linspace(1,20,100) # returns a list of 100 equally-spaced values
Y = [np.sin(x) for x in X]
plt.plot(X,Y)
plt.title('A Smooth-Looking Curve')
plt.xlabel("The X Values")
plt.ylabel("The Y Values")
plt.show()
```





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

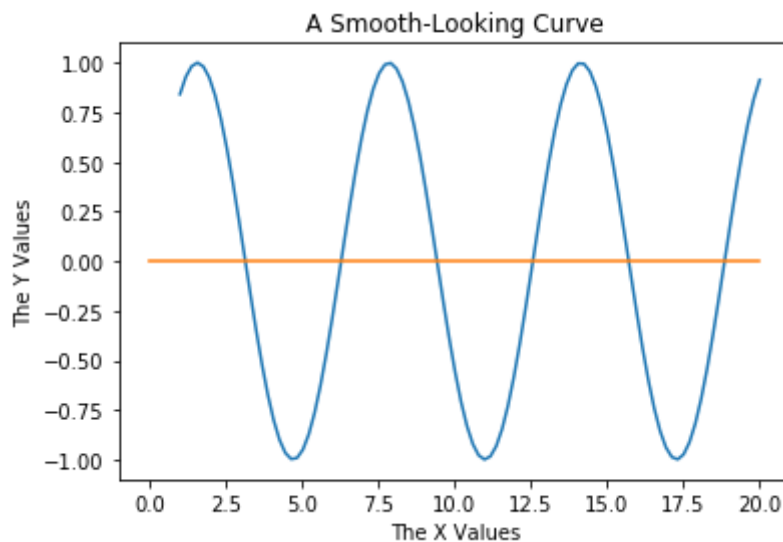
```
In [39]: plt.scatter([1,2,3,4], [2,3,6,8])
plt.plot([1,2,3,4], [2,3,6,8])
plt.title('A Curve Through Some Points, Showing the Points')
plt.xlabel("The X Values")
plt.ylabel("The Y Values")
plt.show()
```



If you want to draw a single line from (x_1, y_1) to (x_2, y_2) you can plot $[x_1, x_2]$ and $[y_1, y_2]$.

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

```
In [40]: X = np.linspace(1,20,100)           # returns a list of 100 equally-spaced values
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 \(https://docs.scipy.org/doc/numpy/user/quickstart.html\)](https://docs.scipy.org/doc/numpy/user/quickstart.html).

```
In [41]: # Run this cell several times and see what happens

random()
```

Out[41]: 0.5488135039273248

```
In [42]: # Run this cell several times 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[42]: [0.7151893663724195,
0.6027633760716439,
0.5448831829968969,
0.4236547993389047,
0.6458941130666561,
0.4375872112626925,
0.8917730007820798,
0.9636627605010293,
0.3834415188257777,
0.7917250380826646]
```

```
In [43]: # Run this cell several times and see what happens
# This produces a Numpy 1D vector, which is
# another way to store a sequence.

size = 5
random(size)
```

```
Out[43]: array([0.52889492, 0.56804456, 0.92559664, 0.07103606, 0.0871293 ])
```

```
In [44]: # In most cases, arrays are interchangeable with lists, but if you need a list, just do this:

list(random(size))
```

```
Out[44]: [0.02021839744032572,
0.832619845547938,
0.7781567509498505,
0.8700121482468192,
0.978618342232764]
```

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

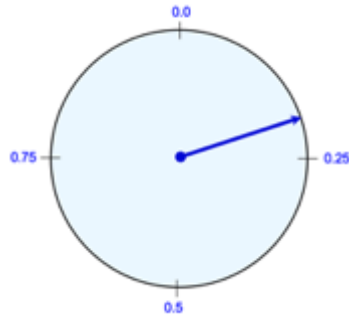
seed(0)
random(5)
```

```
Out[45]: 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 choose a real number in the half-open interval $[0..1)$:



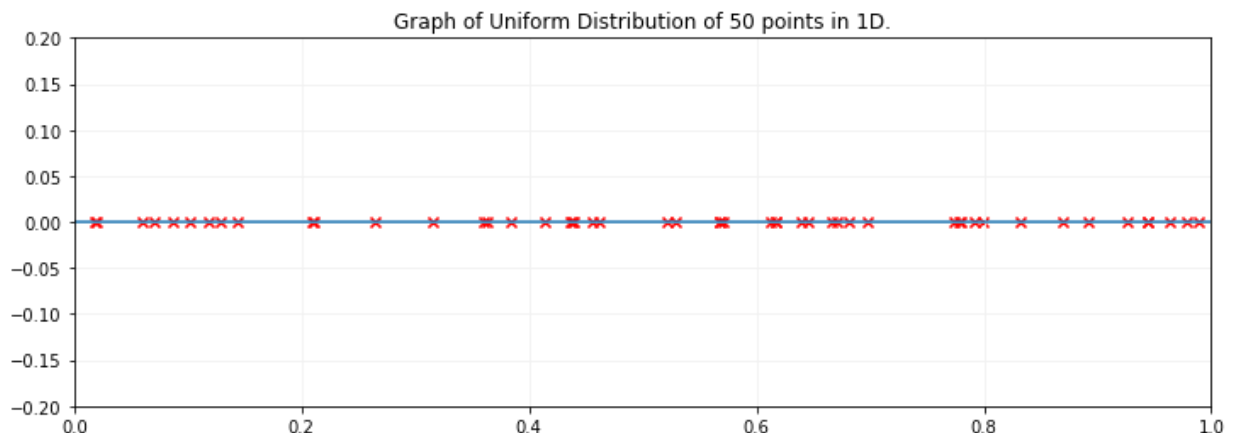
We will simulate this using the Python function `random()` from the `Numpy.random` library we imported above.


```
In [46]: # Use numpy's random() function to build a list of x_vals with the foll
#         - x_vals list should have length num_trials
#         - Each value in x_vals should be a random float in the range [0.
#         Now plot the points (xi,0) along the x axis in the diagram below

def random_line_plot(num_trials):
    x_vals = random(num_trials) # <--- Your code h
    y_vals = [ 0 for k in range(num_trials)] # <--- Your co
    #y_vals=[0]*num_trials
    plt.figure(figsize=(12, 4))
    plt.title('Graph of Uniform Distribution of '+str(num_trials)+' points
    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()

# TO DO: Complete the lines of code indicated above. Then comment out the
# statement and run this cell several times to see the distribution of poin
# put the seed command back in before you submit!

#seed(0)
random_line_plot(50)
```



Part B

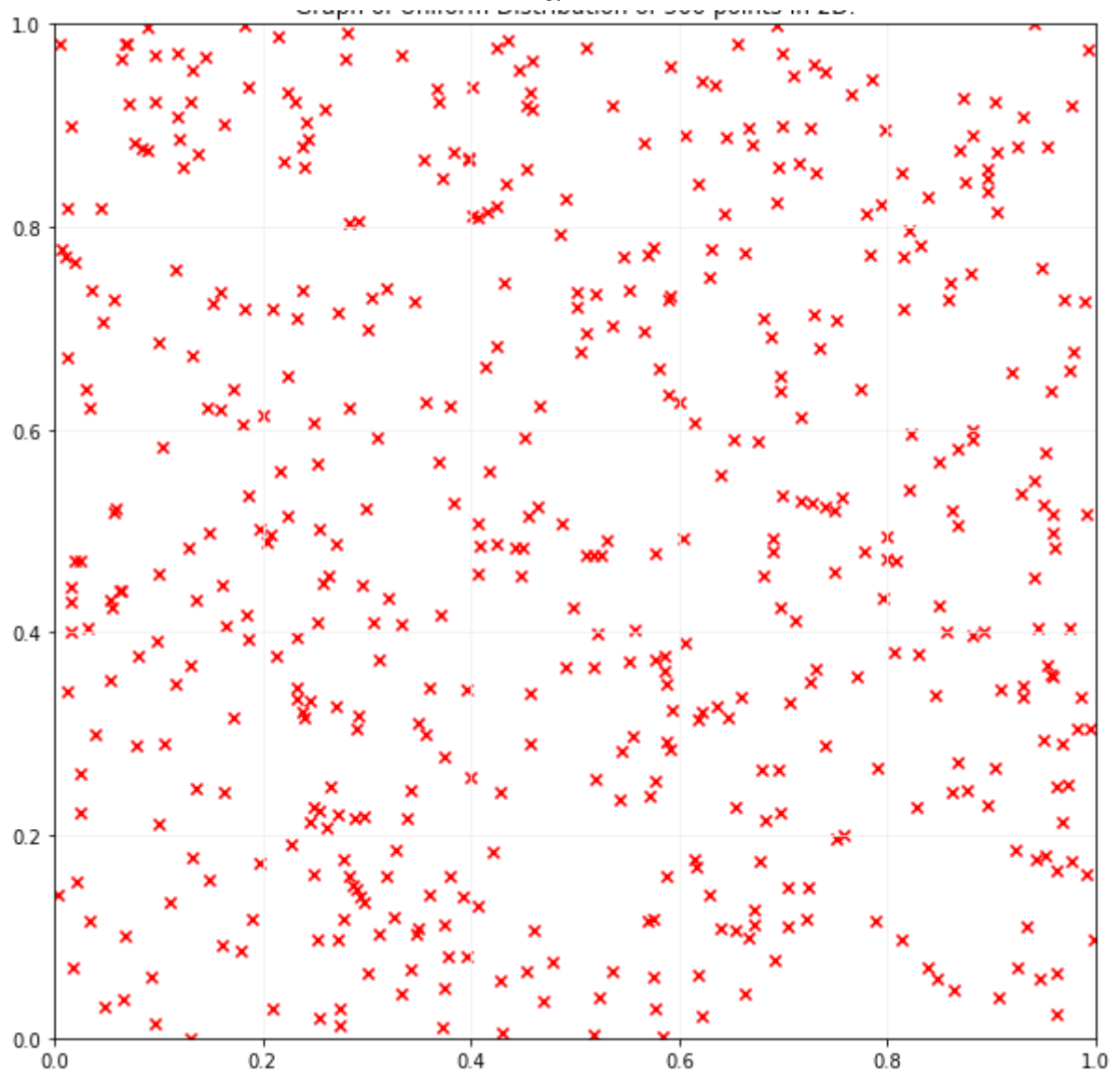
Now we will simulate the experiment of throwing a dart at a unit square, which will produce a scatter plot of random points in a 2D grid.

```
In [47]: # TO DO: Complete the following function stub which will produce a scatter
#         random points in a unit square.

def random_plane_plot(num_trials):
    """Uses numpy's random function to build a list of x and y values with
    - 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(num_trials)          # <--- Your code here
    y_vals = random(num_trials)          # <--- Your code here
    plt.figure(num=None, figsize=(10, 10))
    plt.title('Graph of Uniform Distribution of '+str(num_trials)+' points')
    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()

# Comment out the next line and run this cell several times to see the dist
# put the seed command back in before you submit!

#seed(0)
random_plane_plot(500)
```

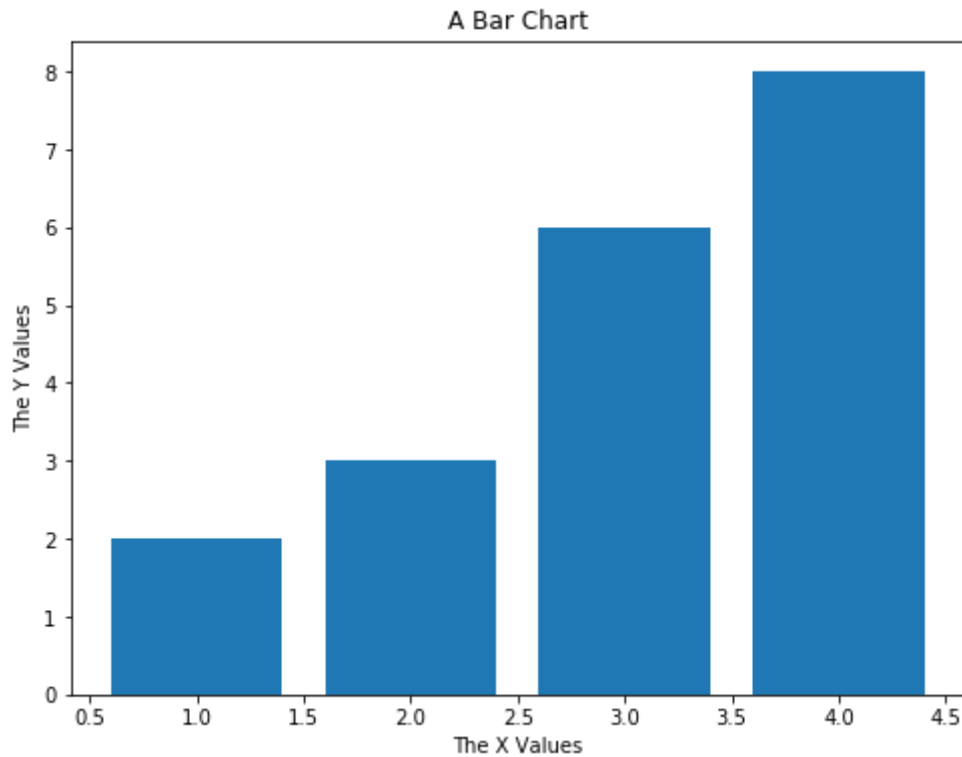


[End of Problem One]

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:

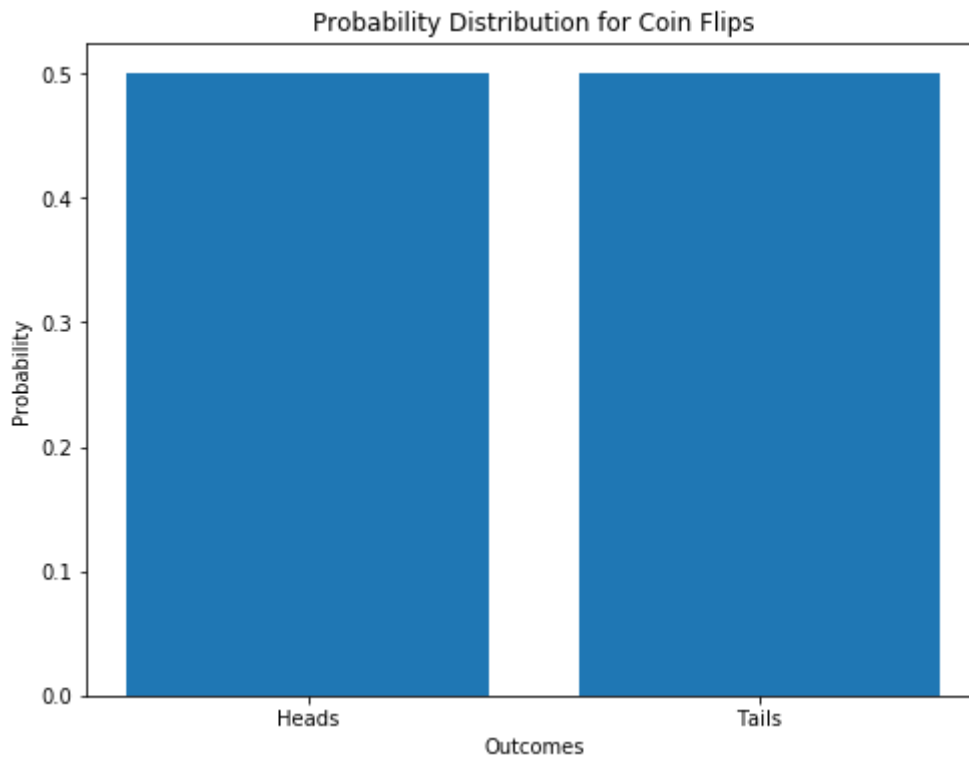
```
In [48]: # To plot the points (1,2), (2,3), (3,6), (4,8) we would list the x values
plt.figure(num=None, figsize=(8, 6))
plt.bar([1,2,3,4], [2,3,6,8])
plt.title('A Bar Chart')
plt.xlabel("The X Values")
plt.ylabel("The Y Values")
plt.show()
```



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 [49]: # 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()
```

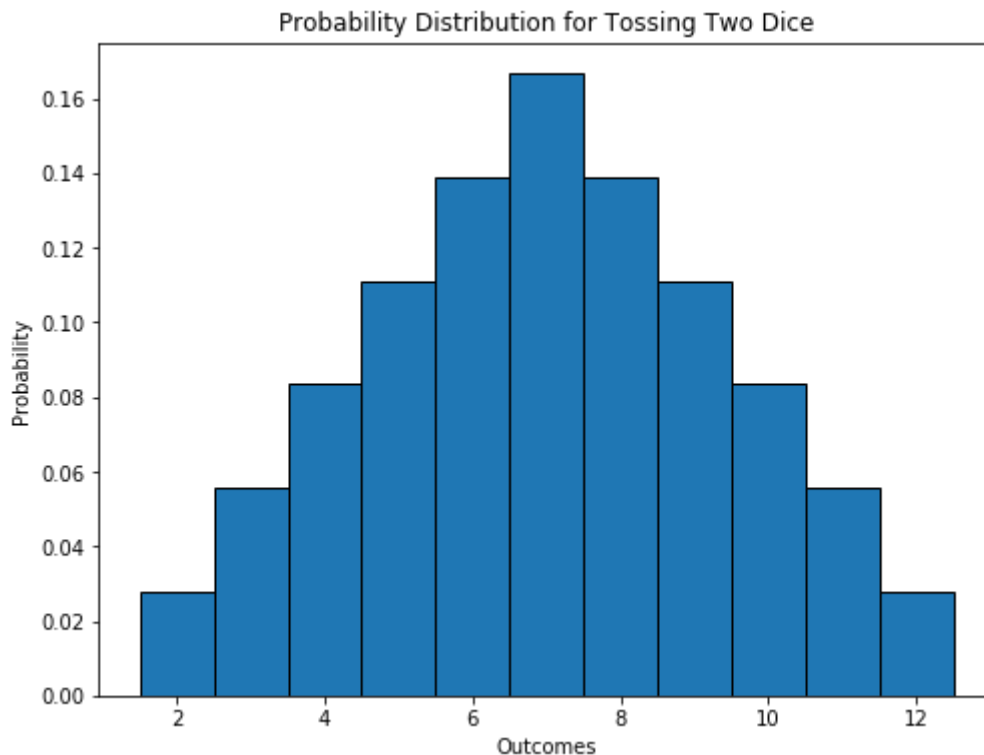


With a few tweaks, you can create an attractive bar chart for arbitrary probability distributions (we will cover the topic of distributions in a few weeks).

```
In [50]: # 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]

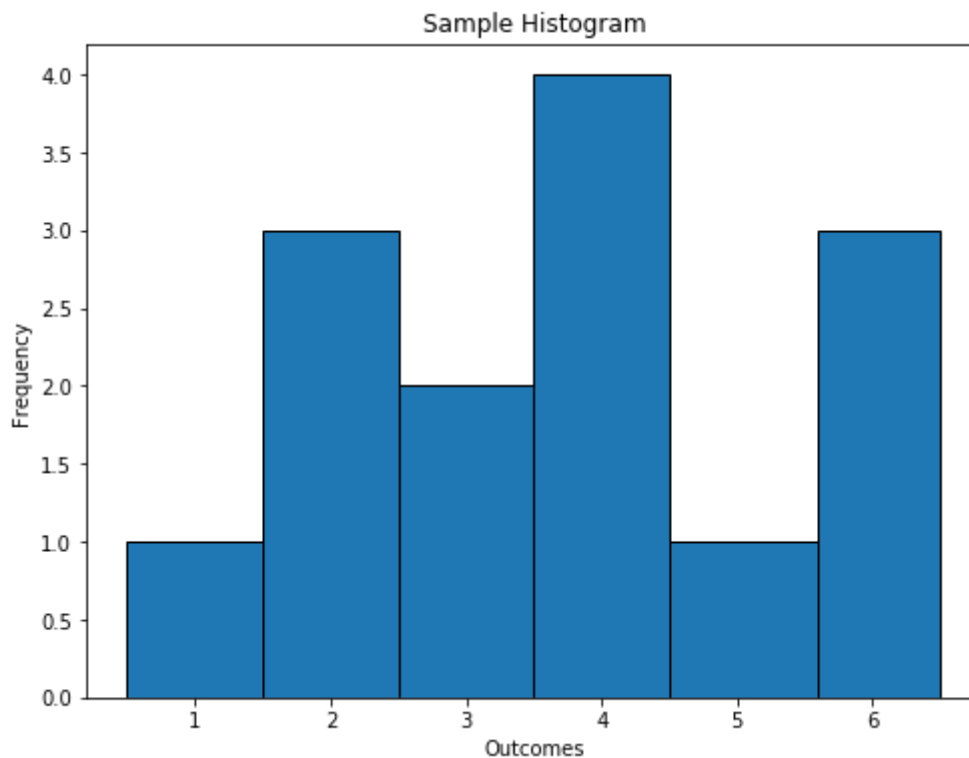
plt.figure(num=None, figsize=(8, 6))
plt.bar(x,y, width=1.0,edgecolor='black') # <--- Note how we set the width
plt.title('Probability Distribution for Tossing Two Dice')
plt.ylabel("Probability")
plt.xlabel("Outcomes")
plt.show()
```



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 [51]: 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],e
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 $k \in \{1, 2, 3, 4, 5, 6\}$ to be

$$\frac{\text{number of times we observed a value } k}{\text{num-trials}} \approx \frac{1}{6}$$

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 [52]: # 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 in
# the Python range function, and so on.

randint(0,4)
```

Out[52]: 2

```
In [53]: # You can also ask it for an array, since it is a Numpy function:

randint(0,4,10)
```

Out[53]: array([0, 3, 0, 3, 2, 1, 3, 1, 3, 1])

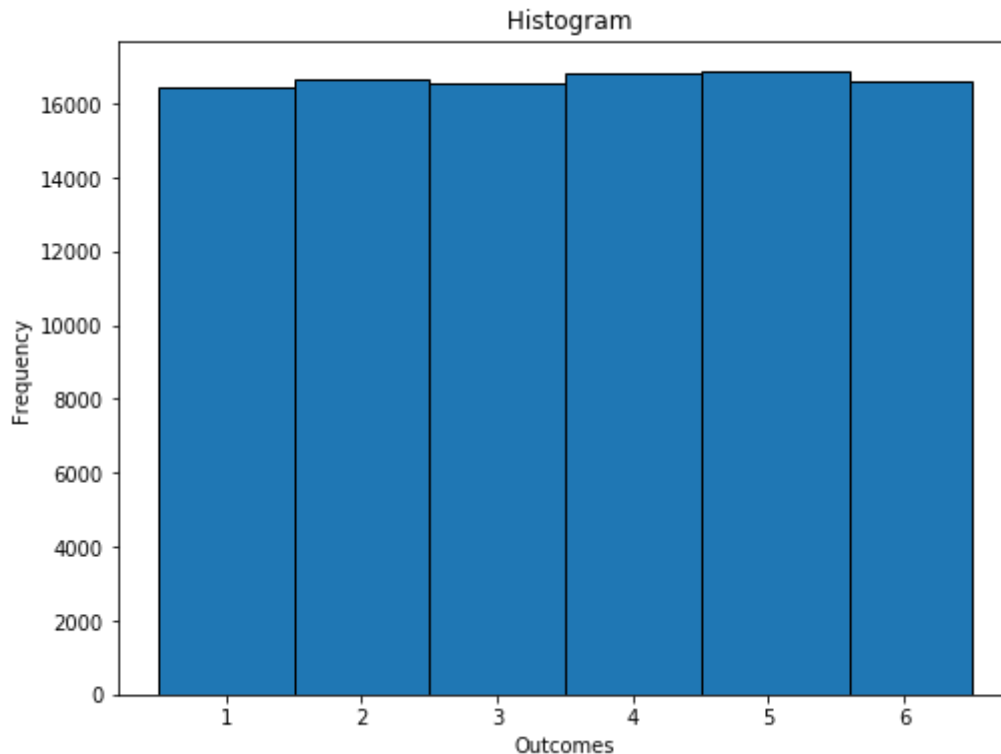
```
In [54]: seed(0)

def roll_die(num_trials = 100000):          # <-- note the use of the default
    trials_and_results = randint(1,7,num_trials) # this creates a 1D array
    return trials_and_results

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



```
In [55]: # Your code here
plt.figure(num=None, figsize=(8, 6))
plt.hist(example_trials, bins=[0.5, 1.5, 2.5, 3.5, 4.5, 5.5, 6.5], edgecolor='black')
plt.title('Histogram')
plt.xlabel("Outcomes")
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 $k \in \{1, 2, 3, 4, 5, 6\}$ to be

$$\frac{\text{number of times we observed a value } k}{\text{num-trials}} \approx \frac{1}{6}$$

- 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 [56]: *# Solution*

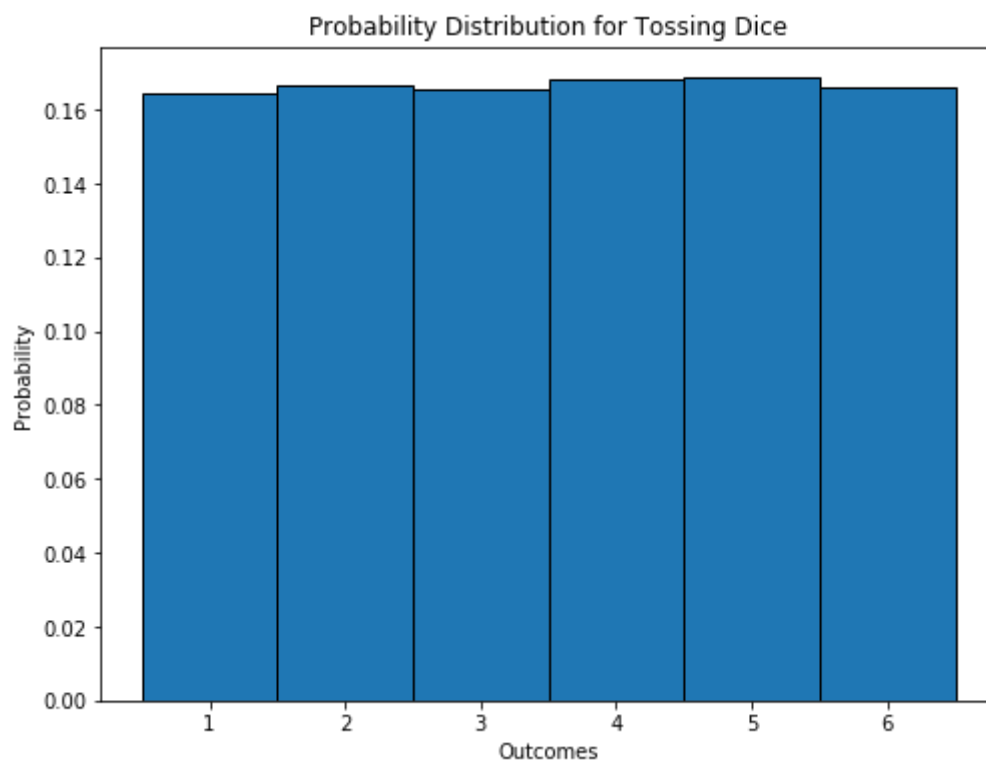
```
seed(0)

def show_distribution(outcomes,title='Probability Distribution'):
    totalfreq = Counter(outcomes)
    #print (totalfreq)
    zuixiao=outcomes.min()
    zuida=outcomes.max()
    zuida1=zuida+1
    zuixiao1=zuixiao-1
    jishu=zuida-zuixiao+1
    eachdis = [totalfreq [x] for x in range (zuixiao,zuida1)]
    #print(eachdis)

    plt.figure(num=None, figsize=(8, 6))
    x = [k for k in range(zuixiao,zuida1)]
    y = [eachdis [x]/100000 for x in range (0,jishu)]
    #print(y)

    plt.bar(x,y, width=1.0,edgecolor='black')    # <--- Note how we set the
    plt.title('Probability Distribution for Tossing Dice')
    plt.ylabel("Probability")
    plt.xlabel("Outcomes")
    plt.show()

show_distribution(example_trials)
```



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 n dies and recording their sum. For example, if $n = 2$ 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.

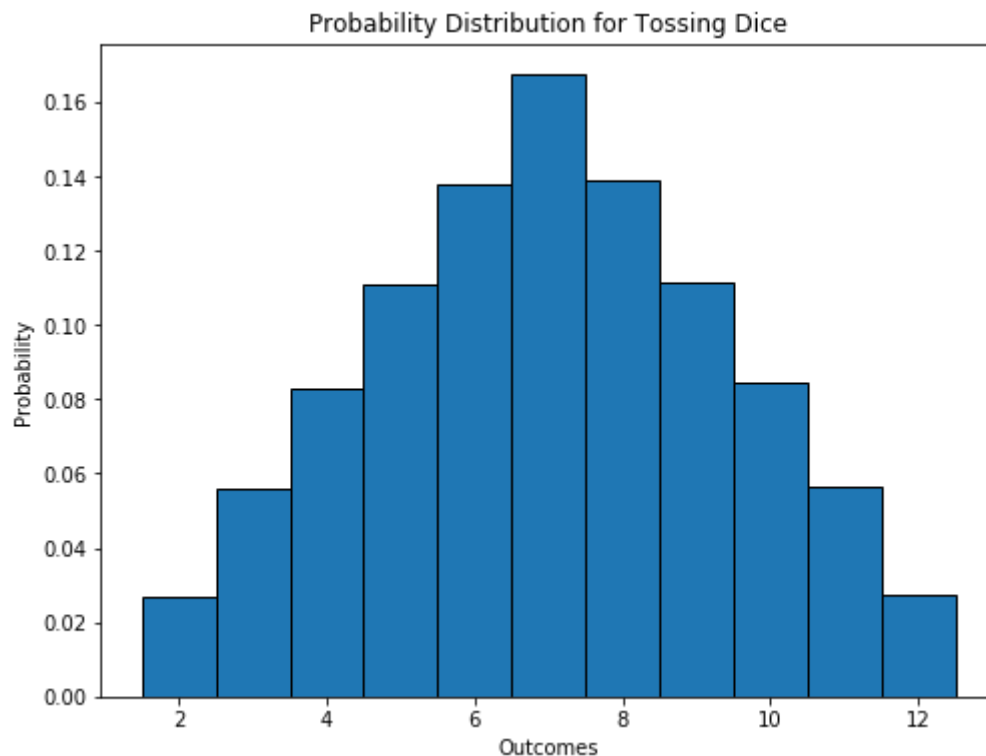
Hint: Not required, but think about how you might do this in one line using Numpy and list comprehensions.

```
In [57]: # Solution
```

```
seed(0)
```

```
def roll_and_add_dice(num_dice, num_trials = 10**5):  
    trials_and_results1 = randint(1,7,num_trials)  
    trials_and_results2 = randint(1,7,num_trials)  
    finalresults=trials_and_results1+trials_and_results2  
    return finalresults
```

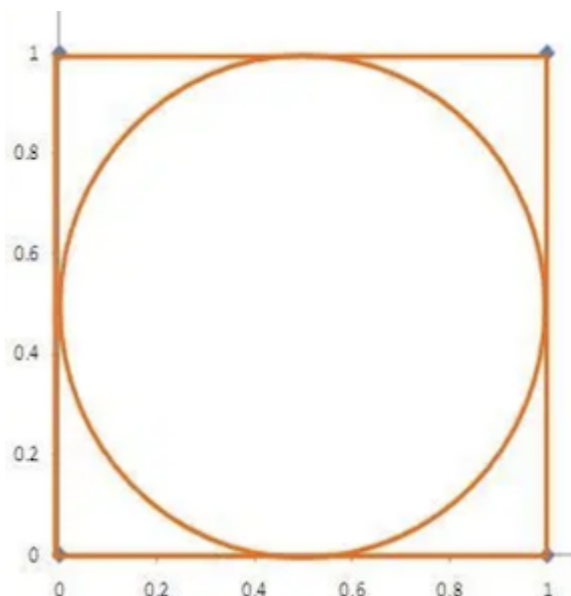
```
show_distribution(roll_and_add_dice(2),title='Probability Distribution for
```



Problem 5 (Monte Carlo Calculation of π)

This final problem is also a Monte Carlo simulation, but this time in the continuous domain: we will calculate the value of π 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 $\pi * (0.5^2) = \frac{\pi}{4}$.

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 π .

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 π .
- (B) Without drawing the diagram, calculate the value of π you would get from 10^5 trials.
- (C) After completing (B), try to get a more accurate value for π 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.

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

In [58]:

```
# (A)
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

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

%matplotlib inline
import matplotlib.pyplot as plt

def plot_a_circle(num_trials):
    x_vals = random(num_trials)
    y_vals = random(num_trials)

    plt.figure(num=None, figsize=(5, 5))
    plt.grid(color='0.95')
    plt.ylim(0,1)
    plt.xlim(0,1)
    plt.plot(x,y)
    plt.scatter(x_vals, y_vals,marker="x",color="r")

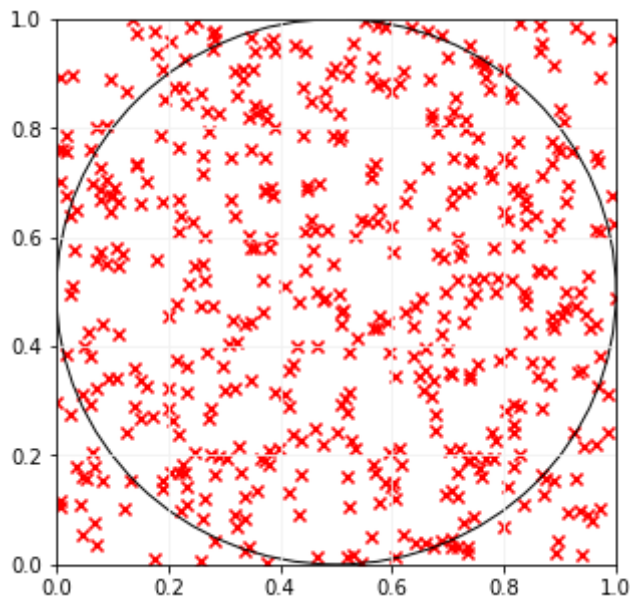
    mycircle=plt.Circle((0.5,0.5),0.5,color='k',fill=False)
    plt.gcf().gca().add_artist(mycircle)
    plt.show()

    ptsinside=0
    for i in range(0,num_trials):
        length=(x_vals[i]-0.5)**2+(y_vals[i]-0.5)**2
        if length<=0.25:
            ptsinside+=1

    mypi=4*ptsinside/500
    print("My pi is: ",mypi)

result=plot_a_circle(500)

seed(0)
```



My pi is: 3.296

```
In [59]: #(B)
def without_plot(num_trials):
    x_vals = random(num_trials)
    y_vals = random(num_trials)

    ptsinside=0
    for i in range(0,num_trials):
        length=(x_vals[i]-0.5)**2+(y_vals[i]-0.5)**2
        if length<=0.25:
            ptsinside+=1

    mypi=4*ptsinside/num_trials
    print("My pi is: ",mypi)

without_plot(100000)
```

My pi is: 3.13088

```
In [65]: # (C)
without_plot(100000000)
```

My pi is: 3.14144108

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 noticeable 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).

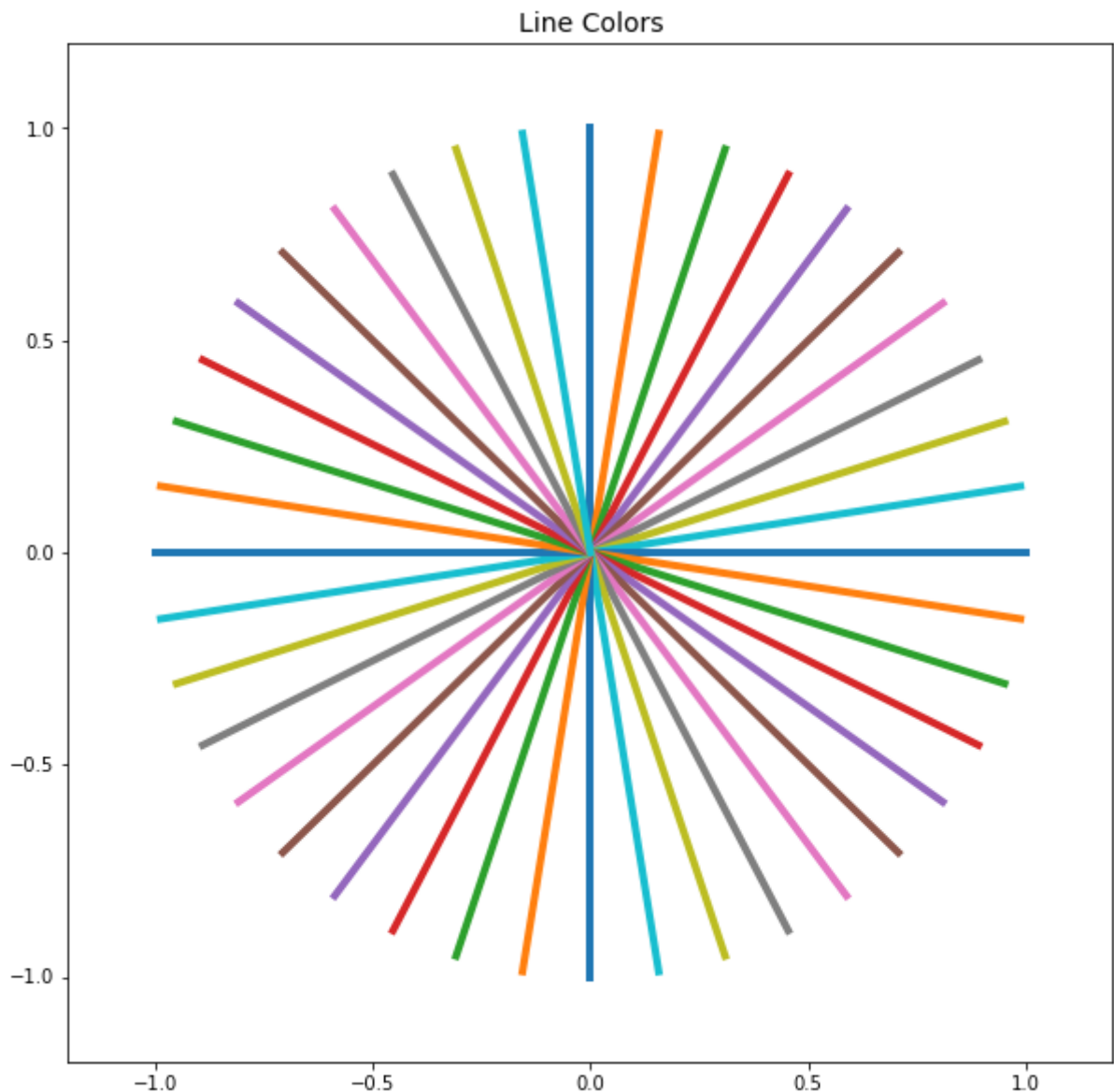
Using Colors

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 [61]: print("\nThe 10 Matplotlib color sequence, starting at 12 o'clock and going clockwise\n")

plt.figure(figsize=(10,10))
for k in np.arange(0,2*np.pi,np.pi/20):
    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()
```

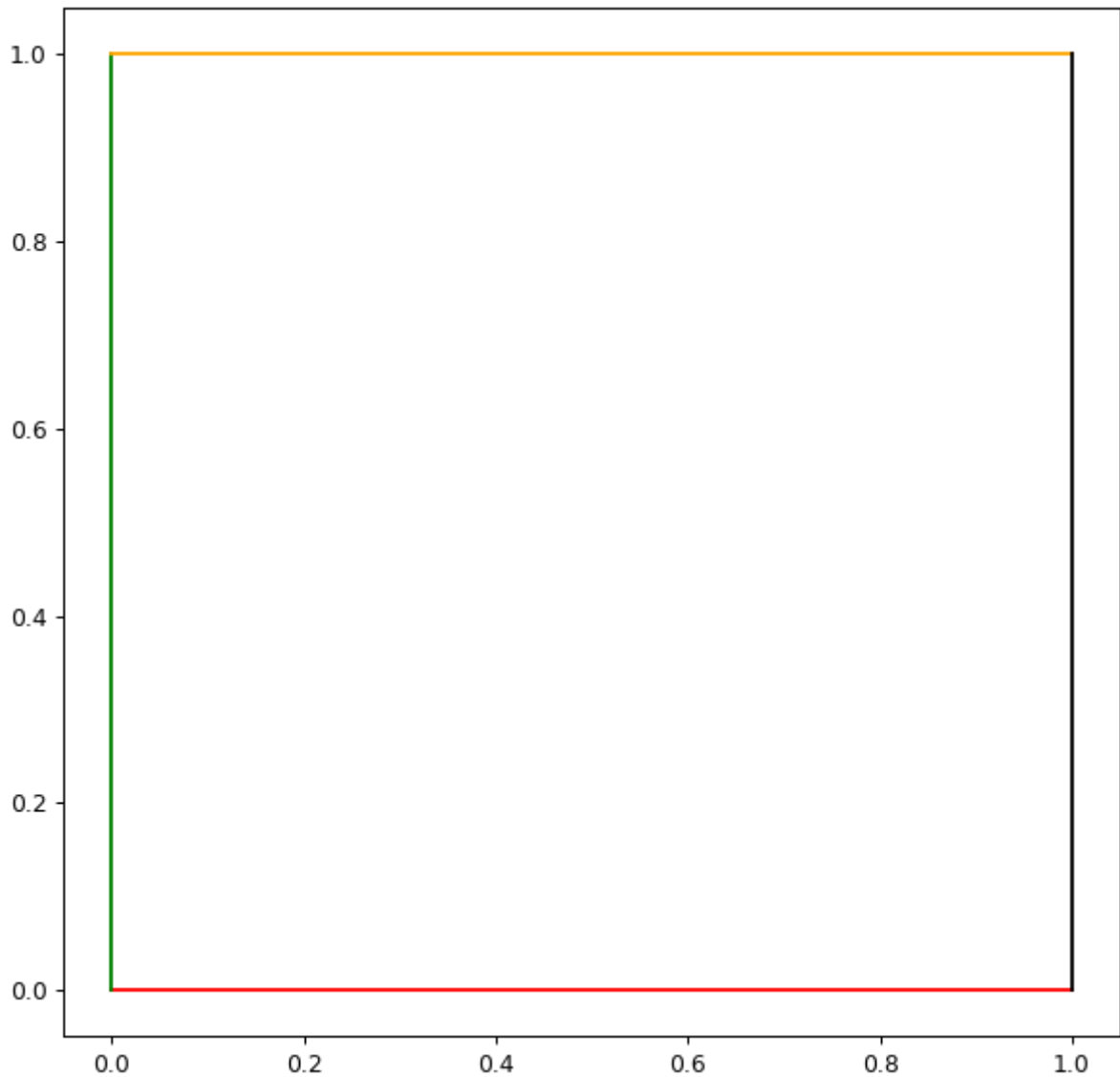
The 10 Matplotlib color sequence, starting at 12 o'clock and going clockwise is:



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 [62]: # 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[62]: [<matplotlib.lines.Line2D at 0x7fba124205f8>]
```



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

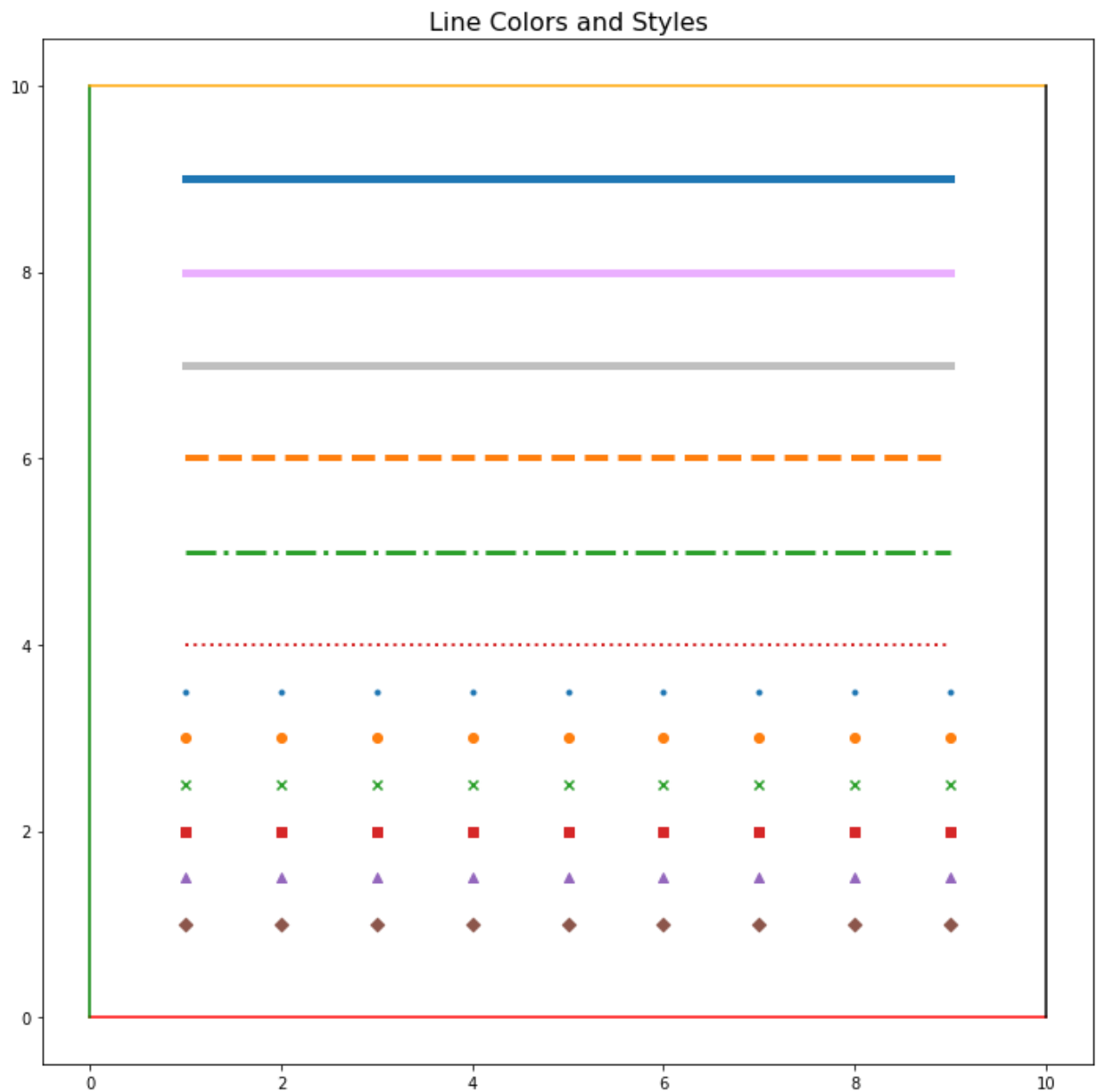
To see a complete list of lines styles see: https://matplotlib.org/2.0.2/api/lines_api.html
(https://matplotlib.org/2.0.2/api/lines_api.html)

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 [63]: # 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
plt.plot([1,9],[8,8], linewidth=5,color = '#eaaaff') # for custom color
plt.plot([1,9],[7,7], linewidth=5,color='0.75') # for grey give the percent
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
plt.scatter(range(1,10),[3]*9,marker='o') # through a bunch of colors, s
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 [64]: # 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  $y = x^2$ ')
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 point
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 = invi
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 poi
plt.text(0,80,"Notice that the  $y$  axis label is rotated to be upright, \na
plt.text(0,60,"When drawing points and lines together it looks \nbetter if
plt.text(0,40,"Honestly I think it is also better to just use\nthe default
plt.plot(x,y,color='b',lw=0.5)
plt.scatter(x,y,color='b',marker='^')
plt.show()

```

