Basic Jupyter Notebook Tutorial

This lab serves as a quick overview of Jupyter Notebooks, numpy, matplotlib, and simulations in Python, which we will use throughout the course. If you are unfamiliar with these, you should carefully read through all of the examples. You are only required to answer the two questions at the bottom for credit.

General Jupyter Notebook Usage Instructions (Overview)

- Click the `Play` button to run and advance a cell. The short-cut for it is `Shift-Enter` .
- To add a new cell, click the `Plus` button.
- You can change the cell mode from code to text in the pulldown menu. Use `Markdown` for writing text.
- You can change the text in `Markdown` cells by double-clicking it. The short-cut for this is `enter` .
- To save your notebook, hit `Command-s` for Mac and `Ctrl-s` for Windows.
- To undo edits within a cell, hit `Command-z` for Mac and `Ctrl-z` for Windows.

Help

Another useful feature is the help command. Type any function followed by `?` and run the cell to return a help window. Hit the `x` button to close it.

In [ ]:
```
abs?
```

```
Signature: abs(x, /)
Docstring: Return the absolute value of the argument.
Type:      builtin_function_or_method
```

Floats and Integers

Doing math in Python is easy, but note that there are `int` and `float` types in Python. In Python 3, integer division returns the same results as floating point division.

In [ ]:
```
59 / 87
```

Out[ ]:
```
0.6781609195402298
```

In [ ]:
```
59 / 87.0
```

Out[ ]:
```
0.6781609195402298
```

Strings

- Double quotes and single quotes are the same thing.
- `'+'` concatenates strings.

In [ ]:
```
# This is a comment.
"Hi " + 'Bye'
```

Out[ ]:
```
'Hi Bye'
```

Printing

Here are some fancy ways of printing:

In [ ]:
```
speed_of_light = 299792458
speed_of_sound = 343
```

In [ ]:
```
print("Light travels at %d meters per second. This can be formatted as: %.2E meters per
```

```
Light travels at 299792458 meters per second. This can be formatted as: 3.00E+08 meters
per second.
```

In [ ]:
```
print ("The speed of sound is {0} meters per second. That is {1}% the speed of light."
    .format(speed_of_sound, speed_of_sound / speed_of_light * 100))
```

```
The speed of sound is 343 meters per second. That is 0.00011441248465296614% the speed o
f light.
```

In [ ]:
```
print("Good Luck! Prepare to work hard and learn a lot of cool stuff!")
```

```
Good Luck! Prepare to work hard and learn a lot of cool stuff!
```

Lists

A list is a mutable array of data, i.e. it can constantly be modified. See http://stackoverflow.com/questions/8056130/immutable-vs-mutable-types-python for more info. If you are not careful, using mutable data structures can lead to bugs in code that passes common data to many different functions.

Important functions:

- Created a list by using square brackets `[ ]`.
- `'+'` appends lists.
- `len(x)` gets the length of list `x`.

In [ ]:
```
x = [1, 2, "asdf"] + [4, 5, 6]

print(x)
```

```
[1, 2, 'asdf', 4, 5, 6]
```

In [ ]:

```
print(len(x))
```

```
6
```

Tuples

A tuple is an immutable list. They can be created using round brackets ( ).

They are usually used as inputs and outputs to functions.

In [ ]:

```
t = (1, 2, "asdf") + (3, 4, 5)
print(t)
```

```
(1, 2, 'asdf', 3, 4, 5)
```

In [ ]:

```
# cannot do assignment
# t[0] = 10

# errors in Jupyter Notebook appear inline
```

Arrays (NumPy)

A NumPy array is like a list with multidimensional support and more functions. We will be using it a lot.

Arithmetic operations on NumPy arrays correspond to elementwise operations.

Important functions:

- `.shape` returns the dimensions of the array.
- `.ndim` returns the number of dimensions.
- `.size` returns the number of entries in the array.
- `len()` returns the first dimension.

To use functions in NumPy, we have to import NumPy to our workspace. This is done by the command `import numpy`. By convention, we rename `numpy` as `np` for convenience.

In [ ]:

```
# by convention, import numpy as np
import numpy as np
x = np.array([[1, 2, 3], [4, 5, 6]])

print(x)
```

```
[[1 2 3]
 [4 5 6]]
```

```
In [ ]: print("Number of Dimensions:", x.ndim)
```

```
Number of Dimensions: 2
```

```
In [ ]: print("Dimensions:", x.shape)
```

```
Dimensions: (2, 3)
```

```
In [ ]: print("Size:", x.size)
```

```
Size: 6
```

```
In [ ]: print("Length:", len(x))
```

```
Length: 2
```

```
In [ ]: a = np.array([1, 2, 3])

        print("a = ", a)

        # elementwise arithmetic
        print("a * a = ", a * a)
```

```
a =  [1 2 3]
a * a =  [1 4 9]
```

```
In [ ]: b = np.array(np.ones((3, 3))) * 2
        print("b =\n", b)
        c = np.array(np.ones((3, 3)))
        print("c =\n", c)
```

```
b =
 [[2. 2. 2.]
 [2. 2. 2.]
 [2. 2. 2.]]
c =
 [[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

Multiply elementwise:

```
In [ ]: print("b * c =\n", b * c)
```

```
b * c =
 [[2. 2. 2.]
 [2. 2. 2.]
 [2. 2. 2.]]
```

Now multiply as matrices (not arrays):

```
In [ ]: print("b * c =\n", np.dot(b, c))
```

```
b * c =
 [[6. 6. 6.]
 [6. 6. 6.]
 [6. 6. 6.]]
```

With Python3, you can also use the "@" operator for the dot product

In [ ]:
```
print("b * c =\n", b @ c)
```

```
b * c =
 [[6. 6. 6.]
 [6. 6. 6.]
 [6. 6. 6.]]
```

Slicing for NumPy Arrays

NumPy uses pass-by-reference semantics so it creates views into the existing array, without implicit copying. This is particularly helpful with very large arrays because copying can be slow.

In [ ]:
```
x = np.array([1, 2, 3, 4, 5, 6])
print(x)
```

```
[1 2 3 4 5 6]
```

We slice an array from `a` to `b - 1` with `[a:b]`.

In [ ]:
```
y = x[0:4]
print(y)
```

```
[1 2 3 4]
```

Since slicing does not copy the array, changing `y` changes `x`:

In [ ]:
```
y[0] = 7
print(x)
print(y)
```

```
[7 2 3 4 5 6]
[7 2 3 4]
```

To actually copy `x`, we should use `.copy`:

In [ ]:
```
x = np.array([1, 2, 3, 4, 5, 6])
y = x.copy()
y[0] = 7
print(x)
print(y)
```

```
[1 2 3 4 5 6]
[7 2 3 4 5 6]
```

Plotting

In this class we will use `matplotlib.pyplot` to plot signals and images.

To begin with, we import `matplotlib.pyplot` as `plt` (again for convenience).

In [ ]:

```
import numpy as np
# by convention, we import pyplot as plt
import matplotlib.pyplot as plt


# Using np.arange is very similair to matlabs built in indexing method and python range
x = np.arange(start=0,stop=1,step=0.01)
a = np.exp(-x)
b = np.sin(x * 10.0) / 4.0 + 0.5

# plot in browser instead of opening new windows
%matplotlib inline
```

In [ ]:

```
# Remember that it this gives x=[0,1)
print(x)
```

```
[0.   0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.09 0.1  0.11 0.12 0.13
 0.14 0.15 0.16 0.17 0.18 0.19 0.2  0.21 0.22 0.23 0.24 0.25 0.26 0.27
 0.28 0.29 0.3  0.31 0.32 0.33 0.34 0.35 0.36 0.37 0.38 0.39 0.4  0.41
 0.42 0.43 0.44 0.45 0.46 0.47 0.48 0.49 0.5  0.51 0.52 0.53 0.54 0.55
 0.56 0.57 0.58 0.59 0.6  0.61 0.62 0.63 0.64 0.65 0.66 0.67 0.68 0.69
 0.7  0.71 0.72 0.73 0.74 0.75 0.76 0.77 0.78 0.79 0.8  0.81 0.82 0.83
 0.84 0.85 0.86 0.87 0.88 0.89 0.9  0.91 0.92 0.93 0.94 0.95 0.96 0.97
 0.98 0.99]
```

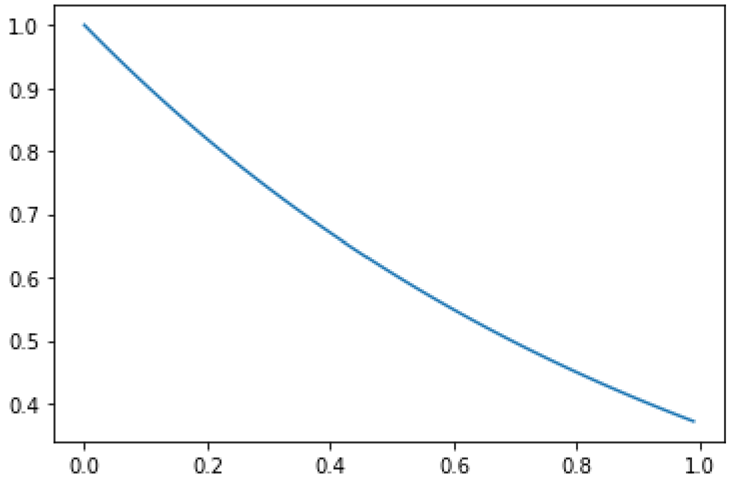
`plt.plot(x, a)` plots `a` against `x`.

In [ ]:

```
plt.figure()
plt.plot(x, a)
```

Out[ ]:

```
[<matplotlib.lines.Line2D at 0x21d782ac160>]
```



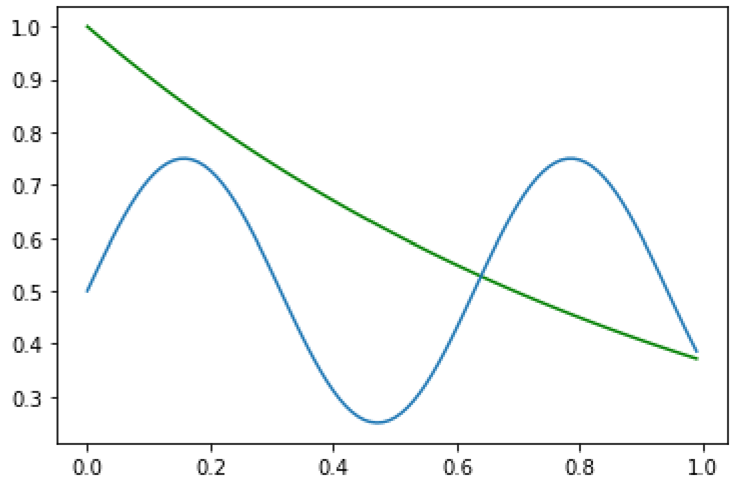
Once you started a figure, you can keep plotting to the same figure.

In [ ]:

```
plt.figure()
plt.plot(x, a, "green")
plt.plot(x, b)
```

Out[ ]:

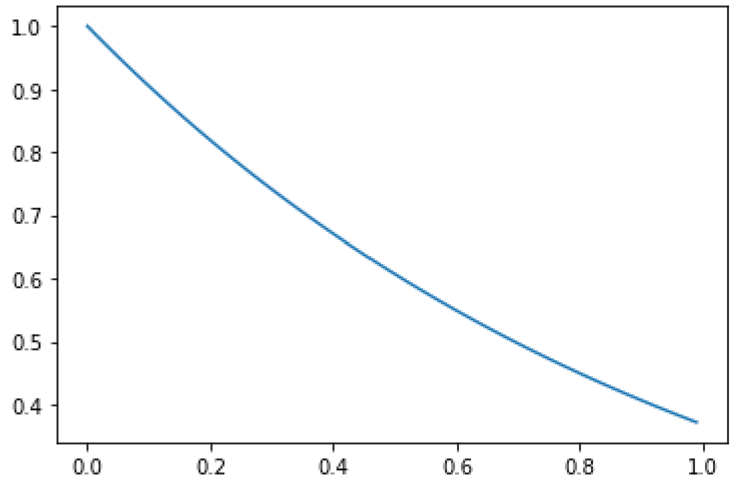
```
[<matplotlib.lines.Line2D at 0x21d783a0b80>]
```

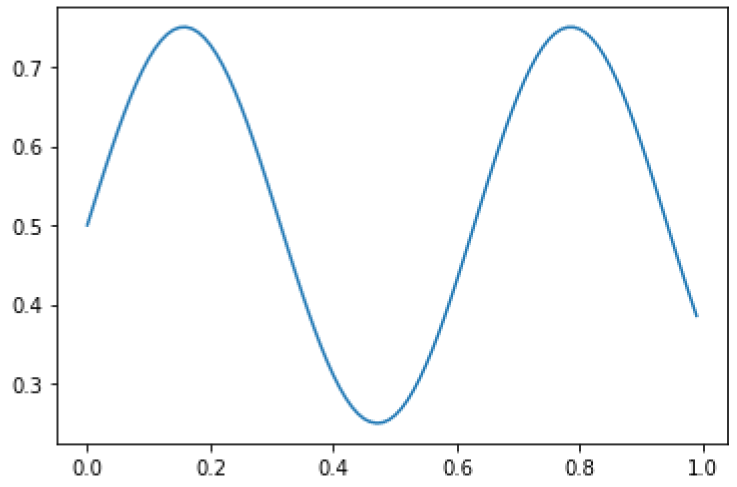


To plot different plots, you can create a second figure.

```
In [ ]:
plt.figure()
plt.plot(x, a)
plt.figure()
plt.plot(x, b)
```

```
Out[ ]: [<matplotlib.lines.Line2D at 0x21d78450f70>]
```

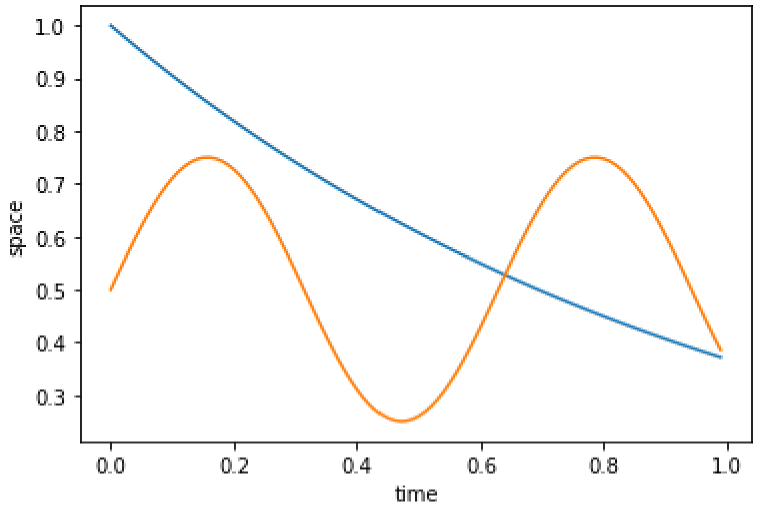




To label the axes, use `plt.xlabel()` and `plt.ylabel()` .

In [ ]:
```
plt.figure()
plt.plot(x, a)
plt.plot(x, b)

plt.xlabel("time")
plt.ylabel("space")
```

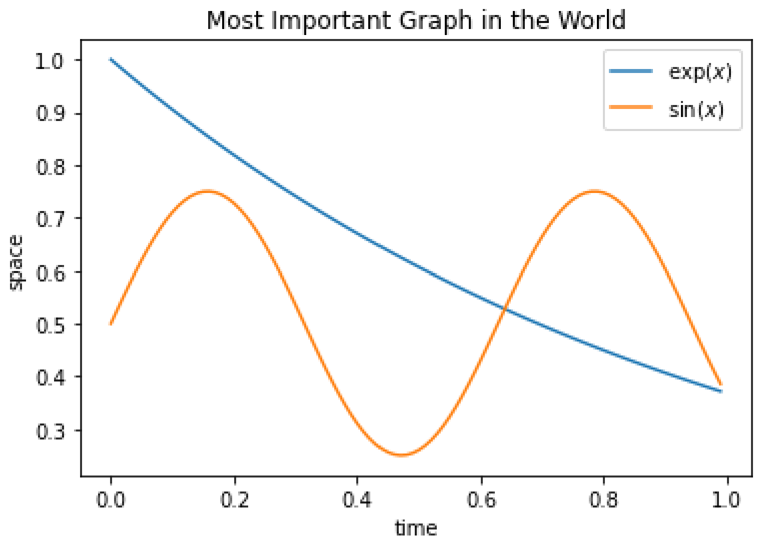
Out[ ]:
```
Text(0, 0.5, 'space')
```



You can also add title and legends using `plt.title()` and `plt.legend()` .

In [ ]:
```
plt.figure()
plt.plot(x, a)
plt.plot(x, b)
plt.xlabel("time")
plt.ylabel("space")

plt.title("Most Important Graph in the World")

plt.legend(("$\exp(x)$", "$\sin(x)$"))
```

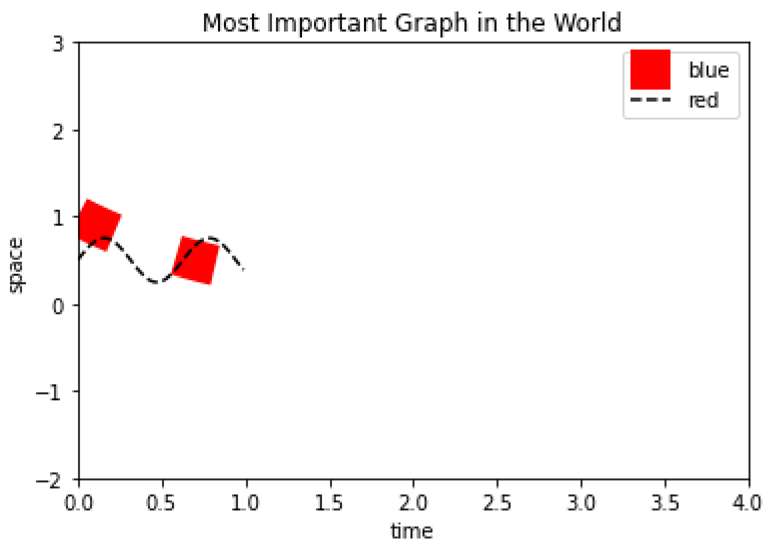
Out[ ]:
```
<matplotlib.legend.Legend at 0x21d78516eb0>
```



There are many options you can specify in `plot()` , such as color and linewidth. You can also change the axis using `plt.axis`.

In [ ]:
```
plt.figure()
plt.plot(x, a, ":r", linewidth=20)
plt.plot(x, b , "--k")
plt.xlabel("time")
plt.ylabel("space")

plt.title("Most Important Graph in the World")

plt.legend(("blue", "red"))

plt.axis([0, 4, -2, 3])
```

Out[ ]:
```
(0.0, 4.0, -2.0, 3.0)
```

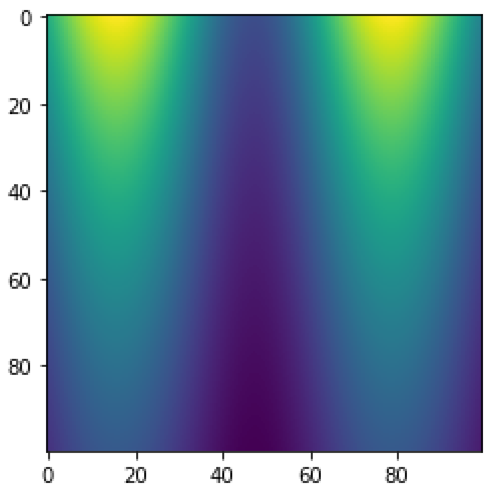


There are many other plotting functions. For example, we will use `plt.imshow()` for showing images and `plt.stem()` for plotting discretized signals.

In [ ]:
```
# image
plt.figure()

# plotting the outer product of a and b
data = np.outer(a, b)
# This returns a 100x100 matrix
print(data.shape)
plt.imshow(data)
```

```
(100, 100)
```

Out[ ]:
```
<matplotlib.image.AxesImage at 0x21d787161f0>
```

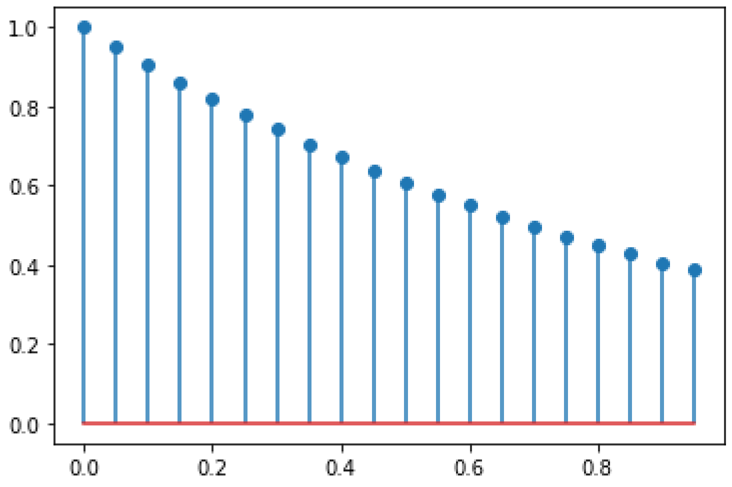


In [ ]:

```
# stem plot
plt.figure()
# subsample by 5
plt.stem(x[::5], a[::5])
```

Out[ ]:

```
<StemContainer object of 3 artists>
```

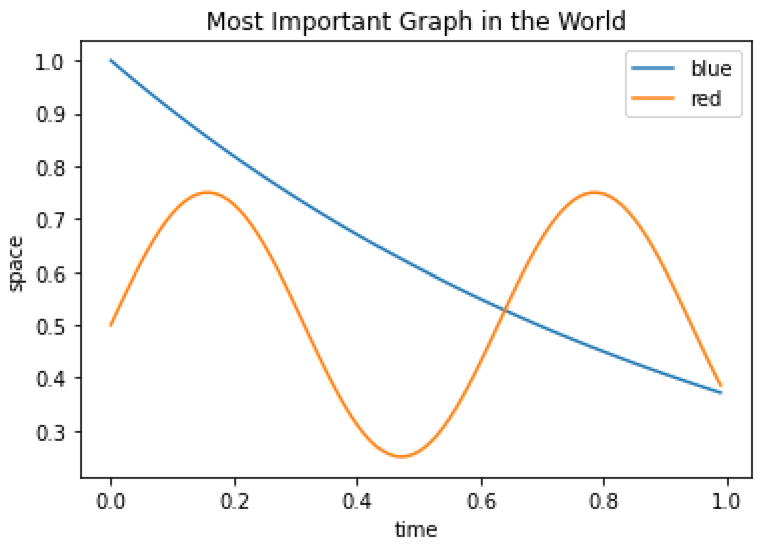


In [ ]:

```
plt.plot(x, a)
plt.plot(x, b)
plt.xlabel("time")
plt.ylabel("space")

plt.title("Most Important Graph in the World")

plt.legend(("blue", "red"))
```

Out[ ]:

```
<matplotlib.legend.Legend at 0x21d78744a00>
```



Logic

For Loop

Indentation matters in Python. Everything indented belongs to the loop:

In [ ]:

```
for i in [4, 6, "asdf", "jkl"]:
    print(i)
```

```
4
6
asdf
jkl
```

In [ ]:

```
for i in np.arange(0, 1, 0.1):
    print(i)
```

```
0.0
0.1
0.2
0.30000000000000004
0.4
0.5
0.6000000000000001
0.7000000000000001
0.8
0.9
```

If-Else

Same goes for If-Else:

In [ ]:

```
if 1 != 0:
    print("1 != 0")
elif 1 == 0:
    print("1 = 0")
```

```
else:
    print("Huh?")
```

```
1 != 0
```

Random Library

The NumPy random library should be your resource for all Monte Carlo simulations which require generating instances of random variables.

The documentation for the library can be found here:
https://numpy.org/doc/stable/reference/random/

This documentation has been changed in the past year, with the new methods relying on explicit use of the Generator class provided by NumPy. You may rely on the legacy versions relying on the RandomState class, but here I will outline the new version.

Call 'default_rng' to get a new instance of a Generator (default generator is the BitGenerator), then call its methods to oibtain samples from different distributions.

For example, if we wanted to sample from a uniform distribution in the range of $[0, 1)$:

In [ ]:

```
# Do this (new version)
from numpy.random import default_rng
# we can explicitly state the seed for the generator
seed = 23
rng = default_rng(seed)

# random number
print(rng.random())
# random vector
print(rng.random(5))
# random matrix
print(rng.random([3,3]))
```

```
0.6939330806573643
[0.64145822 0.12864422 0.11370805 0.65334552 0.85345711]
[[0.20177913 0.21801864 0.71658464]
 [0.47069967 0.41522193 0.3491478 ]
 [0.06385375 0.45466617 0.30145328]]
```

Let's see how we can use this to generate a fair coin toss (i.e. a discrete $\text{Bernoulli}(1/2)$ random variable).

TIP:

Use "Ctrl + Enter" Instead of "Shift + Enter" to excecute a cell without moving down to the next cell.

In [ ]:

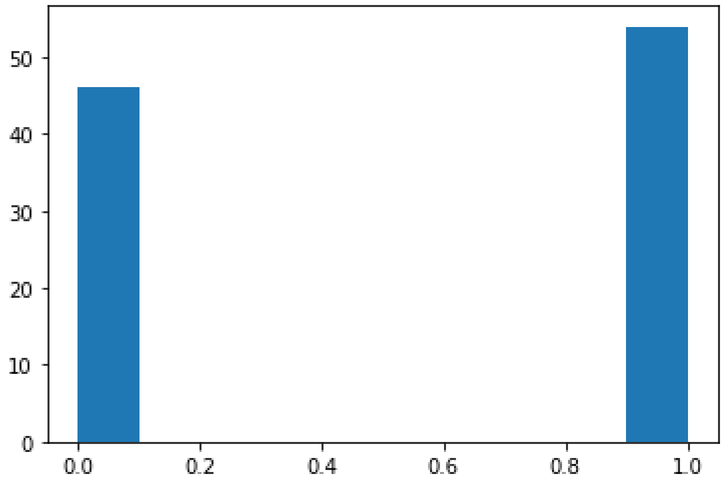
```
# Bernoulli(1/2) random variable
x = round(rng.random())
print(x)
```

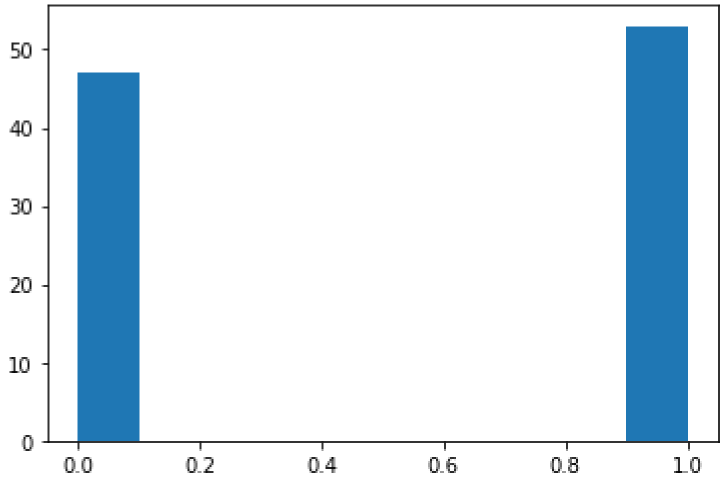
```
0
```

Now let's generate several fair coin tosses and plot a histogram of the results.

In [ ]:
```
k = 100
# _ is commonly used in python when we do not want to reserve memory for a given variab
x1 = [round(rng.random()) for _ in range(k)]
plt.figure()
plt.hist(x1)

# we could also use NumPy's round function to elementwise round the vector.
x2 = np.round(rng.random(k))
plt.figure()
plt.hist(x2)
```

Out[ ]:
```
(array([47.,  0.,  0.,  0.,  0.,  0.,  0.,  0.,  0., 53.]),
 array([0. , 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1. ]),
 <BarContainer object of 10 artists>)
```





We can do something similar for several other distributions, and allow the histogram to give us a sense of what the distribution looks like. As we increase the number of samples we take from the distribution k, the more and more our histogram looks like the actual distribution.

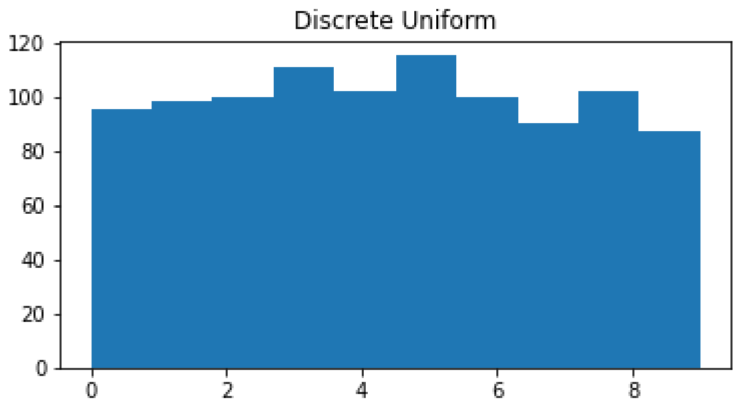
To see a list of distrubutions provided by the Generator method see:
https://numpy.org/doc/stable/reference/random/generator.html#numpy.random.Generator

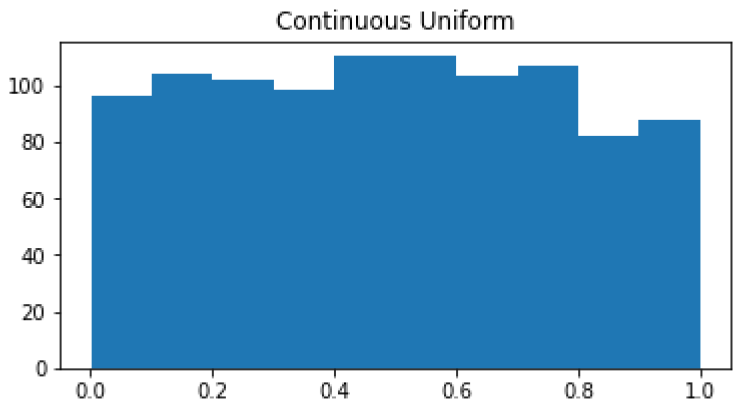
In [ ]:
```
k = 1000
```

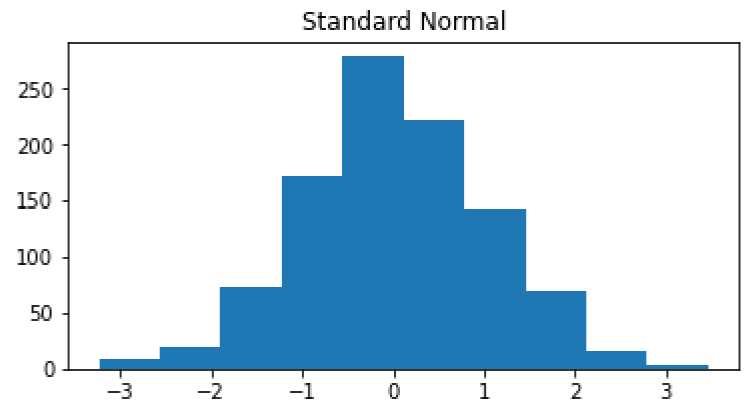
```
# k discrete uniform random variables between 0 and 9
discrete_uniform = rng.integers(0, 10, size=k)
plt.figure(figsize=(6, 3))
plt.hist(discrete_uniform)
plt.title("Discrete Uniform")

continuous_uniform = rng.random(k)
plt.figure(figsize=(6, 3))
plt.hist(continuous_uniform)
plt.title("Continuous Uniform")

# standard_normal replaces the legacy randn method
std_normal = rng.standard_normal(k)
plt.figure(figsize=(6, 3))
plt.hist(std_normal)
plt.title("Standard Normal")

# To generate a normal distribution with mean mu and standard deviation sigma,
# we must mean shift and scale the variable
mu = 100
sigma = 40
normal_mu_sigma = mu + rng.standard_normal(k) * sigma
plt.figure(figsize=(6, 3))
plt.hist(normal_mu_sigma)
plt.title("N({}, {})".format(mu, sigma))
```

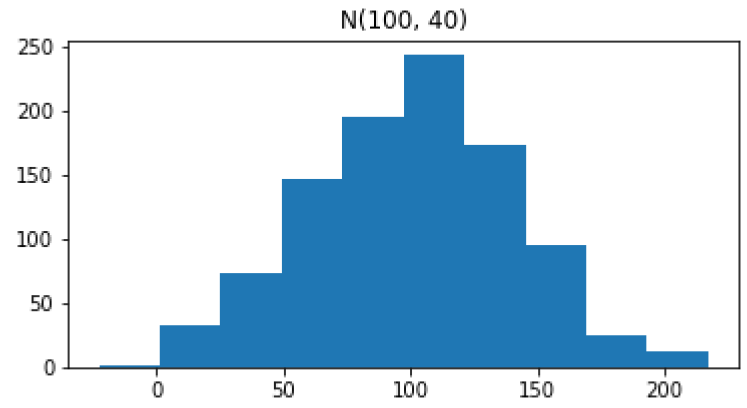
Out[ ]:

```
Text(0.5, 1.0, 'N(100, 40)')
```









^ We could do this all day with all sorts of distributions. I think you get the point.

Specifying a Discrete Probability Distribution for Monte Carlo Sampling

The following function takes n sample from a discrete probability distribution specified by the two arrays `distribution` and `values` .

As an example, let us suppose a random variable X follows the following distribution:

$$X = \begin{cases} 1 \text{ w/ probability } 0.1 \\ 2 \text{ w/ probability } 0.4 \\ 3 \text{ w/ probability } 0.2 \\ 4 \text{ w/ probability } 0.2 \\ 5 \text{ w/ probability } 0.05 \\ 6 \text{ w/ probability } 0.05 \end{cases}$$

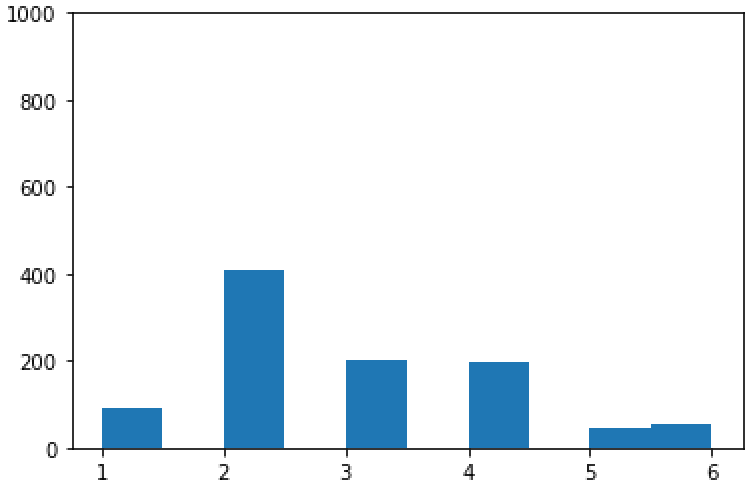
Then we would have: `distribution = [0.1, 0.4, 0.2, 0.2, 0.05, 0.05]` and `values = [1, 2, 3, 4, 5, 6]` .

In [ ]:
```
# Note how default variables are defined in python functions, they must be placed last.
def n_sample(distribution, values, n, seed=123):
    if sum(distribution) != 1:
        print("ERROR: elements of input: 'distribution' should sum to 1")
        return
    rng = default_rng(seed)
    # This method can be implemented with the numpy's choice method
    samples = rng.choice(a=values,size=n,replace=True,p=distribution)
    return samples
```

In [ ]:

```
# collect k samples from X and plot the histogram
samples_from_x = n_sample(
    [0.1, 0.4, 0.2, 0.2, 0.05, 0.05], [1, 2, 3, 4, 5, 6], k)
plt.hist(samples_from_x)
plt.ylim((0, 1000))
print("Wow, if we normalized the y-axis that would be a PMF. Incredible!")
print("I should try that.")
```

```
Wow, if we normalized the y-axis that would be a PMF. Incredible!
I should try that.
```

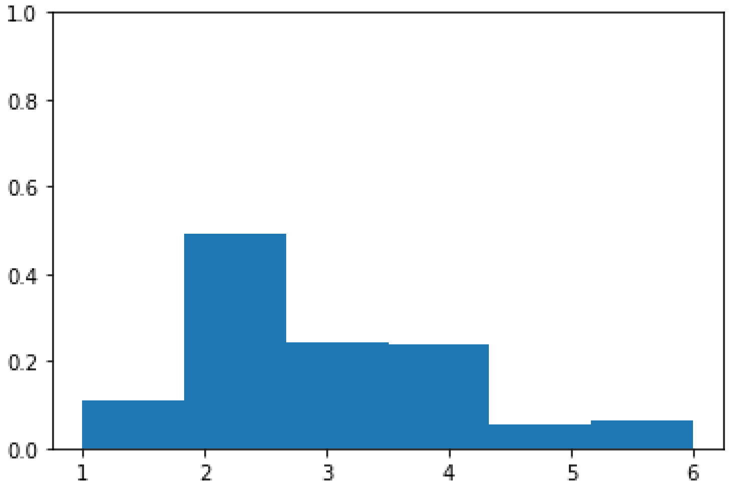


In [ ]:

```
# Normalize the samples from X to plot the probability mass function below:
plt.hist(samples_from_x, bins=6, density=True)
plt.ylim((0, 1))
```

Out[ ]:

```
(0.0, 1.0)
```



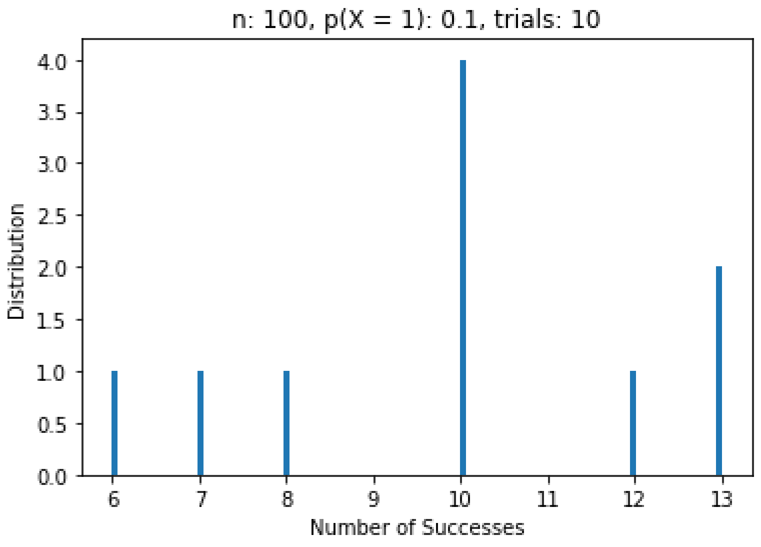
$\mathcal{Q}$uestion 1: Sampling and Plotting a Binomial Random Variable

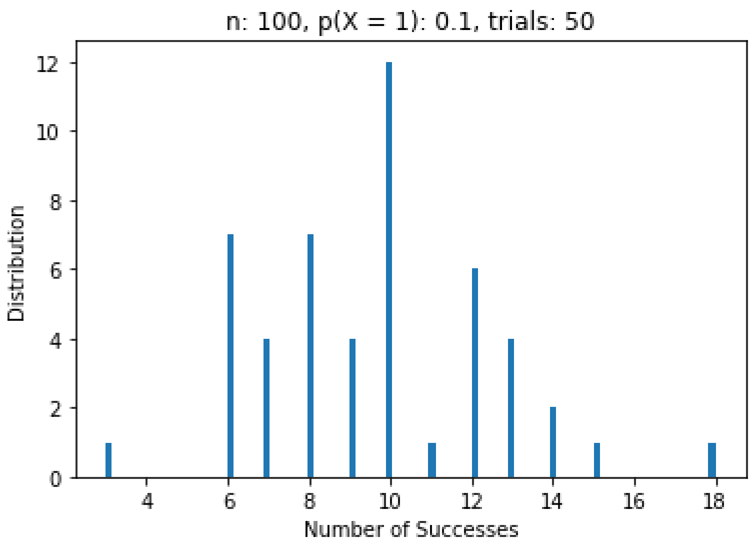
A binomial random variable $X \sim \text{Binomial}(n, p)$ can be thought of as the number of heads in n coin flips where each flip has probability p of coming up heads. We can equivalently think of it as the sum of n Bernoulli random variables: $X = \sum_{i=1}^{n} X_i$ where $X_i \sim \text{Bernoulli}(p)$.

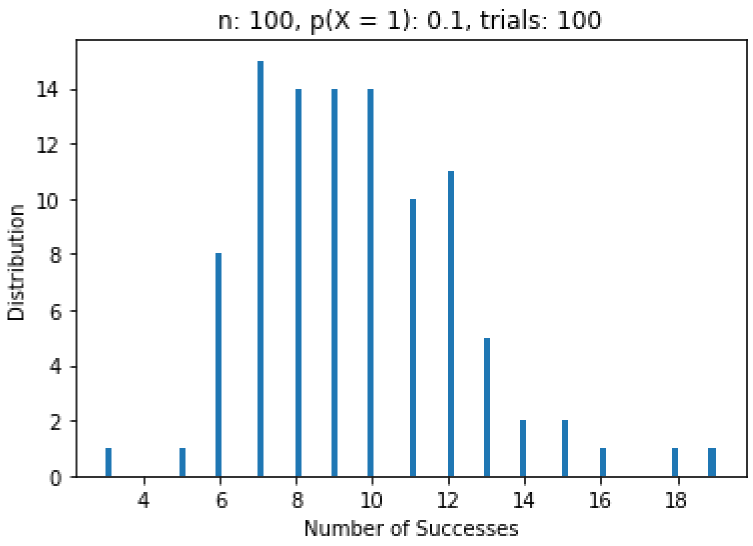
In this question, you will put your new plotting skills to work and sample the values of a binomial random variable.

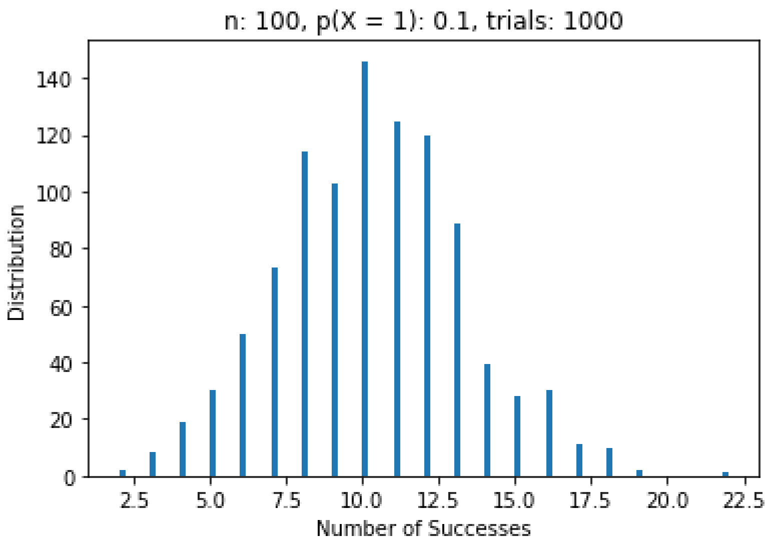
In [ ]:

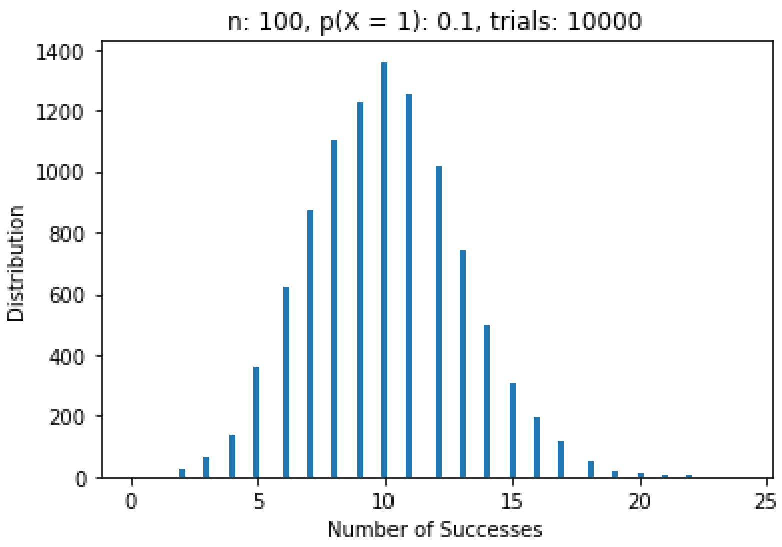
```
def plot_binomial(
        n, trials=[10, 50, 100, 1000, 10000], p_values=[0.1, 0.2, 0.5, 0.8], seed = 123
    """
    On different figures, plot a histogram of the results of the given
    number of trials of the binomial variable with parameters n and p for all
    values in the given list.

    You should generate 20 different plots in total.
    """
    # iterate over the probabilities list
    for j,p in enumerate(p_values):
        # iterate over the trials list
        for i,t in enumerate(trials):
            successes = []
            # generate all the trials at once
            outcomes = n_sample(distribution=[1-p_values[j],p_values[j]], values=[0,1],
            # separate into k partitions of size n
            for k in range(trials[i]):
                # for each realization, count the number of successes
                successes.append(np.sum(outcomes[k*n:k*n+n]))

            # generate the plots
            plt.figure()
            plt.hist(successes,bins=n)
            plt.xlabel("Number of Successes")
            plt.ylabel("Distribution")
            plt.title("n: " + str(n) + ", p(X = 1): " + str(p_values[j]) + ", trials: "


# Feel free to play around with other values of n.
plot_binomial(100)
```

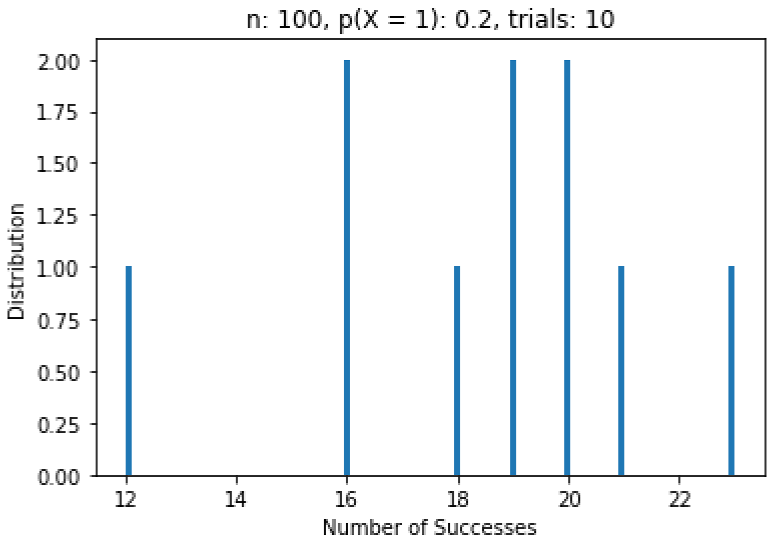


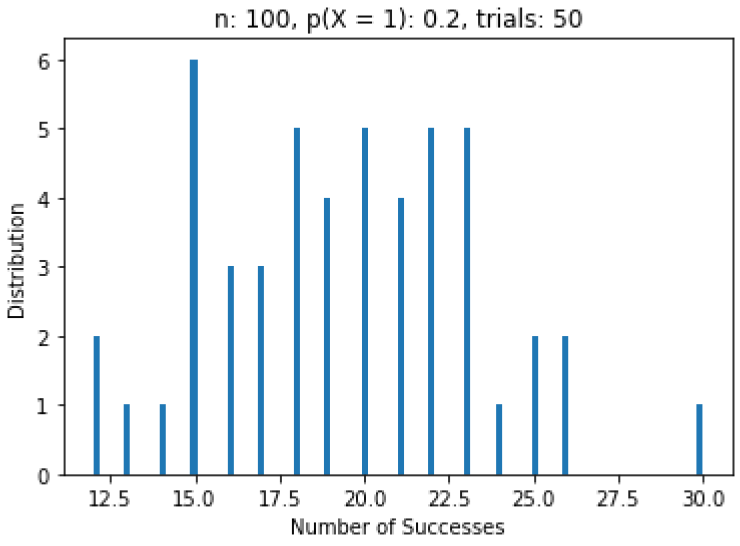

n: 100, p(X = 1): 0.1, trials: 50
Distribution
Number of Successes

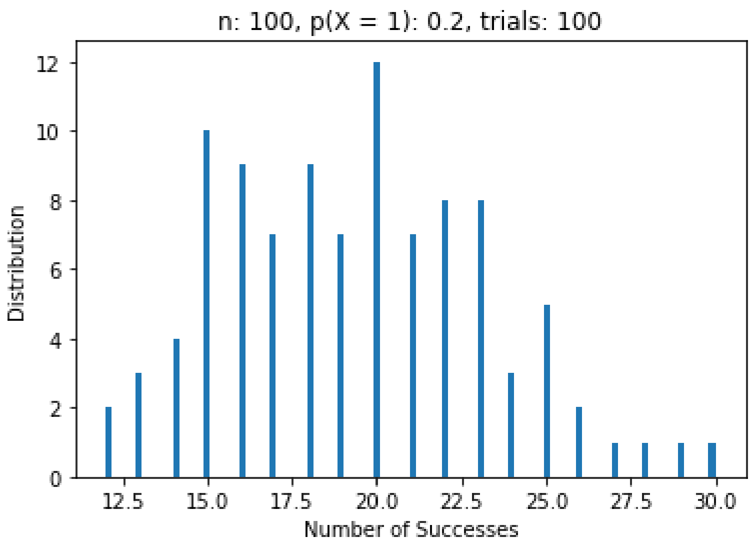

n: 100, p(X = 1): 0.1, trials: 100
Distribution
Number of Successes

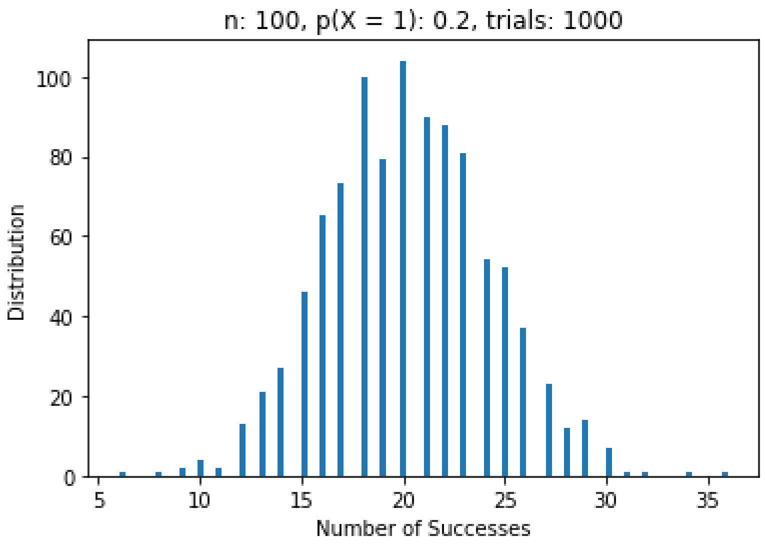

n: 100, p(X = 1): 0.1, trials: 1000
Distribution
Number of Successes

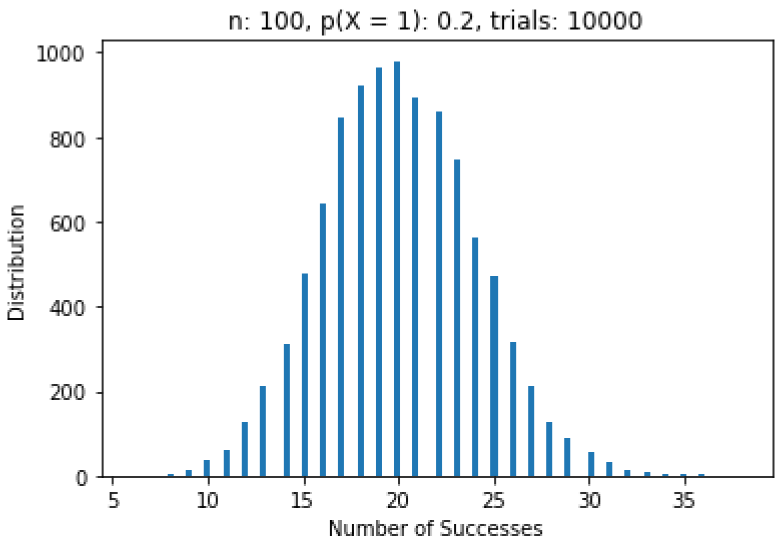

n: 100, p(X = 1): 0.1, trials: 10000
Distribution
Number of Successes

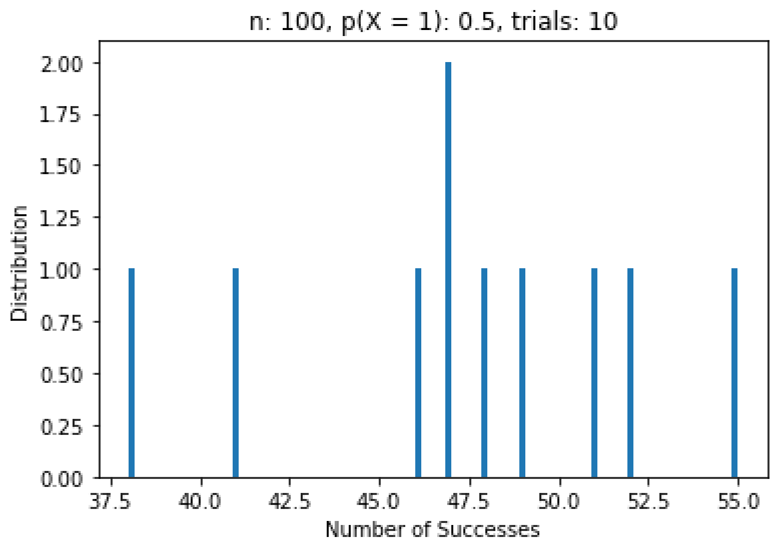

n: 100, p(X = 1): 0.2, trials: 10
Distribution
Number of Successes

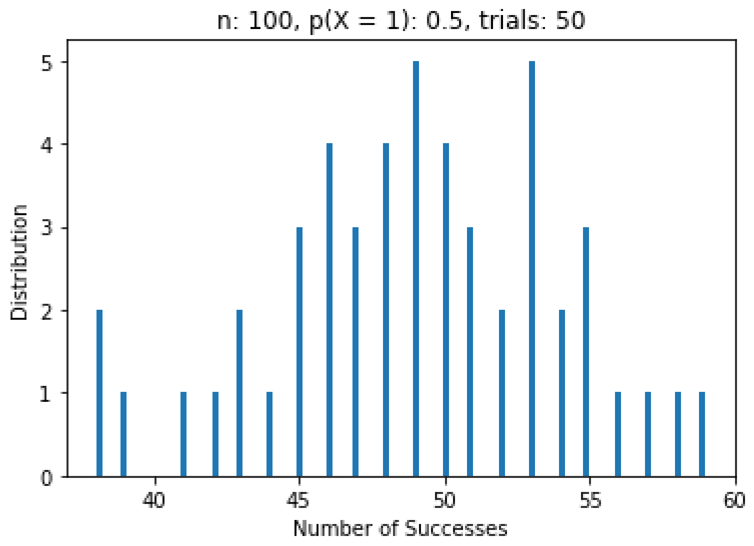

n: 100, p(X = 1): 0.2, trials: 50
Distribution
Number of Successes

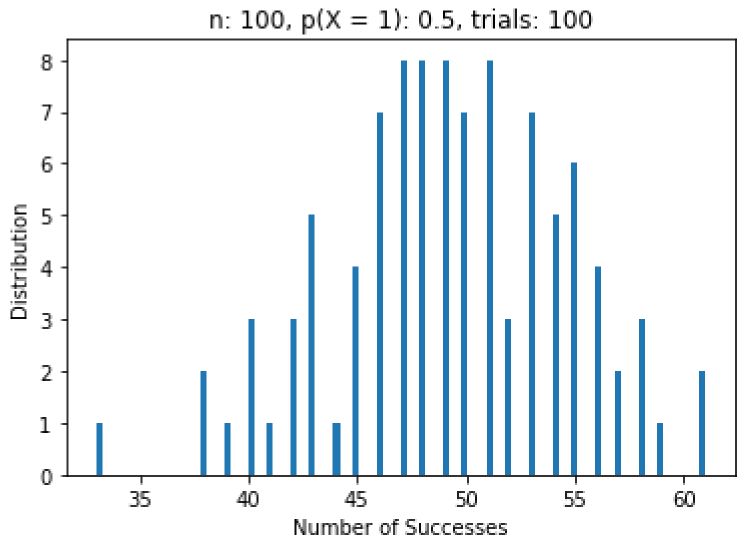

n: 100, p(X = 1): 0.2, trials: 100
Distribution
Number of Successes
0
2
4
6
8
10
12
12.5
15.0
17.5
20.0
22.5
25.0
27.5
30.0

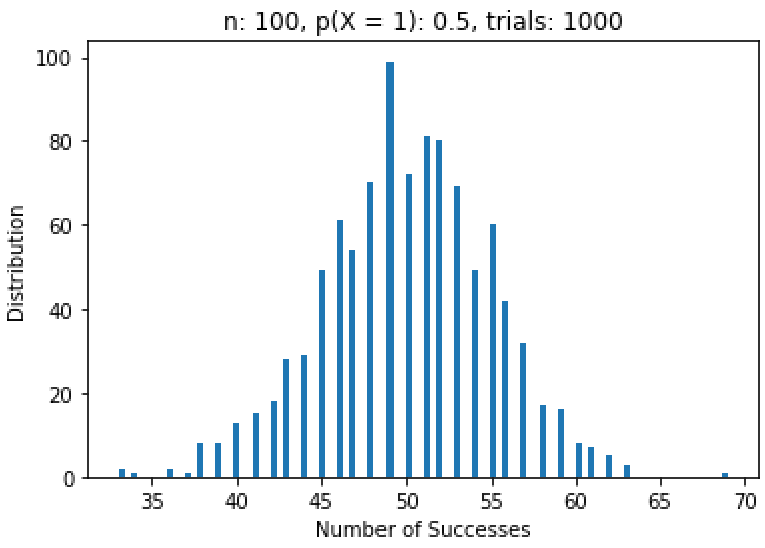

n: 100, p(X = 1): 0.2, trials: 1000
Distribution
Number of Successes
0
20
40
60
80
100
5
10
15
20
25
30
35

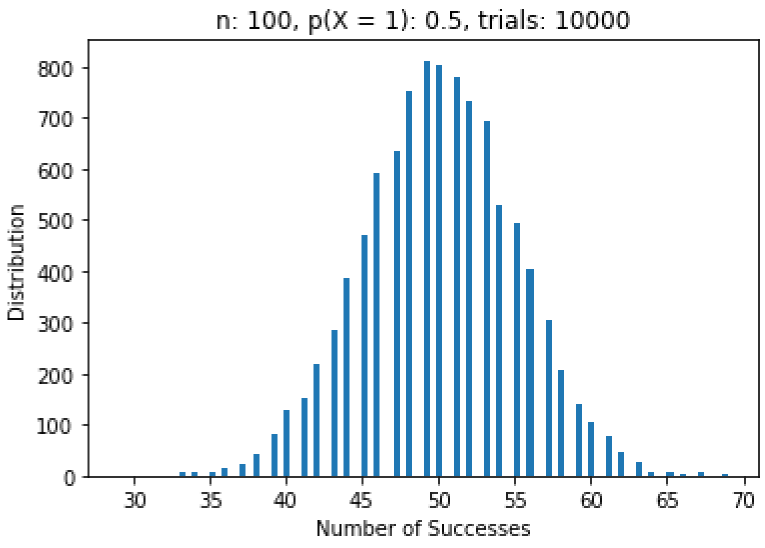

n: 100, p(X = 1): 0.2, trials: 10000
Distribution
Number of Successes
0
200
400
600
800
1000
5
10
15
20
25
30
35

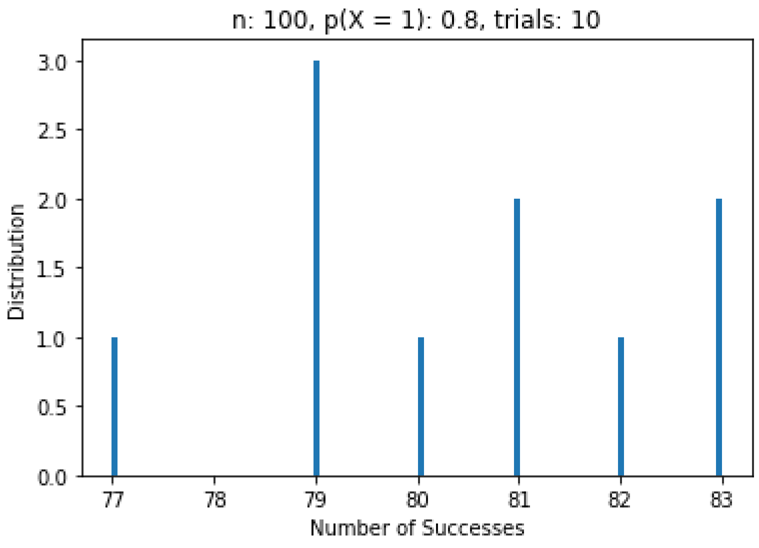

n: 100, p(X = 1): 0.5, trials: 10
Distribution
Number of Successes
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
37.5
40.0
42.5
45.0
47.5
50.0
52.5
55.0

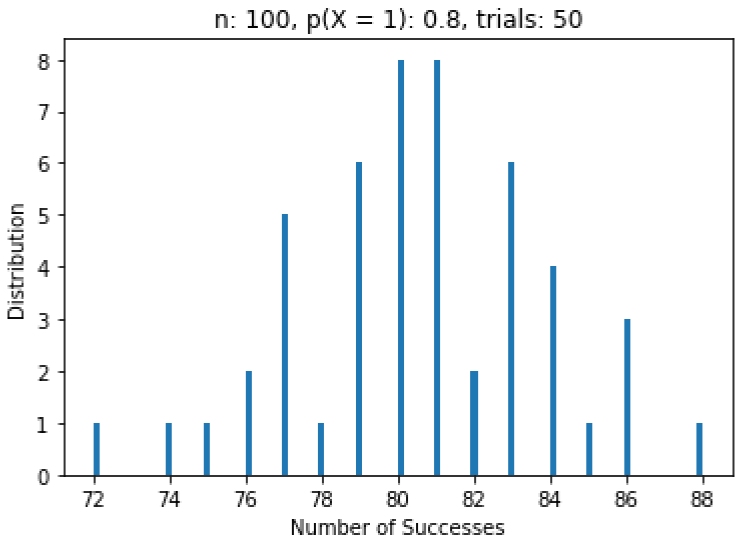

n: 100, p(X = 1): 0.5, trials: 50
Distribution
Number of Successes
0
1
2
3
4
5
40
45
50
55
60

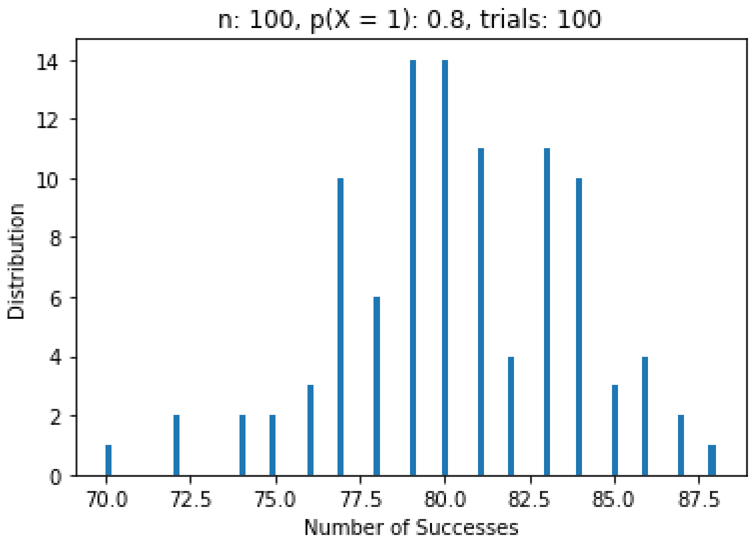

n: 100, p(X = 1): 0.5, trials: 100
Distribution
Number of Successes
0
1
2
3
4
5
6
7
8
35
40
45
50
55
60

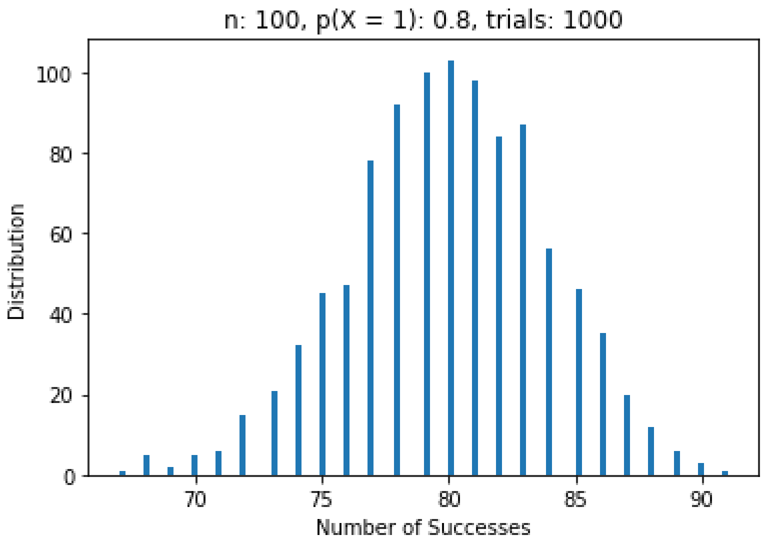

n: 100, p(X = 1): 0.5, trials: 1000
Distribution
Number of Successes
0
20
40
60
80
100
35
40
45
50
55
60
65
70

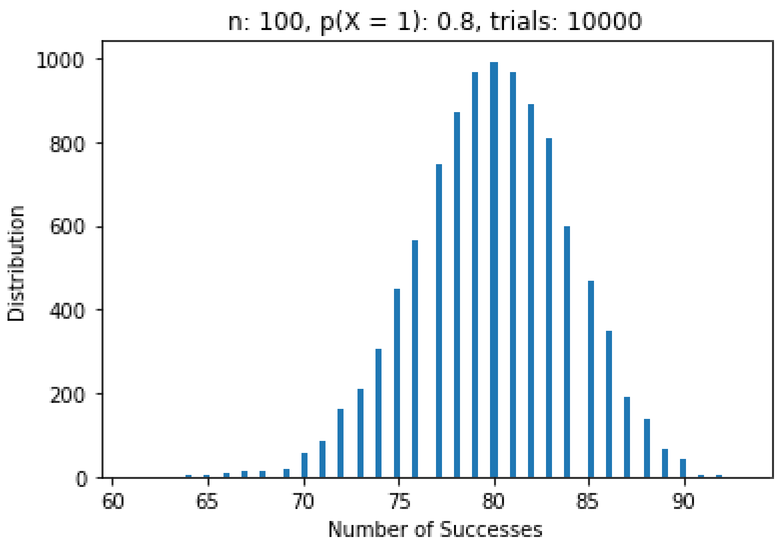

n: 100, p(X = 1): 0.5, trials: 10000
Distribution
Number of Successes
0
100
200
300
400
500
600
700
800
30
35
40
45
50
55
60
65
70


n: 100, p(X = 1): 0.8, trials: 10
Distribution
Number of Successes
0.0
0.5
1.0
1.5
2.0
2.5
3.0
77
78
79
80
81
82
83


n: 100, p(X = 1): 0.8, trials: 50
Distribution
Number of Successes
72
74
76
78
80
82
84
86
88
0
1
2
3
4
5
6
7
8


n: 100, p(X = 1): 0.8, trials: 100
Distribution
Number of Successes
70.0
72.5
75.0
77.5
80.0
82.5
85.0
87.5
0
2
4
6
8
10
12
14


n: 100, p(X = 1): 0.8, trials: 1000
Distribution
Number of Successes
70
75
80
85
90
0
20
40
60
80
100



Now that you have plotted many values of a few different binomial random variables, do the results coincide with what you expect them to?

Yes, in the plots we see that as we increase the number of samples, we get a Gaussian shape centered around the expected value of the given binomial. For example, for (n,p) = (100,0.2) the expected value is n*p = 100*0.2 = 20. In the plot for trials = 10,000 we see that the mean is around the 20.0 mark.

$\mathcal{Q}$uestion 2: Monte Carlo Method for Estimating Coin Flips

After going through this tutorial, you might wonder: why should we bother with NumPy at all?

While many of you may not have used NumPy previously, or not see the purpose in learning this library, we strongly urge you to force yourself to use NumPy as much as possible while doing virtual labs.

NumPy (and using matrix operations rather than loops) is your friend when it comes to efficiently dealing with lots of data or doing elaborate simulations. If you find yourself using many loops or list comprehensions to process data, think about using NumPy. Furthermore, NumPy is widely used in industry and academic research, and so it will benefit you greatly to become comfortable with it this semester!

Let's work through an example to see the usefulness of NumPy.

Estimate the Probability of Having More than 55 Heads in 100 Flips of a Fair Coin.

Monte Carlo methods are algorithms that use probability and randomness to solve problems that would be difficult otherwise. Here, we will use a Monte Carlo simulation to approximate a coin toss experiment.

Suppose we have a a fair coin. We want to find out what is the probability that out of 100 tosses of our coin, **more than 55** of them will be heads. Of course the exact answer can be found using the discrete probabilty mass function of the binomail distribution accordingly, but here we want to leverage computational techniques in order to approximate the probability of the given event.

Lets denote this event as A : "The Event of Getting **More Than 55** Heads out of 100 Tosses of a Fair Coin", where our goal is to approximate $P(A)$

One way to find this probability is to run the experiment multiple times:

$$P(A) \approx \frac{\text{num of times event A occured}}{\text{num of times experiment was run}}$$

As we run the experiment more times, we get a better estimate of $P(A)$

A version of the simulation that does not use NumPy at all is given. Your job is to re-implement the simulation using NumPy to speed it up. For reference, the staff solution is > 10x better.

As a final question:

After computing the probability with Monte Carlo simulation, compare it to the result derived when applying the Central Limit Theorem to our problem.

In [ ]:
```
import time
import random

def monte_carlo_coin(N):
    '''
    This function will return an estimate of P(A) defined above.

    Inputs:
        N - Number of times the experiment was run
    '''
    count = 0
    for _ in range(N):
        num_heads = 0
        for _ in range(100):
            coin = random.random()
            if coin > 0.5:
                num_heads += 1
        if num_heads > 55:
            count += 1

    return count/N


def monte_carlo_coin_numpy(N, seed=12345):
    rng = default_rng(seed)

    # generate all the outcomes at once as a matrix
    outcomes = rng.integers(2,size=(N,100))

    # sum the rowss (each row is 100 tosses)
    counts = outcomes.sum(axis=1)
    # counts the number of rows greater than 55
    return np.sum(counts > 55)/N
```

To see the effectiveness of using NumPy, let's run both implementations of the simulation, with and without NumPy, and compare their speeds.

In [ ]:
```
# number of experiments to run: N
N = 100000
exact_solution = 0.13562651204
```

In [ ]:
```
start = time.perf_counter()
print("Estimate of P(A):", monte_carlo_coin(N))
print ("Actual value of P(A):", exact_solution)
end = time.perf_counter()
total1 = end - start
```

```
Estimate of P(A): 0.13715
Actual value of P(A): 0.13562651204
```

In [ ]:
```
start = time.perf_counter()
print("Estimate of P(A):", monte_carlo_coin_numpy(N))
print ("Actual value of P(A):", exact_solution)
end = time.perf_counter()
total2 = end - start
```

```
Estimate of P(A): 0.13589
Actual value of P(A): 0.13562651204
```

In [ ]:
```
print("w/o NumPy:\t %f s\nw/ NumPy:\t %f s" %(total1, total2))
print("Total Speedup: " + str(total1 / total2) + "x")
```

```
w/o NumPy:       1.909086 s
w/ NumPy:        0.116326 s
Total Speedup: 16.411462918142867x
```

As we can see, the result is close to the true value derived from the Binomail Distribution. What about using the Central Limit Theorem directly? How can we apply it to this setting, and is the result similair?

We want to find $P[X \geq 55] = \sum_{i=55}^{100} \binom{100}{i} (0.5)^i (0.5)^{100-i}$

Rather than calculating this directly, we can use central limit theorem to approximate this value:

$$\lim_{n\to\infty} P[\frac{X_1+\ldots+X_n-n\mu}{\sqrt{n}\sigma}] \sim N(0,1)$$

Now we plug in the μ and σ for this example:

$$\mu = 0.5$$

$$\sigma^2 = p(1-p) = 0.5(0.5) = 25 \rightarrow \sigma = 0.5$$

$$P[X_1+\ldots+X_{100} \geq 56] \approx P[\frac{X_1+\ldots+X_{100}-100*0.5}{\sqrt{100*0.5}} \geq \frac{56-50}{5}]$$

$= P[N(0,1) \geq 1.2]$ We can evaluate numerically using the Q function to get Q(1.2) = 0.115 which is close to the answer we got above.