**Random float**

Randomness has many uses in science, art, statistics, cryptography, gaming, gambling, and other fields. You're going to use randomness to simulate a game.

All the functionality you need is contained in the random package, a sub-package of numpy. In this exercise, you'll be using two functions from this package:

* [**seed()**](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.seed.html): sets the random seed, so that your results are the reproducible between simulations. As an argument, it takes an integer of your choosing. If you call the function, no output will be generated.
* [**rand()**](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.rand.html): if you don't specify any arguments, it generates a random float between zero and one.

##### INSTRUCTIONS

* Import numpy as np.
* Use [**seed()**](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.seed.html) to set the seed; as an argument, pass 123.
* Generate your first random float with [**rand()**](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.rand.html) and print it out.

# Roll the dice

In the previous exercise, you used [**rand()**](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.rand.html), that generates a random float between 0 and 1.

##### INSTRUCTIONS

* Use **[randint()](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.randint.html" \t "_blank)** with the appropriate arguments to randomly generate the integer 1, 2, 3, 4, 5 or 6. This simulates a dice. Print it out.
* Repeat the outcome to see if the second throw is different. Again, print out the result.

# Determine your next move

In the Empire State Building bet, your next move depends on the number of eyes you throw with the dice. We can perfectly code this with an if-elif-else construct!

##### INSTRUCTIONS

* Roll the dice. Use **[randint()](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.randint.html" \t "_blank)** to create the variable dice.
* Finish the if-elif-else construct by replacing \_\_\_:
* If dice is 1 or 2, you go one step down.
* if dice is 3, 4 or 5, you go one step up.
* Else, you throw the dice again. The number of eyes is the number of steps you go up.
* Print out dice and step. Given the value of dice, was step updated correctly?

# The next step

Before, you have already written Python code that determines the next step based on the previous step. Now it's time to put this code inside a for loop so that we can simulate a random walk.

##### INSTRUCTIONS

* Make a list random\_walk that contains the first step, which is the integer 0.
* Finish the for loop:
* The loop should run 100 times.
* On each iteration, set step equal to the last element in the random\_walk list. You can use the index -1 for this.
* Next, let the if-elif-else construct update step for you.
* The code that appends step to random\_walk is already coded.
* Print out random\_walk.

# How low can you go?

Things are shaping up nicely! You already have code that calculates your location in the Empire State Building after 100 dice throws. However, there's something we haven't thought about - you can't go below 0!

A typical way to solve problems like this is by using [**max()**](https://docs.python.org/3/library/functions.html#max). If you pass [**max()**](https://docs.python.org/3/library/functions.html#max) two arguments, the biggest one gets returned. For example, to make sure that a variable x never goes below 10 when you decrease it, you can use:

x = max(10, x - 1)

##### INSTRUCTIONS

* Use [**max()**](https://docs.python.org/3/library/functions.html#max) in a similar way to make sure that step doesn't go below zero if dice <= 2.

# Visualize the walk

Let's visualize this random walk! Remember how you could use matplotlib to build a line plot?

import matplotlib.pyplot as plt

plt.plot(x, y)

plt.show()

The first list you pass is mapped onto the x axis and the second list is mapped onto the y axis.

If you pass only one argument, Python will know what to do and will use the index of the list to map onto the x axis, and the values in the list onto the y axis.

##### INSTRUCTIONS

Add some lines of code after the for loop:

* Import matplotlib.pyplot as plt.
* Use **[plt.plot()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.plot" \t "_blank)** to plot random\_walk.
* Finish off with **[plt.show()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.show" \t "_blank)** to actually display the plot.

# Simulate multiple walks

A single random walk is one thing, but that doesn't tell you if you have a good chance at winning the bet.

To get an idea about how big your chances are of reaching 60 steps, you can repeatedly simulate the random walk and collect the results. That's exactly what you'll do in this exercise.

The sample code already puts you in the right direction. Another for loop is wrapped around the code you already wrote. It's up to you to add some bits and pieces to make sure all results are recorded correctly.

##### INSTRUCTIONS

* Initialize all\_walks to an empty list.
* Fill in the specification of the for loop so that the random walk is simulated 10 times.
* At the end of the top-level for loop, append random\_walk to the all\_walks list.
* Finally, after the top-level for loop, print out all\_walks.

# Initialization

import numpy as np

np.random.seed(123)

# Initialize all\_walks

all\_walks=[]

# Simulate random walk 10 times

for i in range(10) :

# Code from before

random\_walk = [0]

for x in range(100) :

step = random\_walk[-1]

dice = np.random.randint(1,7)

if dice <= 2:

step = max(0, step - 1)

elif dice <= 5:

step = step + 1

else:

step = step + np.random.randint(1,7)

random\_walk.append(step)

# Append random\_walk to all\_walks

all\_walks.append(random\_walk)

# Print all\_walks

print(all\_walks)

# Visualize all walks

all\_walks is a list of lists: every sub-list represents a single random walk. If you convert this list of lists to a Numpy array, you can start making interesting plots! matplotlib.pyplot is already imported as plt.

The nested for loop is already coded for you - don't worry about it. For now, focus on the code that comes after this forloop.

##### INSTRUCTIONS

* Use **[np.array()](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.array.html" \t "_blank)** to convert all\_walks to a Numpy array, np\_aw.
* Try to use **[plt.plot()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.plot" \t "_blank)** on np\_aw. Also include **[plt.show()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.show" \t "_blank)**. Does it work out of the box?
* Transpose np\_aw by calling **[np.transpose()](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.transpose.html" \t "_blank)** on np\_aw. Call the result np\_aw\_t. Now every row in np\_all\_walksrepresents the position after 1 throw for the 10 random walks.
* Use **[plt.plot()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.plot" \t "_blank)** to plot np\_aw\_t; also include a **[plt.show()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.show" \t "_blank)**. Does it look better this time?

import matplotlib.pyplot as plt

import numpy as np

np.random.seed(123)

all\_walks = []

for i in range(10) :

random\_walk = [0]

for x in range(100) :

step = random\_walk[-1]

dice = np.random.randint(1,7)

if dice <= 2:

step = max(0, step - 1)

elif dice <= 5:

step = step + 1

else:

step = step + np.random.randint(1,7)

random\_walk.append(step)

all\_walks.append(random\_walk)

# Implement clumsiness

With this neatly written code of yours, changing the number of times the random walk should be simulated is super-easy. You simply update the [**range()**](https://docs.python.org/3/library/functions.html#func-range) function in the top-level for loop.

There's still something we forgot! You're a bit clumsy and you have a 0.1% chance of falling down. That calls for another random number generation. Basically, you can generate a random float between 0 and 1. If this value is less than or equal to 0.001, you should reset step to 0.

##### INSTRUCTIONS

* Change the [**range()**](https://docs.python.org/3/library/functions.html#func-range) function so that the simulation is performed 250 times.
* Finish the if condition so that step is set to 0 if a random float is less or equal to 0.001. Use **[np.random.rand()](http://docs.scipy.org/doc/numpy-1.10.1/reference/generated/numpy.random.rand.html" \t "_blank)**.

# Plot the distribution

All these fancy visualizations have put us on a sidetrack. We still have to solve the million-dollar problem: What are the odds that you'll reach 60 steps high on the Empire State Building?

Basically, you want to know about the end points of all the random walks you've simulated. These end points have a certain distribution that you can visualize with a histogram.

##### INSTRUCTIONS

* To make sure we've got enough simulations, go crazy. Simulate the random walk 500 times.
* From np\_aw\_t, select the last row. This contains the endpoint of all 500 random walks you've simulated. Store this Numpy array as ends.
* Use **[plt.hist()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.hist" \t "_blank)** to build a histogram of ends. Don't forget **[plt.show()](http://matplotlib.org/api/pyplot_api.html" \l "matplotlib.pyplot.show" \t "_blank)** to display the plot.s