**Random Numbers**

Well done and you're on the home stretch now! This chapter will be a bit different from the others. Step by step, we're going to work towards a cool script, using everything we've learned during this course.

**Image**

Imagine the following: you're walking up the empire state building to DataCamp HeadQuarters and you're playing a game with a friend.

You throw a die one hundred times.

If it's 1 or 2 you'll go one step down.

If it's 3, 4, or 5, you'll go one step up.

If you throw a 6, you'll throw the die again and will walk up the resulting number of steps.

Of course, you can not go lower than step number 0. And also, you admit that you're a bit clumsy and have a chance of 0.1% of falling down the stairs when you make a move. Falling down means that you have to start again from step 0. With all of this in mind, you bet with your friend that you'll reach 60 steps high.

**How to solve?**

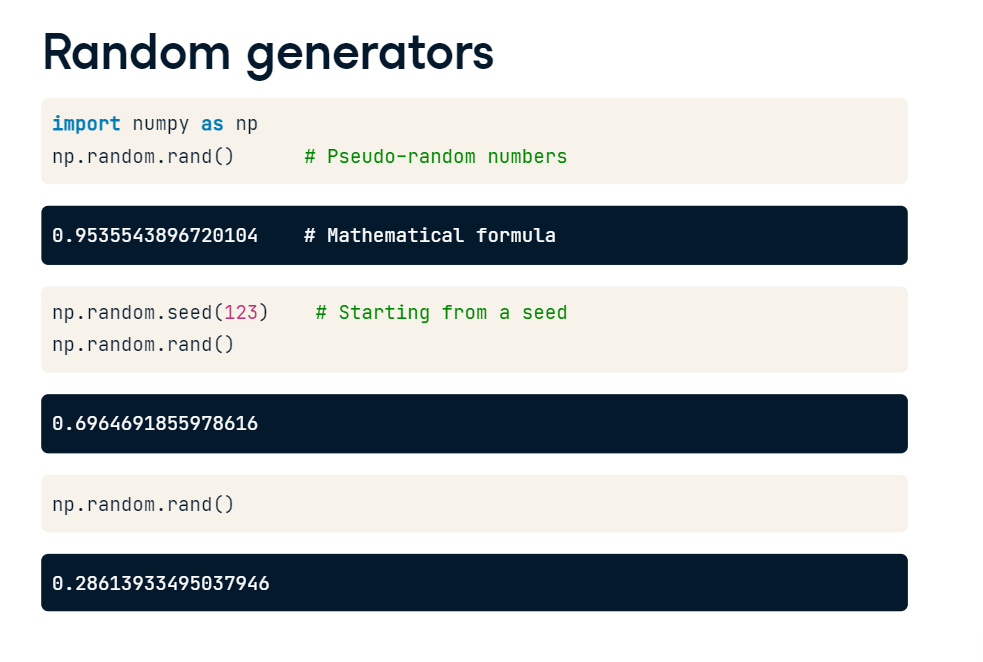
What is the chance that you will win this bet? It's a complex assignment. One way to solve it would be to calculate the chance analytically using equations. Another possible approach, is to simulate this process thousands of times, and see in what fraction of the simulations that you will reach 60 steps. This is a form of -hacker statistics-. As you can probably guess, we're going to opt for the second approach.

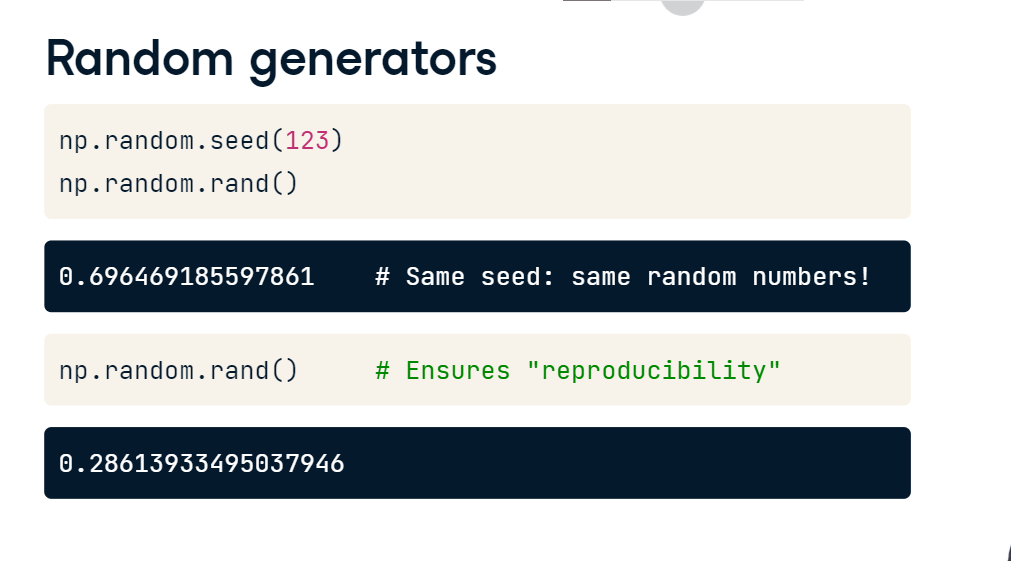
**Random generators**

The first thing we'll need are random generators, so we can simulate the die. You need to import numpy, and inside numpy, there is the random package. Inside that package we find the "rand" function. Let's try it out: we get a random number between zero and one. How was this random number created? Well, computers typically generate so-called pseudo-random numbers. Those are random numbers that are generated using a mathematical formula, starting from a random seed. This seed was chosen by Python when we called the rand function, but you can also set this manually. Suppose we set it to 123, just a number I chose, like this, and then call the rand function twice. We get two random numbers.

**Random generators**

Now, if I set the seed back to 123, and call rand twice more, we get the exact same random numbers. This is funky: you're generating random numbers, but for the same seed, you're generating the same random numbers. That's why it's called pseudo-random; it's random but consistent between runs; this is very useful, because this ensures "reproducibility". Other people can reproduce your analysis. Let's use this randomness in a new example now.



****

**Coin toss**

Suppose we want to simulate a coin toss. First set the seed - again, this could be anything - and then use the randint() function. To have it randomly generate either 0 or 1, we pass two arguments: the first argument should be 0, the second one 2, because 2 is not going to be included. If we print out coin, and then run the script, we get a random integer, 0. You can now use this coin to play a game.



We extend the code with an if-else statement: if coin equals 0, we print out "heads". If it equals 1, we print out "tails". If we now run this script again, coin will again equal 0, because the seed is the same. This also means that the if condition is True, so the string heads is printed out. This was a first example on how you can use random numbers to simulate real life situations that involve chance, or probability.



**Random Walk**

You are doing SO well.

**Random Step**

If you use a dice to determine your next step, you can call this a random step. What if you use a dice 100 times to determine your next step? You would have a succession of random steps, or in other words, a random walk.

**Random Walk**

This is a well known concept in science. For example, the path traced by a molecule as it travels in a liquid or a gas can be modeled as a random walk. The financial status of a gambler is another example. To record every step in your random walk, you need to learn how to gradually build a list with a for loop.

**Heads or Tails**

Have a look at this code. It keeps the outcomes for playing a game of heads or tails ten times, with the random number generator we coded up in the previous video. After importing numpy and setting a seed for the random number generator, we initialize an empty list "outcomes". Next, we build a for loop that should run ten times. We can do this with the range() function, that generates a list of numbers that you can use to iterate over. Inside this for loop, we generate a random integer coin that's either zero or one. Zero corresponds to heads, 1 to tails. If coin is zero, we append the string heads to the list. Else, we append the string tails. In both cases, we do this with the append method, which you learned about in the intro course. Finally, we print the outcomes list we've built up in these 10 iterations. If we run this script, eventually a list with 10 strings will be printed out. This list is random, but it's not a random walk, because the items in the list are not based on the previous ones. It's just a bunch of random steps.



**Heads or Tails: Random Walk**

You could turn this example into a random walk by tracking the -total- number of tails while you're simulating the game. In this case, you start by creating a list, tails, that already contains the number 0, because at the start, you haven't thrown any tails. Then you again start a for loop that runs 10 times, using the range function. In there, you again generate a random number. Instead of the if-else structure, you can simplify things. If coin is 0, so heads, the number of tails you've thrown shouldn't change. If a 1 is generated, the number of tails should be incremented with 1. This means that you can simply add coin to the previous number of tails, and add this count to the list with append. Finally, you again print the list tails. After running this script, a list with 11 elements will be printed out. The final element in this list tells you how often tails was thrown.



**Step to Walk**

If you compare the output of the first script to the output of the second script, you can see that the numbers in the tails list are incremented by one each time you threw tails. This is exactly how a bunch of random steps are converted into a random walk.



**Distribution**

Let's go back to the initial problem. you throw a die one hundred times. Depending on the result you go some steps up or some steps down. This is called a random walk, and you know how to simulate this. But you still have to answer the main question: what is the chance that you'll reach 60 steps high? I'll give you a hint. Each random walk will end up on a different step. If you simulate this walk thousands of times, you will end up with thousands of final steps. This is actually a distribution of final steps. And once you know the distribution, you can start calculating chances.

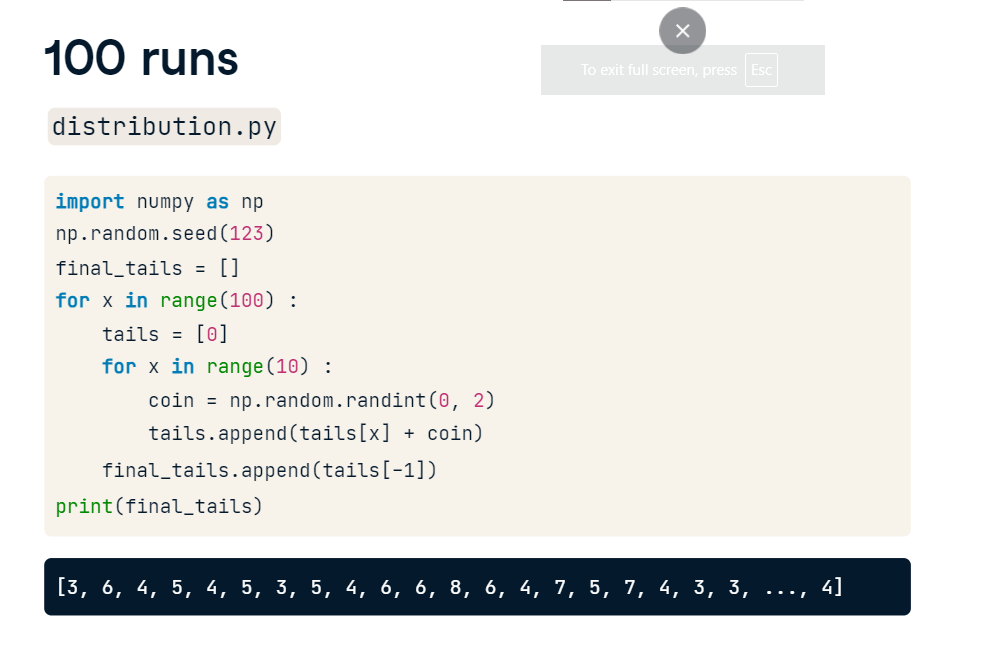
**Random Walk**

Let's go back to the example of the total number of tails after 10 coin tosses. The number of tails starts at zero and, ten times, we calculate a random number which is either 0 or 1. We then update the number of times tails has been thrown by appending it to the list.



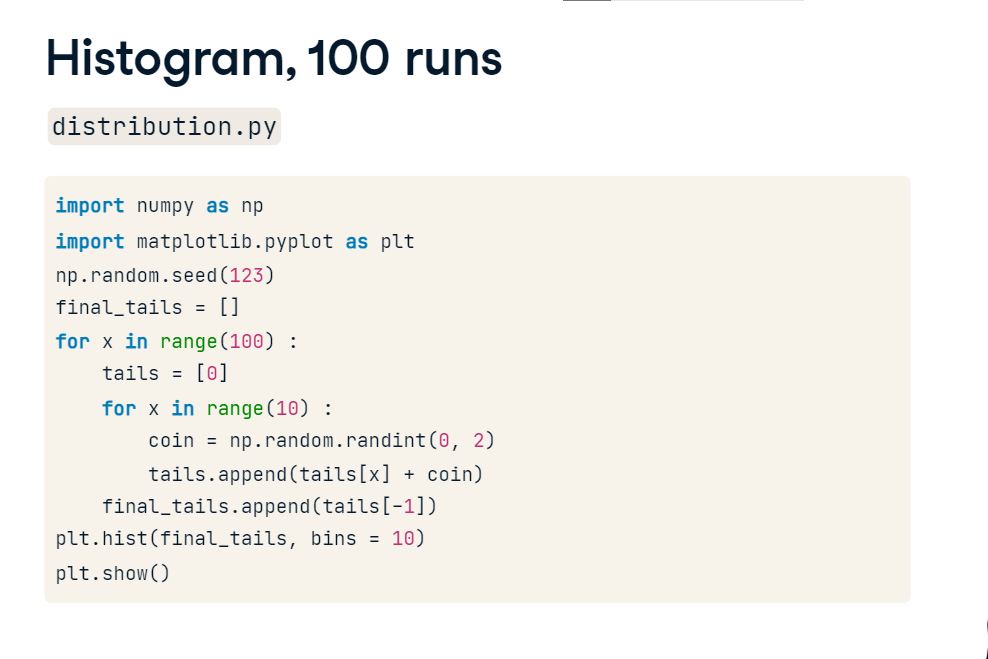
**100 runs**

To find the distribution of this walk, we start by setting a random seed, and then create an empty list named final\_tails. This list will contain the number of tails you end up with if you play this game of tossing a coin 10 times over and over again. Let's write a for loop that runs 100 times. Inside this for loop, we put the code from before, that gradually builds up the tails list. After simulating this single game, we append the last number, so the number of tails after tossing 10 times, to the final\_tails list. Notice that the indentation here specifies that this last line is part of the top-level for loop. If you put a last line in here to print final\_tails, outside of the for loops, and run the script, you see that final\_tails contains numbers between 0 and 10. Each number is the number of tails that were thrown in a game of 10 tosses. All these values actually represent a distribution, that we can visualize. Hmm, visualizing a distribution, that calls for a histogram!



**Histogram, 100 runs**

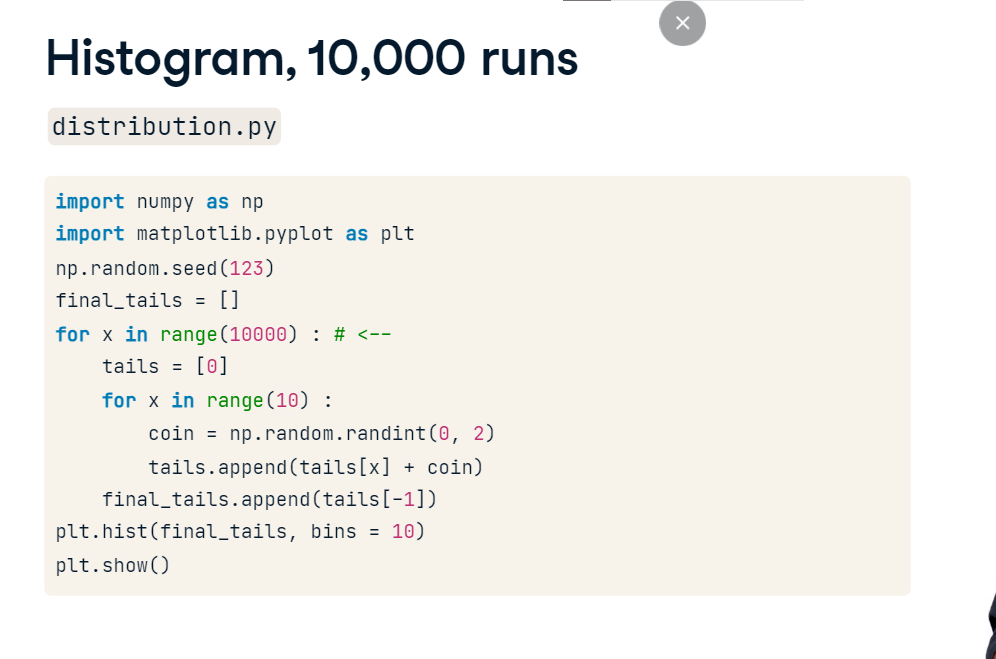
On the top of the script, we add a line to import pyplot, and then, instead of the print statement, we call the hist function, and specify that we want 10 bins. Of course, to actually display the plot, we need plt (dot) show().



If we run the script, the resulting histogram already gives an idea, but is not very smooth yet.

**Histogram, 1,000 runs**

Let's head back to the code, and now simulate the coin toss game one thousand times, by changing the range in the top-level for loop.

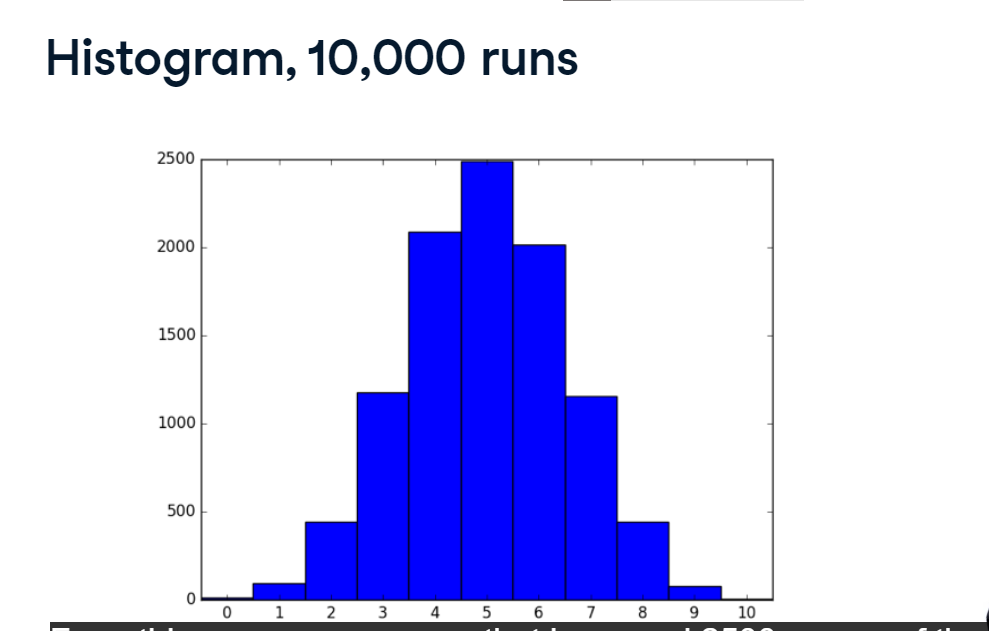


**Histogram, 1,000 runs**

This time, the histogram already looks better.

If we change the code to do ten thousand simulations,

and run the script once more, the distribution starts to converge to a bell-shape. In fact, it starts to look like the theoretical distribution. That means the distribution that you would find by doing analytical pen-and-paper calculations. Ideally, you want to carry out the experiment zillions of times to get a distribution that is exactly the same as the theoretical distribution. This will take too much computer time, though, but ten thousand already gives a pretty good estimate. From this curve, we can see that in around 2500 games of the 10000 games played, you end up with tails 5 times.



**Let's practice!**

In the last exercises of this chapter, you will use a similar technique to simulate the die rolling game in the Empire State Building over and over again. Go out there, and win this thing! And thank you so much for coming on this journey with me. I can't wait to see what you do next with all these skills.