Both baseline Python and **numpy** come with a random variable library. Some of the methods for the default Python **random** library can be seen here and, as usual, we can access this library with the usual syntax

```
import random
```

We will, however, focus on the **numpy** library due to its implicit vectorization benefits, i.e., its ability to generate arrays of random values. We can access this library with the syntax

```
from numpy import random
```

As the name of the library suggests, its main benefit is the ability to generate (pseudo-)random variables and arrays according to different distributions.

For example, if we want to draw a random number between 0 and 1 according to the uniform distribution (i.e., the likelihood of drawing $x \in [a, b]$ is equal to $b - a$) we can call

```
x = random.rand()
```

If we want to draw a 2×3 matrix where each entry in independently normally distributed (i.e., each entry comes from the standard Gaussian distribution), we can call

```
x = random.randn(2, 3)
```

If we want to simulate the outcome of rolling a (fair) dice 10000 times, we can call

```
x = random.randint(1, 7, size=(10000))
```

This will produce an array with 10000 entries, each of which being an independently drawn integer between 1 (inclusive) and 7 (exclusive, just like usual Python slicing syntax for indexing). The underlying distribution of 'randint' is the discrete uniform distribution (in this example, each of the outcomes $1, 2, 3, 4, 5, 6$ has a likelihood of occurring of $\frac{1}{6}$).

It should be pointed out that using simple arithmetic, we can extend the use of these constructor methods to most practical needs. For instance, calling

```
3+7*random.rand()
```

will generate a random number between 3 and 10 according to the uniform distribution on the interval $[3, 10]$.

An alternative to using 'randint' for simulating the outcome of flipping a fair coin would be the code

```
x = random.rand()
if x < 1/2:
        print(0)
else:
        print(1)
```

Note that we don't really need to worry about including or excluding the edge case $\frac{1}{2}$ from the 'else' branch because the event $x = \frac{1}{2}$ has probability 0.

The most common use of random variables in numerics is to perform Monte Carlo experiments in order to simulate the average outcome of a model. This is particularly useful (and sometimes the only feasible way to tackle a problem from a numerics point of view) when we are confronted with a very complicated model (or indeed a model that admits no known analytic formulation, e.g. finding solutions of some differential equations) and we need to understand the average or expected behavior of the model. We will now look at two toy-models where Monte Carlo experimentation is useful.

Approximating π

There are numerous very sophisticated numerical algorithms for obtaining approximations of π to various precision levels. In an analytic sense, π can of course be exactly obtained in various ways, for instance by evaluating $\cos^{-1}(-1)$ or by computing the area of the unit disk using integration. However, π is irrational (in fact transcendental), so it is not possible to have an exact value of π on any computer, no matter how sophisticated of an algorithm we use to try to compute it.

The following is arguably the most elementary (but not very precise) way of approximating π numerically. It does not rely on any trigonometry or analysis, just on the knowledge that the area of the unit disk should be π and the ability to draw random values from a uniform distribution.

Consider the unit square in the plane. If we also construct the unit disk in the plane, we see that exactly a quarter of the disc is contained in the unit square, namely the first quadrant. Therefore the area of the unit disk contained in the unit square is precisely $\frac{\pi}{4}$. The idea now is to approximate this value by uniformly drawing a large number of points from the unit square and counting how many of these points lie in the first quadrant of the disk. Since we draw the points uniformly, we expect the ratio of our counter to the total number of points drawn to approximate the ratio of the area of the first quadrant to the area of the unit square. In other words, we are approximating $\frac{\pi}{4}$.

A simple implementation of this would be something like

```
approx_pi_quarter = 0
for k in range(100_000):
    xrand = random.rand()
    yrand = random.rand()
    if xrand**2+yrand**2 <= 1:
        approx_pi_quarter += 1
print(4*approx_pi_quarter/100_000)
```

One dimensional random walk

Consider the following game. You start with the number 0 and a coin. Each turn, you flip the coin. If the outcome is heads, you add 1 to your number. If the outcome is tails, you subtract 1. Let's say you play the game for 10000 turns. What number will you end up with? What has the highest number you ever reached during the 10000 turns? What was the 'trajectory' of the value of your number during the 10000 turns? Mathematically, this game translates to a *random walk*.

Some questions about it are relatively easy to answer. For instance, if the coin you are using is a fair coin, then it is not difficult to see that after 10000 steps, you would expect to end up with the number 0. Also, since you are playing the game for 10000 steps, you can immediately figure out that you will never end up with an odd number (indeed, if you rolled exactly k heads, you would end up with the number $2k - 10000$ which is always even). However, other questions are much more difficult to settle, like the explicit trajectory your number will describe during the 10000 steps.

The following code is a simple implementation of plotting this trajectory as a function of the step counter.

```
import numpy as np
from numpy import random
import matplotlib.pyplot as plt
x = np.arange(10_000)
y = np.zeros(10_000)
for k in range(1, 10_000):
    random_direction = random.rand()-1/2
    y[k] = y[k-1]+np.sign(random_direction)
plt.plot(x, y)
plt.show()
```

Note that we use the ‘rand’ method and check if the draw is above or below $\frac{1}{2}$ and we use the sign of the difference to update the number. As discussed before, we could have also used ‘randint’. Rerunning the code will produce vastly different, chaotic looking graphs.