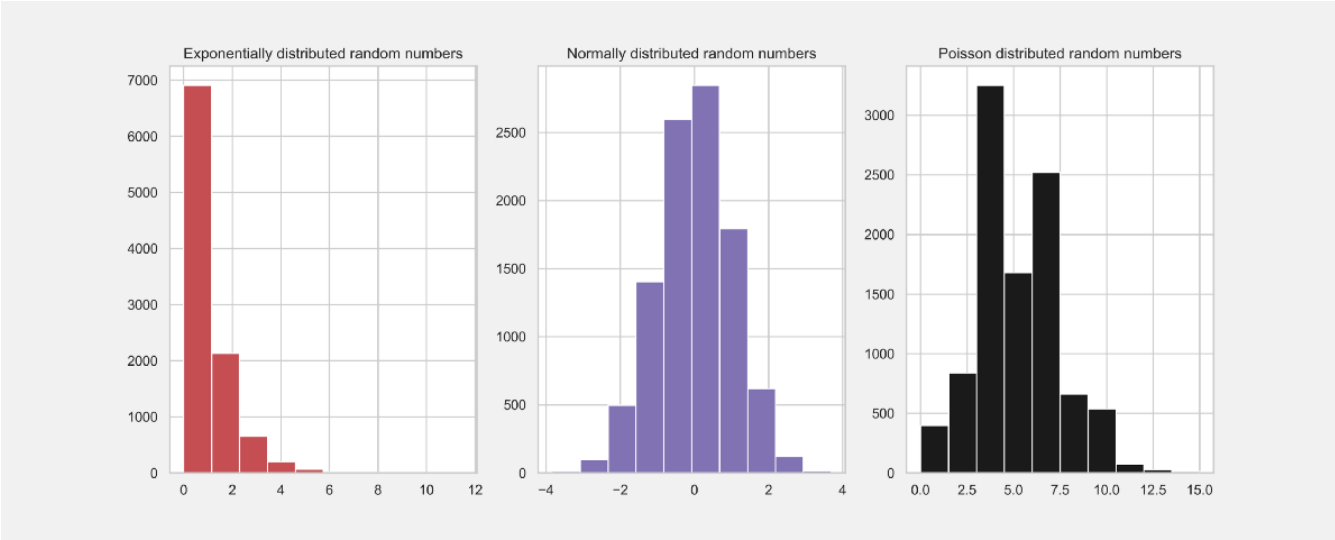
Most Important Random Number Python Modules To Keep Always By Your Side



Histogram plot of exponential, normal, and Poisson distribution functions of randomly generated numbers, created with the Python code in section III. In all plots, a sample of size of 5000 randomly generated numbers has been used. Figure created by the author for educational purposes.

Introduction

Very often in data science, statistical modelling, machine learning, and plotting, one needs to generate random data for simulation because of several reasons. There are many programming languages that one can use to generate random numbers and data, and in this article, I want to focus in particular on the Python programming language. In this article, I will show you which are the most important Python libraries and modules that can help the user to generate random data.

The reason behind this article is because I have observed that in most teaching materials, books on Python, and data science, they mention random number generating functions only thoroughly without dedicating a specific section to them. Clearly, in this article I will select the most important of these libraries and modules which I think are the

most used in data science in general, so, my selection is not complete because I will not discuss the full range of the existing random number modules. As usual, in this article, I assume that the reader knows basic Python and some of its libraries.

I. Not secure random number modules

Random number generation in Python and many other programs are not in reality completely "random" but have a deterministic component. For this reason, these numbers that are generated are called **pseudo-random numbers**. In what follows, when I write a "random number", you should understand that I mean a pseudo-random number. The reason why Python and other programs generate pseudo-random numbers is that one needs to start the sequence of random number generation by using an initial value which is usually called the *seed-number*.

Basic Python distributions, all have a default module called **random** responsible for the generation of random numbers. This is the core module to be called for the generation of different random numbers with different distribution functions. However, this module does not generate secure random numbers to be used for cryptography, that's why it is called a not secure random number module. Cryptographically generated random numbers are usually used for the generation of security tokens, passwords, etc.

1. random.seed(a=None, version=2)

This sub-module is used to fix and initialize the random number generator. The seed number in Python is used to reproduce the same

sequence of random numbers as far as multiple threads are not running. For example, if I set the seed value to a specific number, string, etc. another user by using the same seed number as mine would get the same sequence of random numbers if called in the same order with the same calling functions.

The first argument, a, of *random.seed()* module is the value of the seed and the second argument is the Python version to be used, where version=2 is the default version. For Python versions above 3, the seed value, a, can be int, float, str, bytes, or bytearray. For example, in this article I will use a seed value of 1, namely a=1, to generate random numbers. To start, I use the following Python code:

```
[In]: import random
[In]: random.seed(a = 1)
```

In the code above, the seed is started at the value of a=1, but one can use any value wishes. If you set the same seed value in your computer, you should generate the same sequence of numbers as mine in what follows.

2. random.randint(a, b)

This sub-module generates random integer numbers in the interval [a, b] with interval extremes included. If I choose, a = 0 and b = 10, I get for example:

```
[In]: random.randint(a = 0, b = 10)
[Out]: 2
```

3. random.randrange(*start*, *stop=None*, *step=1*)

This sub-module generates integer numbers in the half-open interval [a, b) with a default step of 1. This function is the same as

random.randint(), if instead of b we have b+1, or equivalently, *random.randint(a, b)* and *random.randrange(a, b+1),* will give the same random number if they are called immediately after *random.seed(a =1)* separately. For example, if I run the code below following the previous code in section 2, I get:

```
[In]: random.randrange(start = 0, stop = 11)
[Out]: 9
```

As you can see the result is different from the previous code because they have been executed in a sequence. However, if you execute these codes independently, you get the same results.

4. random.choice(seq)

This sub-module gives the user the possibility to choose a random number from a given sequence list. For example, suppose I have a list, l = [1, 4., 8, 9.6, 30], and then run the following code:

```
[In]: l = [1, 4., 8, 9.6, 30]
[In]: random.choice(list)
[Out]: 1
```

In the above example, I used a list of numbers, but the sequence can be a string, a tuple, a range, etc.

5. random.random()

This sub-module generates a random number(floating point) in the half-open interval [0, 1). If I run the following code, in sequence after the previous ones, I get:

```
[In]: random.random()
[Out]: 0.2550690257394217
```

6. random.gauss(mu, sigma)

This sub-module gives the user the possibility to generate a random number from a Gaussian distribution of random numbers with mean = mu and standard deviation = sigma. If I choose, for example, the mean $\mu = 0$, and $\sigma = 1$, I get:

```
[In]: random.gauss(mu = 0, sigma = 1)
[Out]: -1.0921732151041414
```

7. random.expovariate(lambda)

This sub-module of generates random numbers from the exponential probability distribution function, which expression is given by

$$f(x;\lambda) = \begin{cases} \lambda e^{-\lambda x} & x \geq 0, \\ 0 & x < 0. \end{cases} \qquad (1)$$

The exponential probability distribution function is important in different fields of science such as Physics, Biology, Hydrology, etc. For example, in Physics, radioactive particle decay distribution approximately follows function (1). The parameter λ is called the rate parameter, and it is a positive number bigger than zero. For a given random variable X, its expectation values is $E(X) = 1/\lambda$ and its variance is $\mathrm{Var}(X) = 1/\lambda^2$. So the *mean* of a random variable X following the exponential probability distribution function is $\mu = 1/\lambda$, where λ must be different from zero.

To generate, for example, a random number following distribution (1) with $\mu = 2$, I run the following Python code:

```
[Int]: random.expovariate(lambda = 1/2)
[Out]: 2.108767728131761
```

8. random.uniform(a, b)

This sub-module generates uniformly distributed floating-point numbers in the interval [a, b] with the extreme a, included but the extreme b may or may not be included depending on the rounding. For example, to generate a random number from a uniform distribution in the interval [0, 10], I run the following code, after the code in point 7 above(please pay attention to the sequence of codes to generate the same outputs as those presented here):

```
[In]: random.uniform(a = 0, b = 10)
[Out]: 7.887233511355132
```

The eight random number modules that I described above have a core module, the **random** module of Python. However, there are libraries in Python that are essentially based on the random module discussed above , that allow the user to perform fast and efficient numerical calculations. Among these libraries is the **NumPy** library. NumPy has many random number modules that can be used to generate numbers and arrays. To simplify the notation, first I import NumPy:

```
[In]: import numpy as np
[In]: np.random.seed(a = 2)
```

After importing NumPy, I also imported the random.seed() module to set the seed of the NumPy generator. This module is the NumPy equivalent of random.seed() in point 1. For simplicity, I choose the value of the seed equal to 2. Below the most important random number sub-modules of NumPy are discussed.

9. np.random.rand(d0, d1,..., dn)

This submodule of NumPy generates random numbers and arrays of different shapes filled with *uniformly distributed* random numbers in the half-open interval [0, 1). If not given an argument to this sub-module, it returns a random number in the interval [0, 1). If given one argument, a one-dimensional NumPy array is returned. If given more than one argument, a multi-dimensional arrays is returned . The following Python code makes these concepts more clear:

```
[In]: np.random.rand()
[Out]: 0.43599490214200376

[In]: np.random.rand(d0 = 3)
[Out]: array([0.02592623, 0.54966248, 0.43532239])

[In]: np.random(d0 = 3, d1 = 2)
[Out]: array([[0.4203678 , 0.33033482],
       [0.20464863, 0.61927097],
       [0.29965467, 0.26682728]])
```

The **np.random.rand()** module has similarities with the **np.random.random()** module which I do not cover in this article.

10. np.random.randn(d0, d1,..., dn)

This submodule of NumPy gives the user the possibility to generate a sample or samples of random data from the normal distribution with mean zero and standard deviation equal to one. The tuple (d0, d1,..., dn) must be of positive integer numbers and its presence is optional. The size of the tuple represents the dimensions of the returned array. If no number is given to the function **np.random.randn(),** its output is just a random floating-point number from the normal distribution. If only a number is given as an argument, the output is a float one-dimensional NumPy array with components equal to the given argument value. If

more than one number is given as arguments, the function returns a floating-point multidimensional array with specific dimensions and components. The following example makes things more clear:

```
[Int]: np.random.randn()
[Out]: 0.5514540445464243

[In]: np.random.randn(d0 = 4)
[Out]: array([ 2.29220801,  0.04153939, -1.11792545,  0.53905832])

[In]: np.random.randn(d0 = 3, d1 = 3)
[Out]: array([[-0.5961597 , -0.0191305 ,  1.17500122],
       [-0.74787095,  0.00902525, -0.87810789],
       [-0.15643417,  0.25657045, -0.98877905]])
```

The NumPy module **np.random.standard_normal()** is very similar to **np.random.rand(),** with the only difference that the former takes an integer or a tuple of integers as an argument.

11. np.random.randint(low, high = None, size = None, dtype = 'I')

This sub-module of NumPy generates random integers from the discrete uniform probability distribution function in the half-open interval [low, high). In the case when high = None, this sub-module generates integers in the interval [0, low). The size is in general either an integer or a tuple of integers and it specifies the size of the output *ndarray* or integer. The following example helps understand better this sub-module:

```
[In]: np.random.randint(low = 3, high = None, size = 4)
[Out]: array([2, 0, 2, 1])

[In]: np.random.randint(low = 3, high = 10, size = 4)
[Out]: array([9, 3, 5, 8])

[In]: np.random.randint(low = 0, high = 10, size = (2, 2))
[Out]: array([[9, 8],
       [7, 1]])
```

12. np.random.exponential(scale = 1, size = None)

This sub-module of NumPy generates samples from the exponential probability distribution function (1). The scale parameter is equal to *1/λ*, and its default value is equal to 1. The size specifies the size of the generated sample of numbers from the distribution (1). Here there are some specific examples:

```
[In]: np.random.exponential(scale = 2, size = 1)
[Out]: array([1.89881464])

[In]: np.random.exponential(scale = 2)
[Out]: 9.388438956222803

[In]: np.random.exponential(scale = 3, size = (2, 2))
[Out]: array([[0.38174426, 0.48249559],
       [5.73733106, 2.13714827]])

[In]: np.random.exponential(scale = 3, size = (1, 2, 2))
[Out]: array([[[ 0.73490213, 15.00188906],
        [ 1.13631244,  0.89818388]]])
```

13. np.random.binomial(n, p, size = None)

This sub-module of NumPy generates scalar numbers and arrays from the Binomial distribution function. Here I assume that the reader knows this type of distribution function that is very important in statistics. The argument n is the trial number and p is the probability of success in each trial. The probability p is in the interval [0, 1] and the trial number n is a positive number bigger or equal zero. The size specifies the dimensions of the output array. The following examples help to understand this sub-module:

```
[In]: np.random.binomial(n = 1, p = 1/2, size = None)
[Out]: 1

[In]: np.random.binomial(n = 10, p = 1/3, size = (2, 2, 2))
[Out]: array([[[3, 3],
        [2, 5]],

       [[5, 6],
```

```
      [5, 3]]])
```

```
[In]: np.random.binomial(n = 0, p = 0.4, size = 3)
[Out]: array([0, 0, 0])

[In]: np.random.binomial(n = (1, 2, 4), p = (0.1, 0.5, 0.4), size = 3)
[Out]: array([0, 2, 2])
```

14. np.random.poisson(lam = 1.0, size = None)

This sub-module of NumPy generates scalar numbers and arrays drawn from the Poisson probability distribution function. Even here, I assume that the reader knows this distribution, its parameters, and its properties. The first argument is interval λ where k events are supposed to occur. The parameter λ can be either a scalar or an array-like of scalars. The size can be None or a tuple that specifies the shape of the output multidimensional array. These examples help understand this sub-module:

```
[In]: np.random.poisson(lam = 2, size = None)
[Out]: 4

[In]: np.random.poisson(lam = 2, size = 4)
[Out]: array([4, 3, 1, 2])

[In]: np.random.poisson(lam = (1, 3, 4), size = (3, 3))
[Out]: array([[0, 6, 7],
      [3, 5, 9],
      [3, 2, 4]])
```

II. Secure random number module

All modules discussed in section II are not secure modules to generate random numbers and arrays of random numbers. Python gives the user the possibility to generate secure random numbers and bytes through the **secrets** Python module. This module creates random numbers and bytes for cryptography, passwords, tokens, etc. In this article I will not discuss about this module and the reader can find more information in

the link above.

III. Plotting random data

The random modules that have been described above are very useful for creating plots of simulations in many situations. They can be useful to check the behaviour of particular probability distribution functions, can be used to compare real data distribution with well know probability distributions like the Poisson, Normal, Bernoulli distributions, etc. In the Python code below I use some of the random number modules explained above to create the figure shown at the top of this article. The Python code is:

```
[In]: import numpy as np
[In]: import matplotlib.pyplot as plt
[In]: import seaborn as sns
[In]: sns.set(style="whitegrid")
[In]: np.random.seed(10)

[In]: fig, axs=plt.subplots(1, 3, figsize=(15, 6))
axs[0].hist(np.random.exponential(scale = 1, size = 10000), color = "r")
axs[0].set_title("Exponentially distributed random numbers")
axs[1].hist(np.random.randn(10000), color = "m")
axs[1].set_title("Normally distributed random numbers")
axs[2].hist(np.random.poisson(lam = 5, size = 10000), color = "k")
axs[2].set_title("Poisson distributed random numbers")
```

IV. Conclusion

In this article, I discussed the most important random number modules that exist in Python. As I mentioned at the beginning, my list is not complete because there are other modules as well, but I tried to be as much as possible unbiased on my list, which is composed of those modules that are most frequently used in data science in general.

If one desires just simple random numbers to use for general purposes,

then the in-build Python module **random** is more than enough. If one desires to create random numbers and arrays filled with random numbers, then the NumPy **random** module is probably the best to use. One can use this module for simulations and create plots of simulations as I showed in section III.

If you liked my article, please share it with your friends that might be interested in this topic and cite/refer to my article in your research studies. Do not forget to subscribe for other related topics that will post in the future.

By Damian Ejlli on September 1, 2021.

Canonical link

Exported from Medium on October 8, 2021.