Generating Random Data in Python (Guide)

by Brad Solomon Jul 11, 2018 7 Comments data-science intermediate python

Mark as Completed

Share Share Email

Table of Contents

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the w
tutorial to deepen your understanding: **Generating Random Data in Python**

Help

How random is random? This is a weird question to ask, but it is one of paramount importance in cases where information security is concerned. Whenever you're generating random data, strings, or numbers in Python, it's a good idea to have at least a rough idea of how that data was generated.

Here, you'll cover a handful of different options for generating random data in Python, and then build up to a comparison of each in terms of its level of security, versatility, purpose, and speed.

I promise that this tutorial will not be a lesson in mathematics or cryptography, which I wouldn't be well equipped to lecture on in the first place. You'll get into just as much math as needed, and no more.

How Random Is Random?

First, a prominent disclaimer is necessary. Most random data generated with Python is not fully random in the scientific sense of the word. Rather, it is **pseudorandom**: generated with a pseudorandom number generator (PRNG), which is essentially any algorithm for generating seemingly random but still reproducible data.

"True" random numbers can be generated by, you guessed it, a true random number generator (TRNG). One example is to repeatedly pick up a die off the floor, toss it in the air, and let it land how it may.

Assuming that your toss is unbiased, you have truly no idea what number the die will land on. Rolling a die is a crude form of using hardware to generate a number that is not deterministic whatsoever. (Or, you can have the dice-o-matic do this for you.) TRNGs are out of the scope of this article but worth a mention nonetheless for comparison's sake.

PRNGs, usually done with software rather than hardware, work slightly differently. Here's a concise description:

> They start with a random number, known as the seed, and then use an algorithm to generate a pseudo-random sequence of bits based on it. (Source)

You've likely been told to "read the docs!" at some point. Well, those people are not wrong. Here's a particularly notable snippet from the `random` module's documentation that you don't want to miss:

> **Warning**: The pseudo-random generators of this module should not be used for security purposes. (Source)

You've probably seen `random.seed(999)`, `random.seed(1234)`, or the like, in Python. This function call is seeding the underlying random number generator used by Python's `random` module. It is what makes subsequent calls to generate random numbers deterministic: input A always produces output B. This blessing can also be a curse if it is used maliciously.

Perhaps the terms "random" and "deterministic" seem like they cannot exist next to each other. To make that clearer, here's an extremely trimmed down version of `random()` that iteratively creates a "random" number by using `x = (x * 3) % 19`. `x` is originally defined as a seed value and then morphs into a deterministic sequence of numbers based on that seed:

Python

```python
class NotSoRandom(object):
    def seed(self, a=3):
        """Seed the world's most mysterious random number generator."""
        self.seedval = a
    def random(self):
        """Look, random numbers!"""
        self.seedval = (self.seedval * 3) % 19
        return self.seedval

_inst = NotSoRandom()
seed = _inst.seed
random = _inst.random
```

Don't take this example too literally, as it's meant mainly to illustrate the concept. If you use the seed value 1234, the subsequent sequence of calls to `random()` should always be identical:

Python

```
>>> seed(1234)
>>> [random() for _ in range(10)]
[16, 10, 11, 14, 4, 12, 17, 13, 1, 3]

>>> seed(1234)
>>> [random() for _ in range(10)]
[16, 10, 11, 14, 4, 12, 17, 13, 1, 3]
```

You'll see a more serious illustration of this shortly.

What Is "Cryptographically Secure?"

If you haven't had enough with the "RNG" acronyms, let's throw one more into the mix: a CSPRNG, or cryptographically secure PRNG. CSPRNGs are suitable for generating sensitive data such as passwords, authenticators, and tokens. Given a random string, there is realistically no way for Malicious Joe to determine what string came before or after that string in a sequence of random strings.

One other term that you may see is **entropy**. In a nutshell, this refers to the amount of randomness introduced or desired. For example, one Python module that you'll cover here defines `DEFAULT_ENTROPY = 32`, the number of bytes to return by default. The developers deem this to be "enough" bytes to be a sufficient amount of noise.

Note: Through this tutorial, I assume that a byte refers to 8 bits, as it has since the 1960s, rather than some other unit of data storage. You are free to call this an *octet* if you so prefer.

A key point about CSPRNGs is that they are still pseudorandom. They are engineered in some way that is internally deterministic, but they add some other variable or have some property that makes them "random enough" to prohibit backing into whatever function enforces determinism.

What You'll Cover Here

In practical terms, this means that you should use plain PRNGs for statistical modeling, simulation, and to make random data reproducible. They're also significantly faster than CSPRNGs, as you'll see later on. Use CSPRNGs for security and cryptographic applications where data sensitivity is imperative.

In addition to expanding on the use cases above, in this tutorial, you'll delve into Python tools for using both PRNGs and CSPRNGs:

- PRNG options include the `random` module from Python's standard library and its array-based NumPy counterpart, `numpy.random`.
- Python's `os`, `secrets`, and `uuid` modules contain functions for generating cryptographically secure objects.

You'll touch on all of the above and wrap up with a high-level comparison.

PRNGs in Python

The `random` Module

Probably the most widely known tool for generating random data in Python is its `random` module, which uses the Mersenne Twister PRNG algorithm as its core generator.

Earlier, you touched briefly on `random.seed()`, and now is a good time to see how it works. First, let's build some random data without seeding. The `random.random()` function returns a random float in the interval [0.0, 1.0). The result will always be less than the right-hand endpoint (1.0). This is also known as a semi-open range:

Python
```
>>> # Don't call `random.seed()` yet
>>> import random
>>> random.random()
0.35553263284394376
>>> random.random()
0.6101992345575074
```

If you run this code yourself, I'll bet my life savings that the numbers returned on your machine will be different. The default when you don't seed the generator is to use your current system time or a "randomness source" from your OS if one is available.

With `random.seed()`, you can make results reproducible, and the chain of calls after `random.seed()` will produce the same trail of data:

Python
```
>>> random.seed(444)
>>> random.random()
0.3088946587429545
>>> random.random()
0.01323751590501987

>>> random.seed(444)  # Re-seed
>>> random.random()
0.3088946587429545
>>> random.random()
0.01323751590501987
```

Notice the repetition of "random" numbers. The sequence of random numbers becomes deterministic, or completely determined by the seed value, 444.

Let's take a look at some more basic functionality of `random`. Above, you generated a random float. You can generate a random integer between two endpoints in Python with the `random.randint()` function. This spans the full [x, y] interval and may include both endpoints:

Python
```
>>> random.randint(0, 10)
7
>>> random.randint(500, 50000)
18601
```

With `random.randrange()`, you can exclude the right-hand side of the interval, meaning the generated number always lies within [x, y) and will always be smaller than the right endpoint:

Python
```
>>> random.randrange(1, 10)
5
```

If you need to generate random floats that lie within a specific [x, y] interval, you can use `random.uniform()`, which plucks from the continuous uniform distribution:

Python
```
>>> random.uniform(20, 30)
27.42639687016509
>>> random.uniform(30, 40)
36.33865802745107
```

To pick a random element from a non-empty sequence (like a list or a tuple), you can use `random.choice()`. There is also `random.choices()` for choosing multiple elements from a sequence with replacement (duplicates are possible):

Python

```
>>> items = ['one', 'two', 'three', 'four', 'five']
>>> random.choice(items)
'four'

>>> random.choices(items, k=2)
['three', 'three']
>>> random.choices(items, k=3)
['three', 'five', 'four']
```

To mimic sampling without replacement, use `random.sample()`:

Python

```
>>> random.sample(items, 4)
['one', 'five', 'four', 'three']
```

You can randomize a sequence in-place using `random.shuffle()`. This will modify the sequence object and randomize the order of elements:

Python

```
>>> random.shuffle(items)
>>> items
['four', 'three', 'two', 'one', 'five']
```

If you'd rather not mutate the original list, you'll need to make a copy first and then shuffle the copy. You can create copies of Python lists with the `copy` module, or just `x[:]` or `x.copy()`, where `x` is the list.

Before moving on to generating random data with NumPy, let's look at one more slightly involved application: generating a sequence of unique random strings of uniform length.

It can help to think about the design of the function first. You need to choose from a "pool" of characters such as letters, numbers, and/or punctuation, combine these into a single string, and then check that this string has not already been generated. A Python `set` works well for this type of membership testing:

Python

```
import string

def unique_strings(k: int, ntokens: int,
               pool: str=string.ascii_letters) -> set:
    """Generate a set of unique string tokens.

    k: Length of each token
    ntokens: Number of tokens
    pool: Iterable of characters to choose from

    For a highly optimized version:
    https://stackoverflow.com/a/48421303/7954504
    """

    seen = set()

    # An optimization for tightly-bound loops:
    # Bind these methods outside of a loop
    join = ''.join
    add = seen.add

    while len(seen) < ntokens:
        token = join(random.choices(pool, k=k))
        add(token)
    return seen
```

`''.join()` joins the letters from `random.choices()` into a single Python `str` of length `k`. This token is added to the set, which can't contain duplicates, and the `while` loop executes until the set has the number of elements that you specify.

Resource: Python's `string` module contains a number of useful constants: `ascii_lowercase`, `ascii_uppercase`, `string.punctuation`, `ascii_whitespace`, and a handful of others.

Let's try this function out:

Python

```
>>> unique_strings(k=4, ntokens=5)
{'AsMk', 'Cvmi', 'GIxv', 'HGsZ', 'eurU'}

>>> unique_strings(5, 4, string.printable)
{"'O*1!", '9Ien%', 'W=m7<', 'mUD|z'}
```

For a fine-tuned version of this function, this Stack Overflow answer uses generator functions, name binding, and some other advanced tricks to make a faster, cryptographically secure version of `unique_strings()` above.

A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You
pythonistacafe.com

Remove ads

PRNGs for Arrays: `numpy.random`

One thing you might have noticed is that a majority of the functions from `random` return a scalar value (a single `int`, `float`, or other object). If you wanted to generate a sequence of random numbers, one way to achieve that would be with a Python list comprehension:

Python

```
>>> [random.random() for _ in range(5)]
[0.021655420657909374,
 0.4031628347066195,
 0.6609991871223335,
 0.5854998250783767,
 0.42886606317322706]
```

But there is another option that is specifically designed for this. You can think of NumPy's own `numpy.random` package as being like the standard library's `random`, but for NumPy arrays. (It also comes loaded with the ability to draw from a lot more statistical distributions.)

Take note that `numpy.random` uses its own PRNG that is separate from plain old `random`. You won't produce deterministically random NumPy arrays with a call to Python's own `random.seed()`:

Python

```
>>> import numpy as np
>>> np.random.seed(444)
>>> np.set_printoptions(precision=2)  # Output decimal fmt.
```

Without further ado, here are a few examples to whet your appetite:

Python

```
>>> # Return samples from the standard normal distribution
>>> np.random.randn(5)
array([ 0.36,  0.38,  1.38,  1.18, -0.94])

>>> np.random.randn(3, 4)
array([[-1.14, -0.54, -0.55,  0.21],
       [ 0.21,  1.27, -0.81, -3.3 ],
       [-0.81, -0.36, -0.88,  0.15]])

>>> # `p` is the probability of choosing each element
>>> np.random.choice([0, 1], p=[0.6, 0.4], size=(5, 4))
array([[0, 0, 1, 0],
       [0, 1, 1, 1],
       [1, 1, 1, 0],
       [0, 0, 0, 1],
       [0, 1, 0, 1]])
```

In the syntax for `randn(d0, d1, ..., dn)`, the parameters `d0, d1, ..., dn` are optional and indicate the shape of the final object. Here, `np.random.randn(3, 4)` creates a 2d array with 3 rows and 4 columns. The data will be i.i.d., meaning that each data point is drawn independent of the others.

Note: If you're looking to create normally distributed random numbers, then you're in luck! How to Get Normally Distributed Random Numbers With NumPy can guide your way.

Another common operation is to create a sequence of random Boolean values, `True` or `False`. One way to do this would be with `np.random.choice([True, False])`. However, it's actually about 4x faster to choose from `(0, 1)` and then view-cast these integers to their corresponding Boolean values:

Python

```
>>> # NumPy's `randint` is [inclusive, exclusive), unlike `random.randint()`
>>> np.random.randint(0, 2, size=25, dtype=np.uint8).view(bool)
array([ True, False,  True,  True, False,  True, False, False, False,
       False, False,  True,  True, False, False, False,  True, False,
        True, False,  True,  True,  True, False,  True])
```

What about generating correlated data? Let's say you want to simulate two correlated time series. One way of going about this is with NumPy's `multivariate_normal()` function, which takes a covariance matrix into account. In other words, to draw from a single normally distributed random variable, you need to specify its mean and variance (or standard deviation).

To sample from the multivariate normal distribution, you specify the means and covariance matrix, and you end up with multiple, correlated series of data that are each approximately normally distributed.

However, rather than covariance, correlation is a measure that is more familiar and intuitive to most. It's the covariance normalized by the product of standard deviations, and so you can also define covariance in terms of correlation and standard deviation:

$$cov(X, Y) = corr(X, Y) * \sigma_X \sigma_Y$$

So, could you draw random samples from a multivariate normal distribution by specifying a correlation matrix and standard deviations? Yes, but you'll need to get the above into matrix form first. Here, *S* is a vector of the standard deviations, *P* is their correlation matrix, and *C* is the resulting (square) covariance matrix:

$$\boldsymbol{C = diag(S) \bullet P \bullet diag(S)}$$

This can be expressed in NumPy as follows:

Python

```python
def corr2cov(p: np.ndarray, s: np.ndarray) -> np.ndarray:
    """Covariance matrix from correlation & standard deviations"""
    d = np.diag(s)
    return d @ p @ d
```

Now, you can generate two time series that are correlated but still random:

Python

```python
>>> # Start with a correlation matrix and standard deviations.
>>> # -0.40 is the correlation between A and B, and the correlation
>>> # of a variable with itself is 1.0.
>>> corr = np.array([[1., -0.40],
...                  [-0.40, 1.]])

>>> # Standard deviations/means of A and B, respectively
>>> stdev = np.array([6., 1.])
>>> mean = np.array([2., 0.5])
>>> cov = corr2cov(corr, stdev)

>>> # `size` is the length of time series for 2d data
>>> # (500 months, days, and so on).
>>> data = np.random.multivariate_normal(mean=mean, cov=cov, size=500)
>>> data[:10]
array([[ 0.58,  1.87],
       [-7.31,  0.74],
       [-6.24,  0.33],
       [-0.77,  1.19],
       [ 1.71,  0.7 ],
       [-3.33,  1.57],
       [-1.13,  1.23],
       [-6.58,  1.81],
       [-0.82, -0.34],
       [-2.32,  1.1 ]])
>>> data.shape
(500, 2)
```

You can think of `data` as 500 pairs of inversely correlated data points. Here's a sanity check that you can back into the original inputs, which approximate `corr`, `stdev`, and `mean` from above:

Python

```python
>>> np.corrcoef(data, rowvar=False)
array([[ 1.  , -0.39],
       [-0.39,  1.  ]])

>>> data.std(axis=0)
array([5.96, 1.01])

>>> data.mean(axis=0)
array([2.13, 0.49])
```

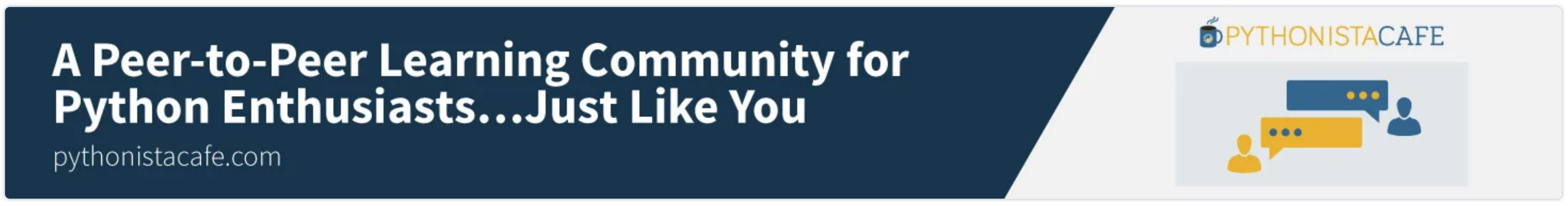
Before we move on to CSPRNGs, it might be helpful to summarize some `random` functions and their `numpy.random` counterparts:

Python `random` Module	NumPy Counterpart	Use
`random()`	`rand()`	Random float in [0.0, 1.0)
`randint(a, b)`	`random_integers()`	Random integer in [a, b]
`randrange(a, b[, step])`	`randint()`	Random integer in [a, b)
`uniform(a, b)`	`uniform()`	Random float in [a, b]
`choice(seq)`	`choice()`	Random element from `seq`

Python `random` Module	NumPy Counterpart	Use
`choices(seq, k=1)`	`choice()`	Random `k` elements from `seq` with replacement
`sample(population, k)`	`choice()` with `replace=False`	Random `k` elements from `seq` without replacement
`shuffle(x[, random])`	`shuffle()`	Shuffle the sequence `x` in place
`normalvariate(mu, sigma)` or `gauss(mu, sigma)`	`normal()`	Sample from a normal distribution with mean `mu` and standard deviation `sigma`

Note: NumPy is specialized for building and manipulating large, multidimensional arrays. If you just need a single value, `random` will suffice and will probably be faster as well. For small sequences, `random` may even be faster too, because NumPy does come with some overhead.

Now that you've covered two fundamental options for PRNGs, let's move onto a few more secure adaptations.


A Peer-to-Peer Learning Community for Python Enthusiasts...Just Like You
pythonistacafe.com
PYTHONISTACAFE

Remove ads

CSPRNGs in Python

`os.urandom()`: About as Random as It Gets

Python's `os.urandom()` function is used by both `secrets` and `uuid` (both of which you'll see here in a moment). Without getting into too much detail, `os.urandom()` generates operating-system-dependent random bytes that can safely be called cryptographically secure:

- On Unix operating systems, it reads random bytes from the special file `/dev/urandom`, which in turn "allow access to environmental noise collected from device drivers and other sources." (Thank you, Wikipedia.) This is garbled information that is particular to your hardware and system state at an instance in time but at the same time sufficiently random.
- On Windows, the C++ function `CryptGenRandom()` is used. This function is still technically pseudorandom, but it works by generating a seed value from variables such as the process ID, memory status, and so on.

With `os.urandom()`, there is no concept of manually seeding. While still technically pseudorandom, this function better aligns with how we think of randomness. The only argument is the number of bytes to return:

Python
```python
>>> os.urandom(3)
b'\xa2\xe8\x02'

>>> x = os.urandom(6)
>>> x
b'\xce\x11\xe7"!\x84'

>>> type(x), len(x)
(bytes, 6)
```

Before we go any further, this might be a good time to delve into a mini-lesson on character encoding. Many people, including myself, have some type of allergic reaction when they see `bytes` objects and a long line of `\x` characters. However, it's useful to know how sequences such as `x` above eventually get turned into strings or numbers.

`os.urandom()` returns a sequence of single bytes:

Python

```
>>> x
b'\xce\x11\xe7"!\x84'
```

But how does this eventually get turned into a Python `str` or sequence of numbers?

First, recall one of the fundamental concepts of computing, which is that a byte is made up of 8 bits. You can think of a bit as a single digit that is either 0 or 1. A byte effectively chooses between 0 and 1 eight times, so both `01101100` and `11110000` could represent bytes. Try this, which makes use of Python f-strings introduced in Python 3.6, in your interpreter:

Python

```
>>> binary = [f'{i:0>8b}' for i in range(256)]
>>> binary[:16]
['00000000',
 '00000001',
 '00000010',
 '00000011',
 '00000100',
 '00000101',
 '00000110',
 '00000111',
 '00001000',
 '00001001',
 '00001010',
 '00001011',
 '00001100',
 '00001101',
 '00001110',
 '00001111']
```

This is equivalent to `[bin(i) for i in range(256)]`, with some special formatting. `bin()` converts an integer to its binary representation as a string.

Where does that leave us? Using `range(256)` above is not a random choice. (No pun intended.) Given that we are allowed 8 bits, each with 2 choices, there are `2 ** 8 == 256` possible bytes "combinations."

This means that each byte maps to an integer between 0 and 255. In other words, we would need more than 8 bits to express the integer 256. You can verify this by checking that `len(f'{256:0>8b}')` is now 9, not 8.

Okay, now let's get back to the `bytes` data type that you saw above, by constructing a sequence of the bytes that correspond to integers 0 through 255:

Python

```
>>> bites = bytes(range(256))
```

If you call `list(bites)`, you'll get back to a Python list that runs from 0 to 255. But if you just print `bites`, you get an ugly looking sequence littered with backslashes:

Python

```
>>> bites
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f\x10\x11\x12\x13\x14\x15'
 '\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f !"#$%&\'()*+,-./0123456789:;<=>?@ABCDEFGHIJK'
 'LMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz{|}~\x7f\x80\x81\x82\x83\x84\x85\x86'
 '\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b'

 # ...
```

These backslashes are escape sequences, and `\xhh` represents the character with hex value `hh`. Some of the elements of `bites` are displayed literally (printable characters such as letters, numbers, and punctuation). Most are expressed with escapes. `\x08` represents a keyboard's backspace, while `\x13` is a carriage return (part of a new line, on Windows systems).

If you need a refresher on hexadecimal, Charles Petzold's *Code: The Hidden Language* is a great place for that. Hex is a base-16 numbering system that, instead of using 0 through 9, uses 0 through 9 and *a* through *f* as its basic digits.

Finally, let's get back to where you started, with the sequence of random bytes `x`. Hopefully this makes a little more sense now. Calling `.hex()` on a `bytes` object gives a `str` of hexadecimal numbers, with each corresponding to a decimal number from 0 through 255:

Python

```
>>> x
b'\xce\x11\xe7"!\x84'

>>> list(x)
[206, 17, 231, 34, 33, 132]

>>> x.hex()
'ce11e7222184'

>>> len(x.hex())
12
```

One last question: how is `b.hex()` 12 characters long above, even though `x` is only 6 bytes? This is because two hexadecimal digits correspond precisely to a single byte. The `str` version of `bytes` will always be twice as long as far as our eyes are concerned.

Even if the byte (such as `\x01`) does not need a full 8 bits to be represented, `b.hex()` will always use two hex digits per byte, so the number 1 will be represented as `01` rather than just `1`. Mathematically, though, both of these are the same size.

Technical Detail: What you've mainly dissected here is how a `bytes` object becomes a Python `str`. One other technicality is how `bytes` produced by `os.urandom()` get converted to a `float` in the interval [0.0, 1.0), as in the cryptographically secure version of `random.random()`. If you're interested in exploring this further, this code snippet demonstrates how `int.from_bytes()` makes the initial conversion to an integer, using a base-256 numbering system.

With that under your belt, let's touch on a recently introduced module, `secrets`, which makes generating secure tokens much more user-friendly.

Python's Best Kept `secrets`

Introduced in Python 3.6 by one of the more colorful PEPs out there, the `secrets` module is intended to be the de facto Python module for generating cryptographically secure random bytes and strings.

You can check out the source code for the module, which is short and sweet at about 25 lines of code. `secrets` is basically a wrapper around `os.urandom()`. It exports just a handful of functions for generating random numbers, bytes, and strings. Most of these examples should be fairly self-explanatory:

Python

```
>>> n = 16

>>> # Generate secure tokens
>>> secrets.token_bytes(n)
b'A\x8cz\xe1o\xf9!;\x8b\xf2\x80pJ\x8b\xd4\xd3'
>>> secrets.token_hex(n)
'9cb190491e01230ec4239cae643f286f'
>>> secrets.token_urlsafe(n)
'MJoi7CknFu3YN41m88SEgQ'

>>> # Secure version of `random.choice()`
>>> secrets.choice('rain')
'a'
```

Now, how about a concrete example? You've probably used URL shortener services like tinyurl.com or bit.ly that turn an unwieldy URL into something like https://bit.ly/2IcCp9u. Most shorteners don't do any complicated hashing from input to output; they just generate a random string, make sure that string has not already been generated previously, and then tie that back to the input URL.

Let's say that after taking a look at the Root Zone Database, you've registered the site **short.ly**. Here's a function to get you started with your service:

Python

```python
# shortly.py

from secrets import token_urlsafe

DATABASE = {}

def shorten(url: str, nbytes: int=5) -> str:
    ext = token_urlsafe(nbytes=nbytes)
    if ext in DATABASE:
        return shorten(url, nbytes=nbytes)
    else:
        DATABASE.update({ext: url})
        return f'short.ly/{ext}
```

Is this a full-fledged real illustration? No. I would wager that bit.ly does things in a slightly more advanced way than storing its gold mine in a global Python dictionary that is not persistent between sessions.

Note: If you'd like to build a full-fledged URL shortener of your own, then check out Build a URL Shortener With FastAPI and Python.

However, it's roughly accurate conceptually:

Python

```
>>> urls = (
...     'https://realpython.com/',
...     'https://docs.python.org/3/howto/regex.html'
... )

>>> for u in urls:
...     print(shorten(u))
short.ly/p_Z4fLI
short.ly/fuxSyNY

>>> DATABASE
{'p_Z4fLI': 'https://realpython.com/',
 'fuxSyNY': 'https://docs.python.org/3/howto/regex.html'}
```

Hold On: One thing you may notice is that both of these results are of length 7 when you requested 5 bytes. *Wait, I thought that you said the result would be twice as long?* Well, not exactly, in this case. There is one more thing going on here: `token_urlsafe()` uses base64 encoding, where each character is 6 bits of data. (It's 0 through 63, and corresponding characters. The characters are A-Z, a-z, 0-9, and +/.)

If you originally specify a certain number of bytes `nbytes`, the resulting length from `secrets.token_urlsafe(nbytes)` will be `math.ceil(nbytes * 8 / 6)`, which you can prove and investigate further if you're curious.

The bottom line here is that, while `secrets` is really just a wrapper around existing Python functions, it can be your go-to when security is your foremost concern.

One Last Candidate: `uuid`

One last option for generating a random token is the `uuid4()` function from Python's `uuid` module. A UUID is a Universally Unique IDentifier, a 128-bit sequence (`str` of length 32) designed to "guarantee uniqueness across space and time." `uuid4()` is one of the module's most useful functions, and this function also uses `os.urandom()`:

Python

```python
>>> import uuid

>>> uuid.uuid4()
UUID('3e3ef28d-3ff0-4933-9bba-e5ee91ce0e7b')
>>> uuid.uuid4()
UUID('2e115fcb-5761-4fa1-8287-19f4ee2877ac')
```

The nice thing is that all of `uuid`'s functions produce an instance of the `UUID` class, which encapsulates the ID and has properties like `.int`, `.bytes`, and `.hex`:

Python

```python
>>> tok = uuid.uuid4()
>>> tok.bytes
b'.\xb7\x80\xfd\xbfIG\xb3\xae\x1d\xe3\x97\xee\xc5\xd5\x81'

>>> len(tok.bytes)
16
>>> len(tok.bytes) * 8  # In bits
128

>>> tok.hex
'2eb780fdbf4947b3ae1de397eec5d581'
>>> tok.int
62097294383572614195530565389543396737
```

You may also have seen some other variations: `uuid1()`, `uuid3()`, and `uuid5()`. The key difference between these and `uuid4()` is that those three functions all take some form of input and therefore don't meet the definition of "random" to the extent that a Version 4 UUID does:

- `uuid1()` uses your machine's host ID and current time by default. Because of the reliance on current time down to nanosecond resolution, this version is where UUID derives the claim "guaranteed uniqueness across time."
- `uuid3()` and `uuid5()` both take a namespace identifier and a name. The former uses an MD5 hash and the latter uses SHA-1.

`uuid4()`, conversely, is entirely pseudorandom (or random). It consists of getting 16 bytes via `os.urandom()`, converting this to a big-endian integer, and doing a number of bitwise operations to comply with the formal specification.

Hopefully, by now you have a good idea of the distinction between different "types" of random data and how to create them. However, one other issue that might come to mind is that of collisions.

In this case, a collision would simply refer to generating two matching UUIDs. What is the chance of that? Well, it is technically not zero, but perhaps it is close enough: there are `2 ** 128` or 340 undecillion possible `uuid4` values. So, I'll leave it up to you to judge whether this is enough of a guarantee to sleep well.

One common use of `uuid` is in Django, which has a `UUIDField` that is often used as a primary key in a model's underlying relational database.

Why Not Just "Default to" `SystemRandom`?

In addition to the secure modules discussed here such as `secrets`, Python's `random` module actually has a little-used class called `SystemRandom` that uses `os.urandom()`. (`SystemRandom`, in turn, is also used by `secrets`. It's all a bit of a web that traces back to `urandom()`.)

At this point, you might be asking yourself why you wouldn't just "default to" this version? Why not "always be safe" rather than defaulting to the deterministic `random` functions that aren't cryptographically secure ?

I've already mentioned one reason: sometimes you want your data to be deterministic and reproducible for others to follow along with.

But the second reason is that CSPRNGs, at least in Python, tend to be meaningfully slower than PRNGs. Let's test that with a script, `timed.py`, that compares the PRNG and CSPRNG versions of `randint()` using Python's `timeit.repeat()`:

Python

```python
# timed.py

import random
import timeit

# The "default" random is actually an instance of `random.Random()`.
# The CSPRNG version uses `SystemRandom()` and `os.urandom()` in turn.
_sysrand = random.SystemRandom()

def prng() -> None:
    random.randint(0, 95)

def csprng() -> None:
    _sysrand.randint(0, 95)

setup = 'import random; from __main__ import prng, csprng'

if __name__ == '__main__':
    print('Best of 3 trials with 1,000,000 loops per trial:')

    for f in ('prng()', 'csprng()'):
        best = min(timeit.repeat(f, setup=setup))
        print('\t{:8s} {:0.2f} seconds total time.'.format(f, best))
```

Now to execute this from the shell:

Shell

```
$ python3 ./timed.py
Best of 3 trials with 1,000,000 loops per trial:
        prng()   1.07 seconds total time.
        csprng() 6.20 seconds total time.
```

A 5x timing difference is certainly a valid consideration in addition to cryptographic security when choosing between the two.

Odds and Ends: Hashing

One concept that hasn't received much attention in this tutorial is that of hashing, which can be done with Python's `hashlib` module.

A hash is designed to be a one-way mapping from an input value to a fixed-size string that is virtually impossible to reverse engineer. As such, while the result of a hash function may "look like" random data, it doesn't really qualify under the definition here.

Recap

You've covered a lot of ground in this tutorial. To recap, here is a high-level comparison of the options available to you for engineering randomness in Python:

Package/Module	Description	Cryptographically Secure
`random`	Fasty & easy random data using Mersenne Twister	No
`numpy.random`	Like `random` but for (possibly multidimensional) arrays	No
`os`	Contains `urandom()`, the base of other functions covered here	Yes
`secrets`	Designed to be Python's de facto module for generating secure random numbers, bytes, and strings	Yes
`uuid`	Home to a handful of functions for building 128-bit identifiers	Yes, `uuid4()`

Feel free to leave some totally random comments below, and thanks for reading.

Additional Links

- Random.org offers "true random numbers to anyone on the Internet" derived from atmospheric noise.
- The Recipes section from the `random` module has some additional tricks.
- The seminal paper on the Mersienne Twister appeared in 1997, if you're into that kind of thing.
- The Itertools Recipes define functions for choosing randomly from a combinatoric set, such as from combinations or permutations.
- Scikit-Learn includes various random sample generators that can be used to build artificial datasets of controlled size and complexity.
- Eli Bendersky digs into `random.randint()` in his article Slow and Fast Methods for Generating Random Integers in Python.
- Peter Norvig's a Concrete Introduction to Probability using Python is a comprehensive resource as well.
- The Pandas library includes a context manager that can be used to set a temporary random state.
- From Stack Overflow:
  - Generating Random Dates In a Given Range
  - Fastest Way to Generate a Random-like Unique String with Random Length
  - How to Use `random.shuffle()` on a Generator
  - Replace Random Elements in a NumPy Array
  - Getting Numbers from /dev/random in Python

Mark as Completed

Watch Now This tutorial has a related video course created by the Real Python team. Watch it together with the written tutorial to deepen your understanding: **Generating Random Data in Python**

🐍 Python Tricks 💌