Random module

All Versions

Examples

Creating a random user password

In order to create a random user password we can use the symbols provided in the `string` module. Specifically `punctuation` for punctuation symbols, `ascii_letters` for letters and `digits` for digits:

```
from string import punctuation, ascii_letters, digits
```

We can then combine all these symbols in a name named `symbols` :

```
symbols = ascii_letters + digits + punctuation
```

Remove either of these to create a pool of symbols with fewer elements.

After this, we can use `random.SystemRandom` to generate a password. For a 10 length password:

```
secure_random = random.SystemRandom()
password = "".join(secure_random.choice(symbols) for i in range(10))
print(password)  # '^@g;J?]M6e'
```

Note that other routines made immediately available by the `random` module — such as `random.choice` , `random.randint` , etc. — are *unsuitable* for cryptographic purposes.

Behind the curtains, these routines use the Mersenne Twister PRNG , which does not satisfy the requirements of a CSPRNG . Thus, in particular, you should not use any of them to generate passwords you plan to use. Always use an instance of `SystemRandom` as shown above.

Python 3.x ≥3.6

Starting from Python 3.6, the `secrets` module is available, which exposes cryptographically safe functionality.

Quoting the official documentation , to generate *"a ten-character alphanumeric password with at least one lowercase character, at least one uppercase character, and at least three digits,"* you could:

```
import string
alphabet = string.ascii_letters + string.digits
while True:
    password = ''.join(choice(alphabet) for i in range(10))
    if (any(c.islower() for c in password)
            and any(c.isupper() for c in password)
            and sum(c.isdigit() for c in password) >= 3):
        break
```

Create cryptographically secure random numbers

By default the Python random module use the Mersenne Twister PRNG to generate random numbers, which, although suitable in domains like simulations, fails to meet security requirements in more demanding environments.

In order to create a cryptographically secure pseudorandom number, one can use `SystemRandom` which, by using `os.urandom` , is able to act as a Cryptographically secure pseudorandom number generator, CPRNG .

The easiest way to use it simply involves initializing the `SystemRandom` class. The methods provided are similar to the ones exported by the random module.

```
from random import SystemRandom
secure_rand_gen = SystemRandom()
```

In order to create a random sequence of 10 `int` s in range `[0, 20]` , one can simply call `randrange()` :

```
print([secure_rand_gen.randrange(10) for i in range(10)])
# [9, 6, 9, 2, 2, 3, 8, 0, 9, 9]
```

To create a random integer in a given range, one can use `randint` :

```
print(secure_rand_gen.randint(0, 20))
# 5
```

and, accordingly for all other methods. The interface is exactly the same, the only change is the underlying

number generator.

You can also use `os.urandom` directly to obtain cryptographically secure random bytes.

Creating random integers and floats: randint, randrange, random, and uniform

```
import random
```

randint()

Returns a random integer between `x` and `y` (inclusive):

```
random.randint(x, y)
```

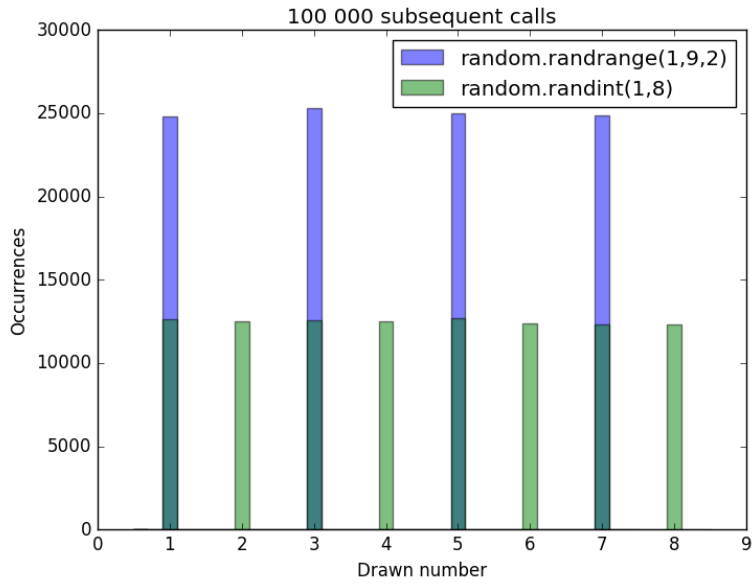
For example getting a random number between `1` and `8` :

```
random.randint(1, 8) # Out: 8
```

randrange()

`random.randrange` has the same syntax as `range` and unlike `random.randint` , the last value is **not** inclusive:

```
random.randrange(100)       # Random integer between 0 and 99
random.randrange(20, 50)    # Random integer between 20 and 49
random.rangrange(10, 20, 3) # Random integer between 10 and 19 with step 3 (10, 13, 16 and 19)
```



random

Returns a random floating point number between 0 and 1:

```
random.random() # Out: 0.66486093215306317
```

uniform

Returns a random floating point number between `x` and `y` (inclusive):

```
random.uniform(1, 8) # Out: 3.726062641730108
```

Random and sequences: shuffle, choice and sample

```
import random
```

shuffle()

You can use `random.shuffle()` to mix up/randomize the items in a **mutable and indexable** sequence. For example a `list` :

```
laughs = ["Hi", "Ho", "He"]

random.shuffle(laughs)     # Shuffles in-place! Don't do: laughs = random.shuffle(laughs)

print(laughs)
# Out: ["He", "Hi", "Ho"]  # Output may vary!
```

choice()

Takes a random element from an arbitary **sequence** :

```
print(random.choice(laughs))
# Out: He                  # Output may vary!
```

sample()

Like `choice` it takes random elements from an arbitary **sequence** but you can specify how many:

```
#                  |--sequence--|--number--|
print(random.sample(    laughs   ,     1    ))  # Take one element
# Out: ['Ho']                   # Output may vary!
```

it will not take the same element twice:

```
print(random.sample(laughs, 3))  # Take 3 random element from the sequence.
# Out: ['Ho', 'He', 'Hi']        # Output may vary!

print(random.sample(laughs, 4))  # Take 4 random element from the 3-item sequence.
```

> ValueError: Sample larger than population

Reproducible random numbers: Seed and State

Setting a specific Seed will create a fixed random-number series:

```
random.seed(5)                # Create a fixed state
print(random.randrange(0, 10))  # Get a random integer between 0 and 9
# Out: 9
print(random.randrange(0, 10))
# Out: 4
```

Resetting the seed will create the same "random" sequence again:

```
random.seed(5)                # Reset the random module to the same fixed state.
print(random.randrange(0, 10))
# Out: 9
print(random.randrange(0, 10))
# Out: 4
```

Since the seed is fixed these results are always `9` and `4` . If having specific numbers is not required only that the values will be the same one can also just use `getstate` and `setstate` to recover to a previous state:

```
save_state = random.getstate()  # Get the current state
print(random.randrange(0, 10))
# Out: 5
print(random.randrange(0, 10))
# Out: 8

random.setstate(save_state)     # Reset to saved state
print(random.randrange(0, 10))
# Out: 5
print(random.randrange(0, 10))
# Out: 8
```

To pseudo-randomize the sequence again you `seed` with `None` :

```
random.seed(None)
```

Or call the `seed` method with no arguments:

```
random.seed()
```

Random Binary Decision

```
import random

probability = 0.3

if random.random() < probability:
    print("Decision with probability 0.3")
else:
    print("Decision with probability 0.7")
```

Syntax

```
random.seed(a=None, version=2) (version is only avaiable for python 3.x)
random.getstate()
random.setstate(state)
random.randint(a, b)
random.randrange(stop)
random.randrange(start, stop, step=1)
random.choice(seq)
random.shuffle(x, random=random.random)
random.sample(population, k)
```

Parameters

Remarks