random — Pseudorandom Number Generators

Purpose: Implements several types of pseudorandom number generators.

The random module provides a fast pseudorandom number generator based on the *Mersenne Twister* algorithm. Originally developed to produce inputs for Monte Carlo simulations, Mersenne Twister generates numbers with nearly uniform distribution and a large period, making it suited for a wide range of applications.

Generating Random Numbers

The `random()` function returns the next random floating point value from the generated sequence. All of the return values fall within the range `0 <= n < 1.0`.

```
# random_random.py

import random

for i in range(5):
    print('%04.3f' % random.random(), end=' ')
print()
```

Running the program repeatedly produces different sequences of numbers.

```
$ python3 random_random.py

0.859 0.297 0.554 0.985 0.452

$ python3 random_random.py

0.797 0.658 0.170 0.297 0.593
```

To generate numbers in a specific numerical range, use `uniform()` instead.

```
# random_uniform.py

import random

for i in range(5):
    print('{:04.3f}'.format(random.uniform(1, 100)), end=' ')
print()
```

Pass minimum and maximum values, and `uniform()` adjusts the return values from `random()` using the formula `min + (max - min) * random()`.

```
$ python3 random_uniform.py

12.428 93.766 95.359 39.649 88.983
```

Seeding

`random()` produces different values each time it is called and has a very large period before it repeats any numbers. This is useful for producing unique values or variations, but there are times when having the same data set available to be processed in different ways is useful. One technique is to use a program to generate random values and save them to be processed by a separate step. That may not be practical for large amounts of data, though, so random includes the `seed()` function for initializing the pseudorandom generator so that it produces an expected set of values.

```
# random_seed.py

import random

random.seed(1)
```

```
random.seed(1)

for i in range(5):
    print('{:04.3f}'.format(random.random()), end=' ')
print()
```

The seed value controls the first value produced by the formula used to produce pseudorandom numbers, and since the formula is deterministic it also sets the full sequence produced after the seed is changed. The argument to seed() can be any hashable object. The default is to use a platform-specific source of randomness, if one is available. Otherwise, the current time is used.

```
$ python3 random_seed.py

0.134 0.847 0.764 0.255 0.495

$ python3 random_seed.py

0.134 0.847 0.764 0.255 0.495
```

Saving State

The internal state of the pseudorandom algorithm used by random() can be saved and used to control the numbers produced in subsequent runs. Restoring the previous state before continuing reduces the likelihood of repeating values or sequences of values from the earlier input. The getstate() function returns data that can be used to re-initialize the random number generator later with setstate().

```
# random_state.py
```

```
import random
import os
import pickle

if os.path.exists('state.dat'):
    # Restore the previously saved state
    print('Found state.dat, initializing random module')
    with open('state.dat', 'rb') as f:
        state = pickle.load(f)
    random.setstate(state)
else:
    # Use a well-known start state
    print('No state.dat, seeding')
    random.seed(1)

# Produce random values
for i in range(3):
    print('{:04.3f}'.format(random.random()), end=' ')
print()

# Save state for next time
with open('state.dat', 'wb') as f:
    pickle.dump(random.getstate(), f)

# Produce more random values
print('\nAfter saving state:')
for i in range(3):
    print('{:04.3f}'.format(random.random()), end=' ')
print()
```

The data returned by getstate() is an implementation detail, so this example saves the data to a file with pickle but otherwise treats it as a black box. If the file exists when the program starts, it loads the old state and continues. Each run produces a few numbers before and after saving the state, to show that restoring the state causes the generator to produce the same values again.

```
$ python3 random_state.py

No state.dat, seeding
0.134 0.847 0.764

After saving state:
0.255 0.495 0.449
```

```
0.255 0.495 0.449

$ python3 random_state.py

Found state.dat, initializing random module
0.255 0.495 0.449

After saving state:
0.652 0.789 0.094
```

Random Integers

`random()` generates floating point numbers. It is possible to convert the results to integers, but using `randint()` to generate integers directly is more convenient.

```
# random_randint.py

import random

print('[1, 100]:', end=' ')

for i in range(3):
    print(random.randint(1, 100), end=' ')

print('\n[-5, 5]:', end=' ')
for i in range(3):
    print(random.randint(-5, 5), end=' ')
print()
```

The arguments to `randint()` are the ends of the inclusive range for the values. The numbers can be positive or negative, but the first value should be less than the second.

```
$ python3 random_randint.py

[1, 100]: 98 75 34
[-5, 5]: 4 0 5
```

`randrange()` is a more general form of selecting values from a range.

```
# random_randrange.py

import random

for i in range(3):
    print(random.randrange(0, 101, 5), end=' ')
print()
```

`randrange()` supports a `step` argument, in addition to start and stop values, so it is fully equivalent to selecting a random value from `range(start, stop, step)`. It is more efficient, because the range is not actually constructed.

```
$ python3 random_randrange.py

15 20 85
```

Picking Random Items

One common use for random number generators is to select a random item from a sequence of enumerated values, even if those values are not numbers. `random` includes the `choice()` function for making a random selection from a sequence. This example simulates flipping a coin 10,000 times to count how many times it comes up heads and how many times tails.

```
# random_choice.py

import random
import itertools

outcomes = {
    'heads': 0,
```

```
    'tails': 0,
}
sides = list(outcomes.keys())

for i in range(10000):
    outcomes[random.choice(sides)] += 1

print('Heads:', outcomes['heads'])
print('Tails:', outcomes['tails'])
```

There are only two outcomes allowed, so rather than use numbers and convert them the words “heads” and “tails” are used with `choice()`. The results are tabulated in a dictionary using the outcome names as keys.

```
$ python3 random_choice.py

Heads: 5091
Tails: 4909
```

Permutations

A simulation of a card game needs to mix up the deck of cards and then deal them to the players, without using the same card more than once. Using `choice()` could result in the same card being dealt twice, so instead, the deck can be mixed up with `shuffle()` and then individual cards removed as they are dealt.

```
# random_shuffle.py

import random
import itertools

FACE_CARDS = ('J', 'Q', 'K', 'A')
SUITS = ('H', 'D', 'C', 'S')


def new_deck():
    return [
        # Always use 2 places for the value, so the strings
        # are a consistent width.
        '{:>2}{}'.format(*c)
        for c in itertools.product(
            itertools.chain(range(2, 11), FACE_CARDS),
            SUITS,
        )
    ]


def show_deck(deck):
    p_deck = deck[:]
    while p_deck:
        row = p_deck[:13]
        p_deck = p_deck[13:]
        for j in row:
            print(j, end=' ')
        print()


# Make a new deck, with the cards in order
deck = new_deck()
print('Initial deck:')
show_deck(deck)

# Shuffle the deck to randomize the order
random.shuffle(deck)
print('\nShuffled deck:')
show_deck(deck)

# Deal 4 hands of 5 cards each
hands = [[], [], [], []]

for i in range(5):
    for h in hands:
```

```
    for h in hands:
        h.append(deck.pop())

# Show the hands
print('\nHands:')
for n, h in enumerate(hands):
    print('{}:'.format(n + 1), end=' ')
    for c in h:
        print(c, end=' ')
    print()

# Show the remaining deck
print('\nRemaining deck:')
show_deck(deck)
```

The cards are represented as strings with the face value and a letter indicating the suit. The dealt "hands" are created by adding one card at a time to each of four lists, and removing it from the deck so it cannot be dealt again.

```
$ python3 random_shuffle.py

Initial deck:
 2H  2D  2C  2S  3H  3D  3C  3S  4H  4D  4C  4S  5H
 5D  5C  5S  6H  6D  6C  6S  7H  7D  7C  7S  8H  8D
 8C  8S  9H  9D  9C  9S 10H 10D 10C 10S  JH  JD  JC
 JS  QH  QD  QC  QS  KH  KD  KC  KS  AH  AD  AC  AS

Shuffled deck:
 QD  8C  JD  2S  AC  2C  6S  6D  6C  7H  JC  QS  QC
 KS  4D 10C  KH  5S  9C 10S  5C  7C  AS  6H  3C  9H
 4S  7S 10H  2D  8S  AH  9S  8H  QH  5D  5H  KD  8D
10D  4C  3S  3H  7D  AD  4H  9D  3D  2H  KC  JH  JS

Hands:
1:  JS  3D  7D 10D  5D
2:  JH  9D  3H  8D  QH
3:  KC  4H  3S  KD  8H
4:  2H  AD  4C  5H  9S

Remaining deck:
 QD  8C  JD  2S  AC  2C  6S  6D  6C  7H  JC  QS  QC
 KS  4D 10C  KH  5S  9C 10S  5C  7C  AS  6H  3C  9H
 4S  7S 10H  2D  8S  AH
```

Sampling

Many simulations need random samples from a population of input values. The `sample()` function generates samples without repeating values and without modifying the input sequence. This example prints a random sample of words from the system dictionary.

```
# random_sample.py

import random

with open('/usr/share/dict/words', 'rt') as f:
    words = f.readlines()
words = [w.rstrip() for w in words]

for w in random.sample(words, 5):
    print(w)
```

The algorithm for producing the result set takes into account the sizes of the input and the sample requested to produce the result as efficiently as possible.

```
$ python3 random_sample.py

streamlet
impestation
violaquercitrin
mycetoid
plethoretical
```

```
plethoretical

$ python3 random_sample.py

nonseditious
empyemic
ultrasonic
Kyurinish
amphide
```

Multiple Simultaneous Generators

In addition to module-level functions, `random` includes a `Random` class to manage the internal state for several random number generators. All of the functions described earlier are available as methods of the `Random` instances, and each instance can be initialized and used separately, without interfering with the values returned by other instances.

```
# random_random_class.py
```

```
import random
import time

print('Default initializiation:\n')

r1 = random.Random()
r2 = random.Random()

for i in range(3):
    print('{:04.3f}  {:04.3f}'.format(r1.random(), r2.random()))

print('\nSame seed:\n')

seed = time.time()
r1 = random.Random(seed)
r2 = random.Random(seed)

for i in range(3):
    print('{:04.3f}  {:04.3f}'.format(r1.random(), r2.random()))
```

On a system with good native random value seeding, the instances start out in unique states. However, if there is no good platform random value generator, the instances are likely to have been seeded with the current time, and therefore produce the same values.

```
$ python3 random_random_class.py

Default initializiation:

0.862  0.390
0.833  0.624
0.252  0.080

Same seed:

0.466  0.466
0.682  0.682
0.407  0.407
```

SystemRandom

Some operating systems provide a random number generator that has access to more sources of entropy that can be introduced into the generator. `random` exposes this feature through the `SystemRandom` class, which has the same API as `Random` but uses `os.urandom()` to generate the values that form the basis of all of the other algorithms.

```
# random_system_random.py
```

```
import random
import time

print('Default initializiation:\n')
```

```
r1 = random.SystemRandom()
r2 = random.SystemRandom()

for i in range(3):
    print('{:04.3f}  {:04.3f}'.format(r1.random(), r2.random()))

print('\nSame seed:\n')

seed = time.time()
r1 = random.SystemRandom(seed)
r2 = random.SystemRandom(seed)

for i in range(3):
    print('{:04.3f}  {:04.3f}'.format(r1.random(), r2.random()))
```

Sequences produced by `SystemRandom` are not reproducible because the randomness is coming from the system, rather than software state (in fact, `seed()` and `setstate()` have no effect at all).

```
$ python3 random_system_random.py

Default initializiation:

0.110  0.481
0.624  0.350
0.378  0.056

Same seed:

0.634  0.731
0.893  0.843
0.065  0.177
```

Non-uniform Distributions

While the uniform distribution of the values produced by `random()` is useful for a lot of purposes, other distributions more accurately model specific situations. The `random` module includes functions to produce values in those distributions, too. They are listed here, but not covered in detail because their uses tend to be specialized and require more complex examples.

Normal

The *normal* distribution is commonly used for non-uniform continuous values such as grades, heights, weights, etc. The curve produced by the distribution has a distinctive shape which has lead to it being nicknamed a “bell curve.” `random` includes two functions for generating values with a normal distribution, `normalvariate()` and the slightly faster `gauss()` (the normal distribution is also called the Gaussian distribution).

The related function, `lognormvariate()` produces pseudorandom values where the logarithm of the values is distributed normally. Log-normal distributions are useful for values that are the product of several random variables which do not interact.

Approximation

The *triangular* distribution is used as an approximate distribution for small sample sizes. The “curve” of a triangular distribution has low points at known minimum and maximum values, and a high point at the mode, which is estimated based on a “most likely” outcome (reflected by the mode argument to `triangular()`).

Exponential

`expovariate()` produces an exponential distribution useful for simulating arrival or interval time values for in homogeneous Poisson processes such as the rate of radioactive decay or requests coming into a web server.

The Pareto, or power law, distribution matches many observable phenomena and was popularized by *The Long Tail*, by Chris Anderson. The `paretovariate()` function is useful for simulating allocation of resources to individuals (wealth to people, demand for musicians, attention to blogs, etc.).

Angular

The von Mises, or circular normal, distribution (produced by `vonmisesvariate()`) is used for computing probabilities of cyclic values such as angles, calendar days, and times.

Sizes

`betavariate()` generates values with the Beta distribution, which is commonly used in Bayesian statistics and applications such as task duration modeling.

The Gamma distribution produced by `gammavariate()` is used for modeling the sizes of things such as waiting times, rainfall, and computational errors.

The Weibull distribution computed by `weibullvariate()` is used in failure analysis, industrial engineering, and weather forecasting. It describes the distribution of sizes of particles or other discrete objects.

See also

- Standard library documentation for random
- "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator" – Article by M. Matsumoto and T. Nishimura from *ACM Transactions on Modeling and Computer Simulation* Vol. 8, No. 1, January pp.3-30 1998.
- Wikipedia: Mersenne Twister – Article about the pseudorandom generator algorithm used by Python.
- Wikipedia: Uniform distribution – Article about continuous uniform distributions in statistics.

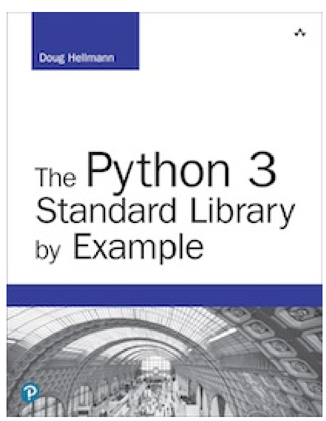
fractions — Rational Numbers
math — Mathematical Functions

Quick Links

This page was last updated 2016-12-28.

Navigation

fractions — Rational Numbers
math — Mathematical Functions



The output from all the example programs from PyMOTW-3 has been generated with Python 3.7.1, unless otherwise noted. Some of the features described here may not be available in earlier versions of Python.

Looking for examples for Python 2?