`random` — Generate pseudo-random numbers

Source code: Lib/random.py

This module implements pseudo-random number generators for various distributions.

For integers, there is uniform selection from a range. For sequences, there is uniform selection of a random element, a function to generate a random permutation of a list in-place, and a function for random sampling without replacement.

On the real line, there are functions to compute uniform, normal (Gaussian), lognormal, negative exponential, gamma, and beta distributions. For generating distributions of angles, the von Mises distribution is available.

Almost all module functions depend on the basic function `random()`, which generates a random float uniformly in the half-open range `0.0 <= X < 1.0`. Python uses the Mersenne Twister as the core generator. It produces 53-bit precision floats and has a period of 2**19937-1. The underlying implementation in C is both fast and threadsafe. The Mersenne Twister is one of the most extensively tested random number generators in existence. However, being completely deterministic, it is not suitable for all purposes, and is completely unsuitable for cryptographic purposes.

The functions supplied by this module are actually bound methods of a hidden instance of the `random.Random` class. You can instantiate your own instances of `Random` to get generators that don't share state.

Class `Random` can also be subclassed if you want to use a different basic generator of your own devising: see the documentation on that class for more details.

The `random` module also provides the `SystemRandom` class which uses the system function `os.urandom()` to generate random numbers from sources provided by the operating system.

Warning: The pseudo-random generators of this module should not be used for security purposes. For security or cryptographic uses, see the `secrets` module.

See also: M. Matsumoto and T. Nishimura, "Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator", ACM Transactions on Modeling and Computer Simulation Vol. 8, No. 1, January pp.3–30 1998.

Complementary-Multiply-with-Carry recipe for a compatible alternative random number generator with a long period and comparatively simple update operations.

Note: The global random number generator and instances of `Random` are thread-safe. However, in the free-threaded build, concurrent calls to the global generator or to the same instance of `Random` may encounter contention and poor performance. Consider using separate instances of `Random` per thread instead.

Bookkeeping functions

`random.`**`seed`**`(`*`a=None, version=2`*`)`

Initialize the random number generator.

If *a* is omitted or `None`, the current system time is used. If randomness sources are provided by the operating system, they are used instead of the system time (see the `os.urandom()` function for details on availability).

If *a* is an int, it is used directly.

With version 2 (the default), a `str`, `bytes`, or `bytearray` object gets converted to an `int` and all of its bits are used.

With version 1 (provided for reproducing random sequences from older versions of Python), the algorithm for `str` and `bytes` generates a narrower range of seeds.

> *Changed in version 3.2:* Moved to the version 2 scheme which uses all of the bits in a string seed.

> *Changed in version 3.11:* The *seed* must be one of the following types: `None`, `int`, `float`, `str`, `bytes`, or `bytearray`.

`random.`**`getstate`**`()`

Return an object capturing the current internal state of the generator. This object can be passed to `setstate()` to restore the state.

`random.`**`setstate`**`(`*`state`*`)`

state should have been obtained from a previous call to `getstate()`, and `setstate()` restores the internal state of the generator to what it was at the time `getstate()` was called.

Functions for bytes

`random.`**`randbytes`**`(`*`n`*`)`

Generate *n* random bytes.

This method should not be used for generating security tokens. Use `secrets.token_bytes()` instead.

> *Added in version 3.9.*

Functions for integers

`random.`**`randrange`**`(`*`stop`*`)`
`random.`**`randrange`**`(`*`start`*`, `*`stop`*`[, `*`step`*`])`

Return a randomly selected element from `range(start, stop, step)`.

This is roughly equivalent to `choice(range(start, stop, step))` but supports arbitrarily large ranges and is optimized for common cases.

The positional argument pattern matches the `range()` function.

Keyword arguments should not be used because they can be interpreted in unexpected ways. For example `randrange(start=100)` is interpreted as `randrange(0, 100, 1)`.

> *Changed in version 3.2:* `randrange()` is more sophisticated about producing equally distributed values. Formerly it used a style like `int(random()*n)` which could produce slightly uneven

distributions.

Changed in version 3.12: Automatic conversion of non-integer types is no longer supported. Calls such as `randrange(10.0)` and `randrange(Fraction(10, 1))` now raise a `TypeError`.

`random.`**`randint`**`(a, b)`

Return a random integer *N* such that `a <= N <= b`. Alias for `randrange(a, b+1)`.

`random.`**`getrandbits`**`(k)`

Returns a non-negative Python integer with *k* random bits. This method is supplied with the Mersenne Twister generator and some other generators may also provide it as an optional part of the API. When available, `getrandbits()` enables `randrange()` to handle arbitrarily large ranges.

Changed in version 3.9: This method now accepts zero for *k*.

Functions for sequences

`random.`**`choice`**`(seq)`

Return a random element from the non-empty sequence *seq*. If *seq* is empty, raises `IndexError`.

`random.`**`choices`**`(population, weights=None, *, cum_weights=None, k=1)`

Return a *k* sized list of elements chosen from the *population* with replacement. If the *population* is empty, raises `IndexError`.

If a *weights* sequence is specified, selections are made according to the relative weights. Alternatively, if a *cum_weights* sequence is given, the selections are made according to the cumulative weights (perhaps computed using `itertools.accumulate()`). For example, the relative weights `[10, 5, 30, 5]` are equivalent to the cumulative weights `[10, 15, 45, 50]`. Internally, the relative weights are converted to cumulative weights before making selections, so supplying the cumulative weights saves work.

If neither *weights* nor *cum_weights* are specified, selections are made with equal probability. If a weights sequence is supplied, it must be the same length as the *population* sequence. It is a `TypeError` to specify both *weights* and *cum_weights*.

The *weights* or *cum_weights* can use any numeric type that interoperates with the `float` values returned by `random()` (that includes integers, floats, and fractions but excludes decimals). Weights are assumed to be non-negative and finite. A `ValueError` is raised if all weights are zero.

For a given seed, the `choices()` function with equal weighting typically produces a different sequence than repeated calls to `choice()`. The algorithm used by `choices()` uses floating-point arithmetic for internal consistency and speed. The algorithm used by `choice()` defaults to integer arithmetic with repeated selections to avoid small biases from round-off error.

Added in version 3.6.

Changed in version 3.9: Raises a `ValueError` if all weights are zero.

`random.`**`shuffle`**`(x)`

Shuffle the sequence *x* in place.

To shuffle an immutable sequence and return a new shuffled list, use `sample(x, k=len(x))` instead.

Note that even for small `len(x)`, the total number of permutations of *x* can quickly grow larger than the period of most random number generators. This implies that most permutations of a long sequence can never be generated. For example, a sequence of length 2080 is the largest that can fit within the period of the Mersenne Twister random number generator.

> *Changed in version 3.11:* Removed the optional parameter *random*.

`random.`**`sample`**`(`*`population, k, *, counts=None`*`)`

Return a *k* length list of unique elements chosen from the population sequence. Used for random sampling without replacement.

Returns a new list containing elements from the population while leaving the original population unchanged. The resulting list is in selection order so that all sub-slices will also be valid random samples. This allows raffle winners (the sample) to be partitioned into grand prize and second place winners (the subslices).

Members of the population need not be hashable or unique. If the population contains repeats, then each occurrence is a possible selection in the sample.

Repeated elements can be specified one at a time or with the optional keyword-only *counts* parameter. For example, `sample(['red', 'blue'], counts=[4, 2], k=5)` is equivalent to `sample(['red', 'red', 'red', 'red', 'blue', 'blue'], k=5)`.

To choose a sample from a range of integers, use a `range()` object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(range(10000000), k=60)`.

If the sample size is larger than the population size, a `ValueError` is raised.

> *Changed in version 3.9:* Added the *counts* parameter.

> *Changed in version 3.11:* The *population* must be a sequence. Automatic conversion of sets to lists is no longer supported.

Discrete distributions

The following function generates a discrete distribution.

`random.`**`binomialvariate`**`(`*`n=1, p=0.5`*`)`

Binomial distribution. Return the number of successes for *n* independent trials with the probability of success in each trial being *p*:

Mathematically equivalent to:

```
sum(random() < p for i in range(n))
```

The number of trials *n* should be a non-negative integer. The probability of success *p* should be between `0.0 <= p <= 1.0`. The result is an integer in the range `0 <= X <= n`.

> *Added in version 3.12.*

Real-valued distributions

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distribution's equation, as used in common mathematical practice; most of these equations can be found in any statistics text.

`random.`**`random`**`()`

Return the next random floating-point number in the range `0.0 <= X < 1.0`

`random.`**`uniform`**`(`*`a`*`, `*`b`*`)`

Return a random floating-point number *N* such that `a <= N <= b` for `a <= b` and `b <= N <= a` for `b < a`.

The end-point value `b` may or may not be included in the range depending on floating-point rounding in the expression `a + (b-a) * random()`.

`random.`**`triangular`**`(`*`low`*`, `*`high`*`, `*`mode`*`)`

Return a random floating-point number *N* such that `low <= N <= high` and with the specified *mode* between those bounds. The *low* and *high* bounds default to zero and one. The *mode* argument defaults to the midpoint between the bounds, giving a symmetric distribution.

`random.`**`betavariate`**`(`*`alpha`*`, `*`beta`*`)`

Beta distribution. Conditions on the parameters are `alpha > 0` and `beta > 0`. Returned values range between 0 and 1.

`random.`**`expovariate`**`(`*`lambd=1.0`*`)`

Exponential distribution. *lambd* is 1.0 divided by the desired mean. It should be nonzero. (The parameter would be called “lambda”, but that is a reserved word in Python.) Returned values range from 0 to positive infinity if *lambd* is positive, and from negative infinity to 0 if *lambd* is negative.

Changed in version 3.12: Added the default value for `lambd`.

`random.`**`gammavariate`**`(`*`alpha`*`, `*`beta`*`)`

Gamma distribution. (*Not* the gamma function!) The shape and scale parameters, *alpha* and *beta*, must have positive values. (Calling conventions vary and some sources define ‘beta’ as the inverse of the scale).

The probability distribution function is:

```
          x ** (alpha - 1) * math.exp(-x / beta)
pdf(x) =  --------------------------------------
            math.gamma(alpha) * beta ** alpha
```

`random.`**`gauss`**`(`*`mu=0.0`*`, `*`sigma=1.0`*`)`

Normal distribution, also called the Gaussian distribution. *mu* is the mean, and *sigma* is the standard deviation. This is slightly faster than the `normalvariate()` function defined below.

Multithreading note: When two threads call this function simultaneously, it is possible that they will receive the same return value. This can be avoided in three ways. 1) Have each thread use a different instance of the random number generator. 2) Put locks around all calls. 3) Use the slower, but thread-safe `normalvariate()` function instead.

Changed in version 3.11: *mu* and *sigma* now have default arguments.

`random.`**`lognormvariate`**`(`*`mu`*`, `*`sigma`*`)`

Log normal distribution. If you take the natural logarithm of this distribution, you'll get a normal distribution with mean *mu* and standard deviation *sigma*. *mu* can have any value, and *sigma* must be greater than zero.

`random.`**`normalvariate`**`(`*`mu=0.0, sigma=1.0`*`)`

Normal distribution. *mu* is the mean, and *sigma* is the standard deviation.

> *Changed in version 3.11:* *mu* and *sigma* now have default arguments.

`random.`**`vonmisesvariate`**`(`*`mu, kappa`*`)`

mu is the mean angle, expressed in radians between 0 and 2**pi*, and *kappa* is the concentration parameter, which must be greater than or equal to zero. If *kappa* is equal to zero, this distribution reduces to a uniform random angle over the range 0 to 2**pi*.

`random.`**`paretovariate`**`(`*`alpha`*`)`

Pareto distribution. *alpha* is the shape parameter.

`random.`**`weibullvariate`**`(`*`alpha, beta`*`)`

Weibull distribution. *alpha* is the scale parameter and *beta* is the shape parameter.

Alternative Generator

`class` `random.`**`Random`**`([`*`seed`*`])`

Class that implements the default pseudo-random number generator used by the `random` module.

> *Changed in version 3.11:* Formerly the *seed* could be any hashable object. Now it is limited to: `None`, `int`, `float`, `str`, `bytes`, or `bytearray`.

Subclasses of `Random` should override the following methods if they wish to make use of a different basic generator:

`seed``(`*`a=None, version=2`*`)`

Override this method in subclasses to customise the `seed()` behaviour of `Random` instances.

`getstate``()`

Override this method in subclasses to customise the `getstate()` behaviour of `Random` instances.

`setstate``(`*`state`*`)`

Override this method in subclasses to customise the `setstate()` behaviour of `Random` instances.

`random``()`

Override this method in subclasses to customise the `random()` behaviour of `Random` instances.

Optionally, a custom generator subclass can also supply the following method:

`getrandbits``(`*`k`*`)`

Override this method in subclasses to customise the `getrandbits()` behaviour of `Random` instances.

`class` `random.`**`SystemRandom`**`([`*`seed`*`])`

Class that uses the `os.urandom()` function for generating random numbers from sources provided by the operating system. Not available on all systems. Does not rely on software state, and sequences are not reproducible. Accordingly, the `seed()` method has no effect and is ignored. The `getstate()` and `setstate()` methods raise `NotImplementedError` if called.

Notes on Reproducibility

Sometimes it is useful to be able to reproduce the sequences given by a pseudo-random number generator. By reusing a seed value, the same sequence should be reproducible from run to run as long as multiple threads are not running.

Most of the random module's algorithms and seeding functions are subject to change across Python versions, but two aspects are guaranteed not to change:

- If a new seeding method is added, then a backward compatible seeder will be offered.
- The generator's `random()` method will continue to produce the same sequence when the compatible seeder is given the same seed.

Examples

Basic examples:

```
>>> random()                             # Random float:  0.0 <= x < 1.0
0.37444887175646646

>>> uniform(2.5, 10.0)                   # Random float:  2.5 <= x <= 10.0
3.1800146073117523

>>> expovariate(1 / 5)                   # Interval between arrivals averaging 5 seconds
5.148957571865031

>>> randrange(10)                        # Integer from 0 to 9 inclusive
7

>>> randrange(0, 101, 2)                 # Even integer from 0 to 100 inclusive
26

>>> choice(['win', 'lose', 'draw'])      # Single random element from a sequence
'draw'

>>> deck = 'ace two three four'.split()
>>> shuffle(deck)                        # Shuffle a list
>>> deck
['four', 'two', 'ace', 'three']

>>> sample([10, 20, 30, 40, 50], k=4)    # Four samples without replacement
[40, 10, 50, 30]
```

Simulations:

```
>>> # Six roulette wheel spins (weighted sampling with replacement)
>>> choices(['red', 'black', 'green'], [18, 18, 2], k=6)
['red', 'green', 'black', 'black', 'red', 'black']

>>> # Deal 20 cards without replacement from a deck
>>> # of 52 playing cards, and determine the proportion of cards
>>> # with a ten-value:  ten, jack, queen, or king.
>>> deal = sample(['tens', 'low cards'], counts=[16, 36], k=20)
>>> deal.count('tens') / 20
0.15

>>> # Estimate the probability of getting 5 or more heads from 7 spins
>>> # of a biased coin that settles on heads 60% of the time.
>>> sum(binomialvariate(n=7, p=0.6) >= 5 for i in range(10_000)) / 10_000
0.4169

>>> # Probability of the median of 5 samples being in middle two quartiles
>>> def trial():
...     return 2_500 <= sorted(choices(range(10_000), k=5))[2] < 7_500
...
>>> sum(trial() for i in range(10_000)) / 10_000
0.7958
```

Example of statistical bootstrapping using resampling with replacement to estimate a confidence interval for the mean of a sample:

```
# https://www.thoughtco.com/example-of-bootstrapping-3126155
from statistics import fmean as mean
from random import choices

data = [41, 50, 29, 37, 81, 30, 73, 63, 20, 35, 68, 22, 60, 31, 95]
means = sorted(mean(choices(data, k=len(data))) for i in range(100))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[5]:.1f} to {means[94]:.1f}')
```

Example of a resampling permutation test to determine the statistical significance or p-value of an observed difference between the effects of a drug versus a placebo:

```
# Example from "Statistics is Easy" by Dennis Shasha and Manda Wilson
from statistics import fmean as mean
from random import shuffle

drug = [54, 73, 53, 70, 73, 68, 52, 65, 65]
placebo = [54, 51, 58, 44, 55, 52, 42, 47, 58, 46]
observed_diff = mean(drug) - mean(placebo)

n = 10_000
count = 0
combined = drug + placebo
for i in range(n):
    shuffle(combined)
    new_diff = mean(combined[:len(drug)]) - mean(combined[len(drug):])
    count += (new_diff >= observed_diff)

print(f'{n} label reshufflings produced only {count} instances with a difference')
print(f'at least as extreme as the observed difference of {observed_diff:.1f}.')
print(f'The one-sided p-value of {count / n:.4f} leads us to reject the null')
print(f'hypothesis that there is no difference between the drug and the placebo.')
```

Simulation of arrival times and service deliveries for a multiserver queue:

```
from heapq import heapify, heapreplace
from random import expovariate, gauss
from statistics import mean, quantiles

average_arrival_interval = 5.6
average_service_time = 15.0
stdev_service_time = 3.5
```

```
num_servers = 3

waits = []
arrival_time = 0.0
servers = [0.0] * num_servers  # time when each server becomes available
heapify(servers)
for i in range(1_000_000):
    arrival_time += expovariate(1.0 / average_arrival_interval)
    next_server_available = servers[0]
    wait = max(0.0, next_server_available - arrival_time)
    waits.append(wait)
    service_duration = max(0.0, gauss(average_service_time, stdev_service_time))
    service_completed = arrival_time + wait + service_duration
    heapreplace(servers, service_completed)

print(f'Mean wait: {mean(waits):.1f}   Max wait: {max(waits):.1f}')
print('Quartiles:', [round(q, 1) for q in quantiles(waits)])
```

See also: Statistics for Hackers a video tutorial by Jake Vanderplas on statistical analysis using just a few fundamental concepts including simulation, sampling, shuffling, and cross-validation.

Economics Simulation a simulation of a marketplace by Peter Norvig that shows effective use of many of the tools and distributions provided by this module (gauss, uniform, sample, betavariate, choice, triangular, and randrange).

A Concrete Introduction to Probability (using Python) a tutorial by Peter Norvig covering the basics of probability theory, how to write simulations, and how to perform data analysis using Python.

Recipes

These recipes show how to efficiently make random selections from the combinatoric iterators in the `itertools` module:

```
def random_product(*args, repeat=1):
    "Random selection from itertools.product(*args, **kwds)"
    pools = [tuple(pool) for pool in args] * repeat
    return tuple(map(random.choice, pools))

def random_permutation(iterable, r=None):
    "Random selection from itertools.permutations(iterable, r)"
    pool = tuple(iterable)
    r = len(pool) if r is None else r
    return tuple(random.sample(pool, r))

def random_combination(iterable, r):
    "Random selection from itertools.combinations(iterable, r)"
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range(n), r))
    return tuple(pool[i] for i in indices)

def random_combination_with_replacement(iterable, r):
    "Choose r elements with replacement.  Order the result to match the iterable."
    # Result will be in set(itertools.combinations_with_replacement(iterable, r)).
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.choices(range(n), k=r))
    return tuple(pool[i] for i in indices)
```

The default `random()` returns multiples of 2^{-53} in the range $0.0 \le x < 1.0$. All such numbers are evenly spaced and are exactly representable as Python floats. However, many other representable floats in that interval are not possible selections. For example, `0.05954861408025609` isn't an integer multiple of 2^{-53}.

The following recipe takes a different approach. All floats in the interval are possible selections. The mantissa comes from a uniform distribution of integers in the range $2^{52} \le mantissa < 2^{53}$. The exponent comes from a geometric distribution where exponents smaller than *-53* occur half as often as the next larger exponent.

```
from random import Random
from math import ldexp

class FullRandom(Random):

    def random(self):
        mantissa = 0x10_0000_0000_0000 | self.getrandbits(52)
        exponent = -53
        x = 0
        while not x:
            x = self.getrandbits(32)
            exponent += x.bit_length() - 32
        return ldexp(mantissa, exponent)
```

All real valued distributions in the class will use the new method:

```
>>> fr = FullRandom()
>>> fr.random()
0.05954861408025609
>>> fr.expovariate(0.25)
8.87925541791544
```

The recipe is conceptually equivalent to an algorithm that chooses from all the multiples of 2^{-1074} in the range $0.0 \le x < 1.0$. All such numbers are evenly spaced, but most have to be rounded down to the nearest representable Python float. (The value 2^{-1074} is the smallest positive unnormalized float and is equal to `math.ulp(0.0)`.)

See also: Generating Pseudo-random Floating-Point Values a paper by Allen B. Downey describing ways to generate more fine-grained floats than normally generated by `random()`.

Command-line usage

Added in version 3.13.

The `random` module can be executed from the command line.

```
python -m random [-h] [-c CHOICE [CHOICE ...] | -i N | -f N] [input ...]
```

The following options are accepted:

`-h, --help`

Show the help message and exit.

`-c CHOICE [CHOICE ...]`
`--choice CHOICE [CHOICE ...]`

Print a random choice, using `choice()`.

-i <N>
--integer <N>

Print a random integer between 1 and N inclusive, using `randint()`.

-f <N>
--float <N>

Print a random floating-point number between 0 and N inclusive, using `uniform()`.

If no options are given, the output depends on the input:

- String or multiple: same as `--choice`.
- Integer: same as `--integer`.
- Float: same as `--float`.

Command-line example

Here are some examples of the `random` command-line interface:

```
$ # Choose one at random
$ python -m random egg bacon sausage spam "Lobster Thermidor aux crevettes with a Mornay sau
Lobster Thermidor aux crevettes with a Mornay sauce

$ # Random integer
$ python -m random 6
6

$ # Random floating-point number
$ python -m random 1.8
1.7080016272295635

$ # With explicit arguments
$ python  -m random --choice egg bacon sausage spam "Lobster Thermidor aux crevettes with a
egg

$ python -m random --integer 6
3

$ python -m random --float 1.8
1.5666339105010318

$ python -m random --integer 6
5

$ python -m random --float 6
3.1942323316565915
```