9.6. 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 semi-open range [0.0, 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: in that case, override the random(), seed(), getstate(), and setstate() methods. Optionally, a

new generator can supply a getrandbits() method — this allows randrange() to produce selections over an arbitrarily large range.

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.

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.

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.

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

Returns a Python integer with *k* random bits. This method is supplied with the MersenneTwister 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.

Functions for integers:

random.**randrange**(*stop*)

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

Return a randomly selected element from `range(start, stop, step)`. This is equivalent to `choice(range(start, stop, step))`, but doesn't actually build a range object.

The positional argument pattern matches that of range(). Keyword arguments should not be used because the function may use them in unexpected ways.

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.

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

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

Functions for sequences:

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

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

random.**shuffle**(*x*[, *random*])

Shuffle the sequence *x* in place. The optional argument *random* is a 0-argument function returning a random float in [0.0, 1.0); by default, this is the function random().

Note that for even rather small `len(x)`, the total number of permutations of *x* is larger than the period of most random number generators; this implies that most permutations of a long sequence can never be generated.

random.**sample**(*population*, *k*)

Return a *k* length list of unique elements chosen from the population sequence or set. 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.

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

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

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, 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 equation `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*)

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.

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

Gamma distribution. (*Not* the gamma function!) Conditions on the parameters are `alpha > 0` and `beta > 0`.

The probability distribution function is:

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

```
        math.gamma(alpha) * beta ** alpha
```

random.**gauss**(*mu*, *sigma*)

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

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*, *sigma*)

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

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.**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.

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.

9.6.1. Notes on Reproducibility

Sometimes it is useful to be able to reproduce the sequences given by a pseudo random number generator. By re-using 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.

9.6.2. Examples and Recipes

Basic usage:

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

>>> random.uniform(1, 10)          # Random float x, 1.0 <= x < 10.0
```

```
1.1800146073117523

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

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

>>> random.choice('abcdefghij')      # Single random element
'c'

>>> items = [1, 2, 3, 4, 5, 6, 7]
>>> random.shuffle(items)
>>> items
[7, 3, 2, 5, 6, 4, 1]

>>> random.sample([1, 2, 3, 4, 5],  3)   # Three samples without replace
[4, 1, 5]
```

A common task is to make a random.choice() with weighted probabilities.

If the weights are small integer ratios, a simple technique is to build a sample population with repeats:

```
>>> weighted_choices = [('Red', 3), ('Blue', 2), ('Yellow', 1), ('Green
>>> population = [val for val, cnt in weighted_choices for i in range(
>>> population
['Red', 'Red', 'Red', 'Blue', 'Blue', 'Yellow', 'Green', 'Green', 'Gree

>>> random.choice(population)
'Green'
```

A more general approach is to arrange the weights in a cumulative distribution with itertools.accumulate(), and then locate the random value with bisect.bisect():

```
>>> choices, weights = zip(*weighted_choices)
>>> cumdist = list(itertools.accumulate(weights))
>>> cumdist             # [3, 3+2, 3+2+1, 3+2+1+4]
[3, 5, 6, 10]
```

```
>>> x = random.random() * cumdist[-1]
>>> choices[bisect.bisect(cumdist, x)]
'Blue'
```