10.6 random — Generate pseudo-random numbers

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

For integers, uniform selection from a range. For sequences, 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. This is especially useful for multi-threaded programs, creating a different instance of Random for each thread, and using the jumpahead() method to make it likely that the generated sequences seen by each thread don't overlap.

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(), setstate() and jumpahead() methods. Optionally, a new generator can supply a getrandbits() method — this allows randrange() to produce selections over an arbitrarily large range. New in version 2.4: the getrandbits() method. As an example of subclassing, the random module provides the WichmannHill class that implements an alternative generator in pure Python. The class provides a backward compatible way to reproduce results from earlier versions of Python, which used the Wichmann-Hill algorithm as the core generator. Note that this Wichmann-Hill generator can no longer be recommended: its period is too short by contemporary standards, and the sequence generated is known to fail some stringent randomness tests. See the references below for a recent variant that repairs these flaws. Changed in version 2.3: Substituted MersenneTwister for Wichmann-Hill. Bookkeeping functions:

seed(*[x]*)
: Initialize the basic random number generator. Optional argument *x* can be any *hashable* object. If *x* is omitted or None, current system time is used; current system time is also used to initialize the generator when the module is first imported. 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). Changed in version 2.4: formerly, operating system resources were not used. If *x* is not None or an int or long, hash(x) is used instead. If *x* is an int or long, *x* is used directly.

getstate()
: Return an object capturing the current internal state of the generator. This object can be passed to setstate() to restore the state. New in version 2.1.Changed in version 2.6: State values produced in Python 2.6 cannot be loaded into earlier versions.

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 setstate() was called. New in version 2.1.

jumpahead(*n*)
: Change the internal state to one different from and likely far away from the current state. *n* is a non-negative integer which is used to scramble the current state vector. This is most useful in multi-threaded programs, in conjunction with multiple instances of the Random class: setstate() or seed() can be used to force all instances into the same internal state, and then jumpahead() can be used to force the instances' states far apart. New in version 2.1.Changed in version 2.3: Instead of jumping to a specific state, *n* steps ahead, jumpahead(n) jumps to another state likely to be separated by many steps.

getrandbits(*k*)
: Returns a python long int 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. New in version 2.4.

Functions for integers:

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. New in version 1.5.2.

randint(*a, b*)
Return a random integer *N* such that `a <= N <= b`.

Functions for sequences:

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

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.

sample(*population, k*)
Return a *k* length list of unique elements chosen from the population sequence. Used for random sampling without replacement. New in version 2.3. 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 `xrange()` object as an argument. This is especially fast and space efficient for sampling from a large population: `sample(xrange(10000000), 60)`.

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()
Return the next random floating point number in the range [0.0, 1.0).

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`.

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. New in version 2.6.

betavariate(*alpha, beta*)
Beta distribution. Conditions on the parameters are `alpha > 0` and `beta > 0`. Returned values range between 0 and 1.

expovariate(*lambd*)
Exponential distribution. *lambd* is 1.0 divided by the desired mean. (The parameter would be called "lambda", but that is a reserved word in Python.) Returned values range from 0 to positive infinity.

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

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.

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.

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

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*.

paretovariate(*alpha*)
Pareto distribution. *alpha* is the shape parameter.

weibullvariate(*alpha, beta*)
Weibull distribution. *alpha* is the scale parameter and *beta* is the shape parameter.

Alternative Generators:

class WichmannHill(*[seed]*)
Class that implements the Wichmann-Hill algorithm as the core generator. Has all of the same methods as `Random` plus the `whseed()` method described below. Because this class is implemented in pure Python, it is not threadsafe and may require locks between calls. The period of the generator is 6,953,607,871,644 which is small enough to require care that two independent random sequences do not overlap.

whseed(*[x]*)
This is obsolete, supplied for bit-level compatibility with versions of Python prior to 2.1. See `seed()` for details. `whseed()` does not guarantee that distinct integer arguments yield distinct internal states, and can yield no more than about 2**24 distinct internal states in all.

class 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()` and `jumpahead()` methods have no effect and are ignored. The `getstate()` and `setstate()` methods raise `NotImplementedError` if called. New in version 2.4.

Examples of 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.randint(1, 10)   # Integer from 1 to 10, endpoints included
7
>>> random.randrange(0, 101, 2)  # Even integer from 0 to 100
26
>>> random.choice('abcdefghij')  # Choose a 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)  # Choose 3 elements
```

```
[4, 1, 5]
```

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.

Wichmann, B. A. & Hill, I. D., "Algorithm AS 183: An efficient and portable pseudo-random number generator", Applied Statistics 31 (1982) 188-190.

10.7 `itertools` — Functions creating iterators for efficient looping

New in version 2.3. This module implements a number of *iterator* building blocks inspired by constructs from the Haskell and SML programming languages. Each has been recast in a form suitable for Python.

The module standardizes a core set of fast, memory efficient tools that are useful by themselves or in combination. Standardization helps avoid the readability and reliability problems which arise when many different individuals create their own slightly varying implementations, each with their own quirks and naming conventions.

The tools are designed to combine readily with one another. This makes it easy to construct more specialized tools succinctly and efficiently in pure Python.

For instance, SML provides a tabulation tool: `tabulate(f)` which produces a sequence `f(0), f(1), ...`. This toolbox provides `imap()` and `count()` which can be combined to form `imap(f, count())` and produce an equivalent result.

Likewise, the functional tools are designed to work well with the high-speed functions provided by the `operator` module.

Whether cast in pure python form or compiled code, tools that use iterators are more memory efficient (and often faster) than their list based counterparts. Adopting the principles of just-in-time manufacturing, they create data when and where needed instead of consuming memory with the computer equivalent of "inventory".

See Also:

The Standard ML Basis Library, The Standard ML Basis Library.

Haskell, A Purely Functional Language, Definition of Haskell and the Standard Libraries.

10.7.1 Itertool functions

The following module functions all construct and return iterators. Some provide streams of infinite length, so they should only be accessed by functions or loops that truncate the stream.

chain(**iterables*)
Make an iterator that returns elements from the first iterable until it is exhausted, then proceeds to the next iterable, until all of the iterables are exhausted. Used for treating consecutive sequences as a single sequence. Equivalent to:

```
def chain(*iterables):
    # chain('ABC', 'DEF') --> A B C D E F
    for it in iterables:
        for element in it:
            yield element
```

from_iterable(*iterable*)
Alternate constructor for `chain()`. Gets chained inputs from a single iterable argument that is evaluated lazily.