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 thread-safe. 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. For security or cryptographic uses, see the `secrets` module.

9.6.1. 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.

9.6.2. 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)`.

9.6.3. 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).

New in version 3.6.

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()`.

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.

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 sub-slices).

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 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.

9.6.4. 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, 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.

9.6.5. 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.

9.6.6. 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.

9.6.7. Examples and Recipes

Basic examples:

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

>>> uniform(2.5, 10.0)                     # Random float:  2.5 <= x < 1
3.1800146073117523

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

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

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

>>> choice(['win', 'lose', 'draw'])       # Single random element from
'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 replac
[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 card
>>> # and determine the proportion of cards with a ten-value
>>> # (a ten, jack, queen, or king).
>>> deck = collections.Counter(tens=16, low_cards=36)
>>> seen = sample(list(deck.elements()), k=20)
>>> seen.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.
>>> trial = lambda: choices('HT', cum_weights=(0.60, 1.00), k=7).count
>>> sum(trial() for i in range(10000)) / 10000
0.4169
```

```
>>> # Probability of the median of 5 samples being in middle two quart
>>> trial = lambda : 2500 <= sorted(choices(range(10000), k=5))[2] <
>>> sum(trial() for i in range(10000)) / 10000
0.7958
```

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

```
# http://statistics.about.com/od/Applications/a/Example-Of-Bootstrapp
from statistics import mean
from random import choices

data = 1, 2, 4, 4, 10
means = sorted(mean(choices(data, k=5)) for i in range(20))
print(f'The sample mean of {mean(data):.1f} has a 90% confidence '
      f'interval from {means[1]:.1f} to {means[-2]:.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 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 = 10000
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
print(f'at least as extreme as the observed difference of {observed_d
print(f'The one-sided p-value of {count / n:.4f} leads us to reject th
print(f'hypothesis that there is no difference between the drug and th
```

Simulation of arrival times and service deliveries in a single server queue:

```
from random import expovariate, gauss
from statistics import mean, median, stdev

average_arrival_interval = 5.6
```

```
average_service_time = 5.0
stdev_service_time = 0.5

num_waiting = 0
arrivals = []
starts = []
arrival = service_end = 0.0
for i in range(20000):
    if arrival <= service_end:
        num_waiting += 1
        arrival += expovariate(1.0 / average_arrival_interval)
        arrivals.append(arrival)
    else:
        num_waiting -= 1
        service_start = service_end if num_waiting else arrival
        service_time = gauss(average_service_time, stdev_service_time)
        service_end = service_start + service_time
        starts.append(service_start)

waits = [start - arrival for arrival, start in zip(arrivals, starts)]
print(f'Mean wait: {mean(waits):.1f}.  Stdev wait: {stdev(waits):.1f}.
print(f'Median wait: {median(waits):.1f}.  Max wait: {max(waits):.1f}.
```

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.