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.

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: *NoneType*, `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.

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.

New 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 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.

Deprecated since version 3.10: The automatic conversion of non-integer types to equivalent

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*.

Real-valued distributions

The following functions generate specific real-valued distributions. Function parameters are named after the corresponding variables in the distributio equation, as used in common mathematical practi most of these equations can be found in any statis ics text.

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

Return the next random floating point numbe 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`*`)`

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.pro
    pools = [tuple(pool) for pool in arg
    return tuple(map(random.choice, pool

def random_permutation(iterable, r=None)
    "Random selection from itertools.per
    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.com
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.sample(range
    return tuple(pool[i] for i in indice

def random_combination_with_replacement(
    "Random selection from itertools.com
    pool = tuple(iterable)
    n = len(pool)
    indices = sorted(random.choices(rang
    return tuple(pool[i] for i in indice
```