

---

# **SampleSpace**

***Release 1.0.2***

**Coriander V. Pines**

**Jun 28, 2020**



## SUBMODULES

1	<code>samplespace.repeatablerandom</code> - Repeatable Random Sequences	3
2	<code>samplespace.distributions</code> - Serializable Probability Distributions	15
3	<code>samplespace.algorithms</code> - General Sampling Algorithms	31
4	<code>samplespace.pyyaml_support</code> - YAML serialization support	33
5	Random Generator Quality	37
	Python Module Index	41
	Index	43





SampleSpace is a cross-platform library for describing and sampling from random distributions.

While SampleSpace is primarily intended for creating procedurally-generated content, it is also useful for Monte-Carlo simulations, unit testing, and anywhere that flexible, repeatable random numbers are required.

Installation is simple:

```
$ pip install pysamplespace
```

SampleSpace's only dependency is [xxHash](#), though it optionally offers additional functionality if [PyYAML](#) is installed.

SampleSpace was created by Coriander V. Pines and is available under the BSD 3-Clause License.

The source is available on [GitLab](#).



## SAMPLESPACE.REPEATABLERANDOM - REPEATABLE RANDOM SEQUENCES

*RepeatableRandomSequence* allows for generating repeatable, deterministic random sequences. It is compatible with the built-in `random` module as a drop-in replacement.

A key feature of *RepeatableRandomSequence* is its ability to get, serialize, and restore internal state. This is especially useful when generating procedural content from a fixed seed.

A *RepeatableRandomSequence* can also be used for unit testing by replacing the built-in `random` module. Because each random sequence is deterministic and repeatable for a given seed, expected values can be recorded and compared against within unit tests.

*RepeatableRandomSequence* produces high-quality pseudo-random values. See *Random Generator Quality* for results from randomness tests.

**class** `samplespace.repeatablerandom.RepeatableRandomSequence` (*seed=None*)

A deterministic and repeatable random number generator compatible with Python's builtin `random` module.

**Parameters** *seed* (*int*, *str*, *bytes*, *bytearray*) – The sequence's initial seed. See the *seed()* method below for more details.

`RepeatableRandomSequence.BLOCK_SIZE_BITS: int = 64`

The number of bits generated for each unique index.

`RepeatableRandomSequence.BLOCK_MASK: int = 18446744073709551615`

A bitmask corresponding to  $(1 \ll \text{BLOCK\_SIZE\_BITS}) - 1$

### 1.1 Bookkeeping functions

`RepeatableRandomSequence.seed` (*value=None*) → `None`

Re-initialize the random generator with a new seed. Resets the sequence to its first value.

**Caution:** This method cannot be called from within *cascade()*, and will raise a `RuntimeError` if attempted.

**Parameters** *value* (*int*, *str*, *bytes*, *bytearray*) – The value to seed with. If this is a sequence type, the sequence is first hashed with seed 0.

**Raises** `ValueError` – If the seed value is not a supported type.

`RepeatableRandomSequence.getseed()`

Returns: the original value passed to `RepeatableRandomSequence()` or `seed()`.

`RepeatableRandomSequence.getstate()` → `samplespace.repeatablerandom.RepeatableRandomSequenceState`

Returns an opaque object representing the sequence's current state, used for saving, restoring, and serializing the sequence. This object can be passed to `setstate()` to restore the sequence's state.

`RepeatableRandomSequence.setstate(state: samplespace.repeatablerandom.RepeatableRandomSequenceState)`  
→ `None`

Restore the sequence's state from previous call to `getstate()`.

`RepeatableRandomSequence.reset()` → `None`

Reset the sequence to the beginning, setting `index` to 0.

**Caution:** This method cannot be called from within `cascade()`, and will raise a `RuntimeError` if attempted.

`RepeatableRandomSequence.index`

The sequence's current index. Generating random values will always increase the index.

---

**Tip:** Prefer `getstate()` and `setstate()` over index manipulation when saving and restoring state.

---

**Caution:** The index cannot be read or written within `cascade()`, and will raise a `RuntimeError` if attempted.

**Type** `int`

`RepeatableRandomSequence.getnextblock()` → `int`

Get a block of random bits from the random generator.

---

**Tip:** Calling this method will advance the sequence exactly one time. The resulting index depends on whether or not the call occurs within a cascade.

---

**Returns** A block of `BLOCK_SIZE_BITS` random bits as an `int`

`RepeatableRandomSequence.getrandbits(k: int)` → `int`

Generate an `int` with `k` random bits.

This method may call `getnextblock()` multiple times if `k` is greater than `BLOCK_SIZE_BITS`; regardless of the number of calls, the index will only advance once.

---

**Tip:** Note that `k == 0` is a valid input, which will advance the sequence even though no elements are chosen.

---

**Parameters** `k (int)` – The number of random bits to generate.

**Returns** A random integer in `[0, max(2k, 1))`

`RepeatableRandomSequence.cascade()`

Returns a context manager that defines a generation cascade.



Cascades are not required for most applications, but may be useful for certain advanced procedural generation techniques.

A cascade allows multiple random samples to be treated as part the same logical unit.

For example, generating a random position in space can be treated as a single “transaction” by grouping the fields into a single cascade:

```
>>> print (rrs.index)
10
>>> with rrs.cascade():
...     x = rrs.random() # index is 10
...     y = rrs.random() # index is the previously-generated random block
...     z = rrs.random() # index is the previously generated random block
...
>>> print (rrs.index)
11
```

The use of cascades ensure that random values are repeatable and based only on the number of previously-generated values, not the type of the values themselves. This is often useful when creating procedurally-generated content using pre-defined seeds.

While cascading, subsequent calls to random generation functions use the result of the most recently generated random data rather than incrementing the index. When a cascade completes, the index is set to one more than the index when the cascade began.

```
class samplespace.repeatablerandom.RepeatableRandomSequenceState(_seed: Any,
                                                                    _hash_input:
                                                                    bytes, _in-
                                                                    dex: int)
```

An object representing a *RepeatableRandomSequence*’s internal state.

**as\_dict()**

Return the sequence state as a dictionary for serialization.

**classmethod from\_dict(as\_dict)**

Construct a new sequence state from a dictionary, as returned by *as\_dict()*.

## Examples

```
>>> rrs = RepeatableRandomSequence()
>>>
>>> state_as_dict = rrs.getstate().as_dict()
>>> state_as_dict
{'seed': 0, 'hash_input': 'NmlqzcrbG7s=', 'index': 0}
>>>
>>> new_state = RepeatableRandomSequenceState.from_dict(state_as_dict)
>>> rrs.setstate(new_state)
```

## 1.2 Integer distributions

`RepeatableRandomSequence.randrange` (*start*: `int`, *stop*: `Optional[int] = None`, *step*: `int = 1`) → `int`  
Generate a random integer from `range(start[, stop[, step]])`.

If only one argument is provided, samples from `range(start)`.

### Parameters

- **start** (`int`) – The starting point for the range, inclusive. If *stop* is `None`, this becomes the endpoint for the range, exclusive. *None None*
- **stop** (`int`, *optional*) – The end point for the range, exclusive.
- **step** (`int`, *default 1*) – Steps between possible values. May not be 0.

### Raises

- **TypeError** – if any arguments are not integral.
- **ValueError** – if step is 0

`RepeatableRandomSequence.randint` (*a*: `int`, *b*: `int`) → `int`  
Generate a random integer in the range `[a, b]`.

This is an alias for `randrange(a, b + 1)`, included for compatibility with the builtin `random` module.

`RepeatableRandomSequence.randbytes` (*num\_bytes*) → `bytes`  
Generate a sequence of random bytes.

The index will only increment once, regardless of the number of bytes in the sequence.

This method produces similar results to

```
with rrs.cascade():
    result = bytes([rrs.randrange(256) for _ in range(num_bytes)])
```

but offers significantly-improved performance and does not discard excess random bits.

**Returns** A `bytes` object with *num\_bytes* random integers in `[0, 255]`.

`RepeatableRandomSequence.geometric` (*mean*: `float`, *include\_zero*: `bool = False`) → `int`  
Generate integers according to a geometric distribution.

If *include\_zero* is `False`, returns integers from 1 to infinity according to  $\Pr(x = k) = p(1 - p)^{k-1}$  where  $p = \frac{1}{\text{mean}}$ .

If *include\_zero* is `True`, returns integers from 0 to infinity according to  $\Pr(x = k) = p(1 - p)^k$  where  $p = \frac{1}{\text{mean}+1}$ .

### Parameters

- **mean** (`float`) – The desired mean value.
- **include\_zero** (`bool`) – Whether or not the distribution's support includes zero.

**Raises** **ValueError** – if *mean* is less than 1 if *include\_zero* is `False`, or less than 0 if *include\_zero* is `True`.

`RepeatableRandomSequence.finitegeometric` (*s*: `float`, *n*: `int`)

Generate a random integer according to a geometric-like distribution with exponent *s* and finite support `{1, ..., n}`.

The finite geometric distribution is defined by the equation

$$\Pr(x = k) = \frac{s^k}{\sum_{i=1}^N s^i}$$

**Raises** `ValueError` – if `n` is not at least 1

`RepeatableRandomSequence.zipfmandelbrot` (*s*: `float`, *q*: `float`, *n*: `int`)

Generate a random integer according to a Zipf-Mandelbrot distribution with exponent *s*, offset *q*, and support  $\{1, \dots, n\}$ .

The Zipf-Mandelbrot distribution is defined by the equation

$$\Pr(x = k) = \frac{(k + q)^{-s}}{\sum_{i=1}^N (i + q)^{-s}}$$

**Raises** `ValueError` – if `n` is not at least 1

## 1.3 Categorical distributions

`RepeatableRandomSequence.choice` (*sequence*: `Sequence`)

Choose a single random element from within a sequence.

**Raises** `IndexError` – if the sequence is empty.

`RepeatableRandomSequence.choices` (*population*: `Sequence`, *weights*: `Optional[Sequence[float]]` = `None`, *\**, *cum\_weights*: `Optional[Sequence[float]]` = `None`, *k*: `int` = 1) → `Sequence`

Choose *k* elements from a population, **with** replacement.

Either relative (via *weights*) or cumulative (via *cum\_weights*) weights may be specified. If no weights are specified, selections are made uniformly with equal probability.

---

**Tip:** Note that `k == 0` is a valid input, which returns an empty list and advances the sequence once even though no elements are chosen.

---

### Parameters

- **population** (*sequence*) – The population to sample from, which may include duplicate elements.
- **weights** (*sequence[float]*, *optional*) – Relative weights for each element in the population. Need not sum to 1.
- **cum\_weights** (*sequence[float]*, *optional*) – Cumulative weights for each element in the population, as calculated by something like `list(accumulate(weights))`.
- **k** (*int*) – The number of elements to choose.

### Raises

- `IndexError` – The population is empty.
- `ValueError` – The length of *weights* or *cum\_weights* does not match the population size.
- `TypeError` – Both *weights* and *cum\_weights* are specified.

`RepeatableRandomSequence.shuffle(sequence: Sequence) → None`  
Shuffle a sequence in place.

The index is only incremented once.

`RepeatableRandomSequence.sample(population, k: int) → Sequence`  
Choose  $k$  unique random elements from a population, **without** replacement.

Elements are returned in selection order, so all subsets of the returned list are valid samples.

If the population includes duplicate values, each occurrence is as distinct possible selection in the result.

---

**Tip:** Note that  $k == 0$  is a valid input, which returns an empty list and advances the sequence once even though no elements are chosen.

---

### Parameters

- **population** (*list*, *tuple*, *set*, *range*) – The source population.
- **k** (*int*) – The number of samples to choose, no more than the total number of elements in population.

### Raises

- **IndexError** – if population is empty.
- **ValueError** – if  $k$  is greater than the population size.

`RepeatableRandomSequence.chance(p: float) → bool`  
Returns True with probability  $p$ , else False.

Alias for `random() < p`.

## 1.4 Continuous distributions

`RepeatableRandomSequence.random() → float`  
Return a random float in  $[0.0, 1.0)$ .

`RepeatableRandomSequence.uniform(a: float, b: float) → float`  
Return a random float uniformly distributed in  $[a, b)$ .

`RepeatableRandomSequence.triangular(low: float = 0.0, high: float = 1.0, mode: Optional[float] = None) → float`  
Sample from a triangular distribution with lower limit  $low$ , upper limit  $high$ , and mode  $mode$ .

The triangular distribution is defined by

$$P(x) = \begin{cases} 0 & \text{for } x < l, \\ \frac{2(x-l)}{(h-l)(m-l)} & \text{for } l \leq x < m, \\ \frac{2}{h-l} & \text{for } x = m, \\ \frac{2(h-x)}{(h-l)(h-m)} & \text{for } m < x \leq h, \\ 0 & \text{for } x > h \end{cases}$$

**Raises** **ValueError** – if the mode is not in  $[low, high]$ .

RepeatableRandomSequence.**gauss** (*mu*: float, *sigma*: float) → float

Sample from a Gaussian distribution with parameters *mu* and *sigma*.

The Gaussian, or Normal distribution is defined by

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

**Tip:** If multiple normally-distributed values are required, consider using `gausspair()` for improved performance.

RepeatableRandomSequence.**gausspair** (*mu*: float, *sigma*: float) → Tuple[float, float]

Return a pair of *independent* samples from a Gaussian distribution with parameters *mu* and *sigma*.

This method produces similar results to

```
with rrs.cascade():
    result = rrs.gauss(mu, sigma), rrs.gauss(mu, sigma)
```

but offers improved performance.

RepeatableRandomSequence.**lognormvariate** (*mu*: float, *sigma*: float) → float

Sample from a log-normal distribution with parameters *mu* and *sigma*.

The logarithms of values sampled from a log-normal distribution are normally distributed with mean *mu* and standard deviation *sigma*. The distribution is defined by

$$P(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right)$$

RepeatableRandomSequence.**expovariate** (*lambd*: float) → float

Sample from an exponential distribution with rate *lambd*.

The probability density function for the exponential distribution is defined by  $P(x) = \lambda e^{-\lambda x}$  where  $x \geq 0$

RepeatableRandomSequence.**vonmisesvariate** (*mu*: float, *kappa*: float) → float

Sample from a von Mises distribution with parameters *mu* and *kappa*.

Samples from a von Mises distribution represent randomly-chosen angles clustered around a mean angle. This distribution is an approximation to the wrapped normal distribution and is defined by

$$P(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)}$$

where  $I_0(\kappa)$  is the modified Bessel function of order 0.

#### Parameters

- **mu** (float) – The mean angle, in radians, around which to cluster.
- **kappa** (float) – The concentration parameter, which must be at least 0.

**Returns** An angle in radians within  $[0, 2\pi)$

RepeatableRandomSequence.**gammavariate** (*alpha*: float, *beta*: float) → float

Sample from a gamma distribution with parameters *alpha* and *beta*. Not to be confused with `math.gamma()`!

The gamma distribution is defined as

$$P(x) = \frac{x^{\alpha-1} e^{-\frac{x}{\beta}}}{\Gamma(\alpha)\beta^\alpha}$$

**Caution:** This implementation defines its parameters to match `random.gammavariate()`. The parametrization differs from most common definitions of the gamma distribution, as defined on Wikipedia, et al. Take care when setting *alpha* and *beta*!

**Raises** `ValueError` – if either *alpha* or *beta* is not greater than 0.

`RepeatableRandomSequence.betavariate(alpha: float, beta: float) → float`  
Sample from a beta distribution with parameters *alpha* and *beta*.

The beta distribution is defined by

$$P(x) = x^{\alpha-1}(1-x)^{\beta-1} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}$$

**Returns** A random, beta-distributed value in [0.0, 1.0]

**Raises** `ValueError` – if either *alpha* or *beta* is not greater than 0.

`RepeatableRandomSequence.paretovariate(alpha: float) → float`  
Sample from a Pareto distribution with shape parameter *alpha* and minimum value 1.

The Pareto distribution has PDF

$$P(x) = \frac{\alpha}{x^{\alpha+1}}$$

**Raises** `ValueError` – if *alpha* is zero.

`RepeatableRandomSequence.weibullvariate(alpha: float, beta: float) → float`  
Sample from a Weibull distribution with scale parameter *alpha* and shape parameter *beta*.

The distribution is defined by

$$P(x) = \frac{\beta}{\alpha} \left(\frac{x}{\alpha}\right)^{\beta-1} e^{-(x/\alpha)^\beta}$$

where  $x \geq 0$ .

**Raises** `ValueError` – if *alpha* is zero.

`RepeatableRandomSequence.normalvariate(mu: float, sigma: float) → float`  
Sample from a normal distribution with mean *mu* and standard variation *sigma*.

---

**Note:** Unlike `random.normalvariate()`, this is an alias for `gauss(mu, sigma)`. It is included for compatibility with the built-in `random` module.

---

## 1.5 Examples

Generating random values:

```
import samplespace

rrs = samplespace.RepeatableRandomSequence(seed=1234)

samples = [rrs.randrange(30) for _ in range(10)]
print(samples)
# Will always print:
# [21, 13, 28, 19, 16, 29, 28, 24, 29, 25]
```

Distinct seeds produce unique results:

```
import samplespace

# Each seed will generate a unique sequence of values
for seed in range(5):
    rrs = samplespace.RepeatableRandomSequence(seed=seed)
    samples = [rrs.random() for _ in range(5)]
    print('Seed: {0}\tResults: {1}'.format(
        seed,
        ' '.join('{:.3f}'.format(x) for x in samples)))
```

Results depend only on number of previous calls, not their type:

```
import samplespace

rrs = samplespace.RepeatableRandomSequence(seed=1234)

dummy = rrs.random()
x1 = rrs.random()

rrs.reset()
dummy = rrs.gauss(6.0, 2.0)
x2 = rrs.random()
assert x2 == x1

rrs.reset()
dummy = rrs.randrange(50)
x3 = rrs.random()
assert x3 == x1

rrs.reset()
dummy = list('abcdefg')
rrs.shuffle(dummy)
x4 = rrs.random()
assert x4 == x1

rrs.reset()
with rrs.cascade():
    dummy = [rrs.random() for _ in range(10)]
x5 = rrs.random()
assert x5 == x1
```

Replay random sequences using `RepeatableRandomSequence.reset()`:

```
import samplespace

rrs = samplespace.RepeatableRandomSequence(seed=1234)

samples = [rrs.random() for _ in range(10)]
print(' '.join(samples))

# Using reset() returns the sequence to its initial state
rrs.reset()
samples2 = [rrs.random() for _ in range(10)]
print(' '.join(samples2))
assert samples == samples2
```

Replay      random      sequences      using      `RepeatableRandomSequence.getstate()`

*RepeatableRandomSequence.setstate():*

```
import samplespace

rrs = samplespace.RepeatableRandomSequence(seed=12345)

# Generate some random values to advance the state
[rrs.random() for _ in range(100)]

# Save the state for later recall
state = rrs.getstate()
print(rrs.random())
# Will print 0.2736967629462168

# Generate some more values
[rrs.random() for _ in range(100)]

# Return the sequence to the saved state. The next value will match
# the value following when the state was saved.
rrs.setstate(state)
print(rrs.random())
# Will also print 0.2736967629462168
```

Replay random sequences using *RepeatableRandomSequence.index()*:

```
import samplespace

rrs = samplespace.RepeatableRandomSequence(seed=5)

# Generate a sequence of values
samples = [rrs.randrange(10) for _ in range(15)]
print(samples)
# Will print
# [0, 2, 2, 7, 9, 4, 1, 5, 5, 6, 7, 1, 7, 6, 8]

# Rewind the sequence by 5
rrs.index -= 5

# Generate a new sequence, will overlap by 5 elements
samples = [rrs.randrange(10) for _ in range(15)]
print(samples)
# Will print
# [7, 1, 7, 6, 8, 6, 6, 6, 3, 7, 6, 9, 5, 2, 7]
```

Serialize sequence state as simple data types:

```
import samplespace
import samplespace.repeatablerandom
import json

rrs = samplespace.RepeatableRandomSequence(seed=12345)

# Generate some random values to advance the state
[rrs.random() for _ in range(100)]

# Save the state for later recall
# State can be serialized to a dict and serialized as JSON
state = rrs.getstate()
```

(continues on next page)



(continued from previous page)

```
state_as_dict = state.as_dict()
state_as_json = json.dumps(state_as_dict)
print(state_as_json)
# Prints {"seed": 12345, "hash_input": "gxzNfDj4Ypc=", "index": 100}

print(rrs.random())
# Will print 0.2736967629462168

# Generate some more values
[rrs.random() for _ in range(100)]

# Return the sequence to the saved state. The next value will match
# the value following when the state was saved.
new_state_as_dict = json.loads(state_as_json)
new_state = samplespace.repeatablerandom.RepeatableRandomSequenceState.from_dict(new_
↪state_as_dict)
rrs.setstate(new_state)
print(rrs.random())
# Will also print 0.2736967629462168
```



## SAMPLESPACE.DISTRIBUTIONS - SERIALIZABLE PROBABILITY DISTRIBUTIONS

---

This module implements a number of useful probability distributions.

Each distribution can be sampled using any random number generator providing at least the same functionality as the `random` module; this includes `samplespace.repeatablerandom.RepeatableRandomSequence`.

The classes in this module are primarily intended for storing information on random distributions in configuration files using `Distribution.as_dict()/distribution_from_dict()` or `Distribution.as_list()/distribution_from_list()`. See the *Examples* section for examples on how to do this.

### 2.1 Integer distributions

**class** `samplespace.distributions.DiscreteUniform` (*min\_val: int, max\_val: int*)

Represents a discrete uniform distribution of integers in [*min\_value*, *max\_value*).

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The ‘distribution’ key is the name of the distribution, and the remaining keys are the distribution’s kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property** `max_val`

Read-only property for the distribution’s upper limit.

**property** `min_val`

Read-only property for the distribution’s lower limit.

**sample** (*rand*) → int

Sample from the distribution.

**Parameters** `rand` – The random generator used to generate the sample.

**class** `samplespace.distributions.Geometric` (*mean: float, include\_zero: bool = False*)

Represents a geometric distribution.

If `include_zero` is `False`, returns integers from 1 to infinity according to  $\Pr(x = k) = p(1 - p)^{k-1}$  where  $p = \frac{1}{\text{mean}}$ .

If `include_zero` is `True`, returns integers from 0 to infinity according to  $\Pr(x = k) = p(1 - p)^k$  where  $p = \frac{1}{\text{mean} + 1}$ .

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The ‘distribution’ key is the name of the distribution, and the remaining keys are the distribution’s kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property include\_zero**

Read-only property for whether or not the distribution’s support includes zero.

**property mean**

Read-only property for the distribution’s mean.

**sample** (*rand*) → int

Sample from the distribution.

**Parameters** *rand* – The random generator used to generate the sample.

**class** `samplespace.distributions.FiniteGeometric` (*s*: float, *n*: int)

Represents a geometric-like distribution with exponent *s* and finite support  $\{1, \dots, n\}$ .

The finite geometric distribution is defined by the equation

$$\Pr(x = k) = \frac{s^k}{\sum_{i=1}^N s^i}$$

The distribution is defined such that each result is *s* times as likely to occur as the previous; i.e.  $\Pr(x = k) = s\Pr(x = k - 1)$  over the support.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The ‘distribution’ key is the name of the distribution, and the remaining keys are the distribution’s kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property n**

Read-only property for the number of values in the distribution’s support.

**property s**

Read-only property for the distribution’s *s* exponent.

**sample** (*rand*) → int

Sample from the distribution.

**Parameters** *rand* – The random generator used to generate the sample.

**class** `samplespace.distributions.ZipfMandelbrot` (*s*: float, *q*: float, *n*: int)

Represents a Zipf-Mandelbrot distribution with exponent *s*, offset *q*, and support  $\{1, \dots, n\}$ .

The Zipf-Mandelbrot distribution is defined by the equation

$$\Pr(x = k) = \frac{(k + q)^{-s}}{\sum_{i=1}^N (i + q)^{-s}}$$

When  $q == 0$ , the distribution becomes the Zipf distribution, and as  $n$  increases, it approaches the Zeta distribution.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The ‘distribution’ key is the name of the distribution, and the remaining keys are the distribution’s kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property n**

Read-only property for the number of values in the distribution’s support.

**property q**

Read-only property for the distribution’s  $q$  offset.

**property s**

Read-only property for the distribution’s  $s$  exponent.

**sample** (rand) → int

Sample from the distribution.

**Parameters** **rand** – The random generator used to generate the sample.

**class** samplespace.distributions.**Bernoulli** ( $p$ : float)

Represents a Bernoulli distribution with parameter  $p$ .

Returns True with probability  $p$ , and False otherwise.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The ‘distribution’ key is the name of the distribution, and the remaining keys are the distribution’s kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property p**

Read-only property for the distribution’s  $p$  parameter.

**sample** (rand) → bool

Sample from the distribution.

**Parameters** **rand** – The random generator used to generate the sample.

## 2.2 Categorical distributions

```
class samplespace.distributions.WeightedCategorical (items: Optional[Sequence[Tuple[float, Any]]] = None, population: Optional[Sequence] = None, weights: Optional[Sequence[float]] = None, *, cum_weights: Optional[Sequence[float]] = None)
```

Represents a categorical distribution defined by a population and a list of relative weights.

Either *items*, *population* and *weights*, or *population* and *cum\_weights* should be provided, not all three.

### Parameters

- **items** (*Sequence[Tuple[Any]]*) – A sequence of tuples in the format (weight, relative value).
- **population** (*Sequence*) – A sequence of possible values.
- **weights** (*Sequence[float]*, *optional*) – A sequence of relative weights corresponding to each item in the population. Must be the same length as the population list.
- **cum\_weights** (*Sequence[float]*, *optional*) – A sequence of cumulative weights corresponding to each item in the population. Must be the same length as the population list. Only one of *weights* and *cum\_weights* should be provided.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The ‘distribution’ key is the name of the distribution, and the remaining keys are the distribution’s kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property cum\_weights**

A read-only property for the distribution’s cumulative weights.

**property items**

A read-only property returning a sequence of tuples in the format (weight, relative value).

**property population**

A read-only property for the distribution’s population.

**sample** (*rand*)

Sample from the distribution.

**Parameters** **rand** – The random generator used to generate the sample.

```
class samplespace.distributions.UniformCategorical (population: Sequence)
```

Represents a uniform categorical distribution over a given population.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The ‘distribution’ key is the name of the distribution, and the remaining keys are the distribution’s kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property population**

A read-only property for the distribution's population.

**sample** (*rand*)

Sample from the distribution.

**Parameters** *rand* – The random generator used to generate the sample.

**class** `samplespace.distributions.FiniteGeometricCategorical` (*population: Sequence*,  
*s: float*)

Represents a categorical distribution with weights corresponding to a finite geometric-like distribution with exponent *s*.

The finite geometric distribution is defined by the equation

$$\Pr(x = k) = \frac{s^k}{\sum_{i=1}^N s^i}$$

The distribution is defined such that each result is *s* times as likely to occur as the previous; i.e.  $\Pr(x = k) = s\Pr(x = k - 1)$  over the support.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The 'distribution' key is the name of the distribution, and the remaining keys are the distribution's kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property cum\_weights**

A read-only property for the distribution's cumulative weights.

**property items**

A read-only property returning a sequence of tuples in the format (weight, relative value).

**property n**

Read-only property for the number of values in the distribution's support.

**property population**

A read-only property for the distribution's population.

**property s**

Read-only property for the distribution's *s* exponent.

**sample** (*rand*)

Sample from the distribution.

**Parameters** *rand* – The random generator used to generate the sample.

**class** `samplespace.distributions.ZipfMandelbrotCategorical` (*population: Sequence*,  
*s: float, q: float*)

Represents a categorical distribution with weights corresponding to a Zipf-Mandelbrot distribution with exponent *s* and offset *q*.

The Zipf-Mandelbrot distribution is defined by the equation

$$\Pr(x = k) = \frac{(k + q)^{-s}}{\sum_{i=1}^N (i + q)^{-s}}$$

When  $q == 0$ , the distribution becomes the Zipf distribution, and as  $n$  increases, it approaches the Zeta distribution.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The 'distribution' key is the name of the distribution, and the remaining keys are the distribution's kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property cum\_weights**

A read-only property for the distribution's cumulative weights.

**property items**

A read-only property returning a sequence of tuples in the format (weight, relative value).

**property n**

Read-only property for the number of values in the distribution's support.

**property population**

A read-only property for the distribution's population.

**property q**

Read-only property for the distribution's  $q$  offset.

**property s**

Read-only property for the distribution's  $s$  exponent.

**sample** (*rand*)

Sample from the distribution.

**Parameters** **rand** – The random generator used to generate the sample.

## 2.3 Continuous distributions

**class** `samplespace.distributions.Constant` (*value*)

Represents a distribution that always returns a constant value.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The 'distribution' key is the name of the distribution, and the remaining keys are the distribution's kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**sample** (*rand*)

Sample from the distribution.

**Parameters** **rand** – The random generator used to generate the sample.

**property value**

Read-only property for the distribution's constant value.



**class** `samplespace.distributions.Uniform` (*min\_val*: *float* = 0.0, *max\_val*: *float* = 1.0)

Represents a continuous uniform distribution with support [*min\_val*, *max\_val*).

The uniform distribution is defined as

$$P(x) = \begin{cases} \frac{1}{b-a} & \text{for } x \in [a, b) \\ 0 & \text{otherwise} \end{cases}$$

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The ‘distribution’ key is the name of the distribution, and the remaining keys are the distribution’s kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property max\_val**

Read-only property for the distribution’s upper limit.

**property min\_val**

Read-only property for the distribution’s lower limit.

**sample** (*rand*) → *float*

Sample from the distribution.

**Parameters** *rand* – The random generator used to generate the sample.

**class** `samplespace.distributions.Gamma` (*alpha*: *float*, *beta*: *float*)

Represents a gamma distribution with parameters *alpha* and *beta*.

The gamma distribution is defined as

$$P(x) = \frac{x^{\alpha-1} e^{-\frac{x}{\beta}}}{\Gamma(\alpha) \beta^{\alpha}}$$

**Caution:** This implementation defines its parameters to match `random.gammavariate()`. The parametrization differs from most common definitions of the gamma distribution, as defined on Wikipedia, et al. Take care when setting *alpha* and *beta*!

**property alpha**

Read-only property for the distribution’s *alpha* parameter.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The ‘distribution’ key is the name of the distribution, and the remaining keys are the distribution’s kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property beta**

Read-only property for the distribution’s *beta* parameter.

**sample** (*rand*) → *float*

Sample from the distribution.

**Parameters** *rand* – The random generator used to generate the sample.

**class** `samplespace.distributions.Triangular` (*low*: *float* = 0.0, *high*: *float* = 1.0, *mode*: *Optional[float]* = None)

Represents a triangular distribution with lower limit *low*, upper limit *high*, and mode *mode*.

The triangular distribution is defined by

$$P(x) = \begin{cases} 0 & \text{for } x < l, \\ \frac{2(x-l)}{(h-l)(m-l)} & \text{for } l \leq x < m, \\ \frac{2}{h-l} & \text{for } x = m, \\ \frac{2(h-x)}{(h-l)(h-m)} & \text{for } m \leq x < h, \\ 0 & \text{for } x \geq h \end{cases}$$

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The ‘distribution’ key is the name of the distribution, and the remaining keys are the distribution’s kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property high**

Read-only property for the distribution’s upper bound.

**property low**

Read-only property for the distribution’s lower bound.

**property mode**

Read-only property for the distribution’s mode, if specified, otherwise None.

**sample** (*rand*) → *float*

Sample from the distribution.

**Parameters** *rand* – The random generator used to generate the sample.

**class** `samplespace.distributions.LogNormal` (*mu*: *float* = 0.0, *sigma*: *float* = 1.0)

Represents a log-normal distribution with parameters *mu* and *sigma*.

The logarithms of values sampled from a log-normal distribution are normally distributed with mean *mu* and standard deviation *sigma*. The distribution is defined by

$$P(x) = \frac{1}{x\sigma\sqrt{2\pi}} \exp\left(-\frac{(\ln x - \mu)^2}{2\sigma^2}\right)$$

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The ‘distribution’ key is the name of the distribution, and the remaining keys are the distribution’s kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property mu**

Read-only property for the distribution's *mu* parameter.

**sample** (*rand*) → float

Sample from the distribution.

**Parameters** *rand* – The random generator used to generate the sample.

**property sigma**

Read-only property for the distribution's *sigma* parameter.

**class** samplespace.distributions.**Exponential** (*lambda*: float)

Represents an exponential distribution with rate *lambda*.

The probability density function for the exponential distribution is defined by  $P(x) = \lambda e^{-\lambda x}$  where  $x \geq 0$

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The 'distribution' key is the name of the distribution, and the remaining keys are the distribution's kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property lambda**

Read-only property for the distribution's rate parameter.

**sample** (*rand*) → float

Sample from the distribution.

**Parameters** *rand* – The random generator used to generate the sample.

**class** samplespace.distributions.**VonMises** (*mu*: float, *kappa*: float)

Represents a von Mises distribution with parameters *mu* and *kappa*.

Samples from a von Mises distribution represent randomly-chosen angles clustered around a mean angle. This distribution is an approximation to the wrapped normal distribution and is defined by

$$P(x) = \frac{e^{\kappa \cos(x-\mu)}}{2\pi I_0(\kappa)}$$

where  $I_0(\kappa)$  is the modified Bessel function of order 0.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The 'distribution' key is the name of the distribution, and the remaining keys are the distribution's kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property kappa**

Read-only property for the distribution's *kappa* parameter.

**property mu**

Read-only property for the distribution's *mu* parameter.

**sample** (*rand*) → float

Sample from the distribution.

**Parameters** **rand** – The random generator used to generate the sample.

**class** `samplespace.distributions.Beta` (*alpha*: *float*, *beta*: *float*)

Represents a beta distribution with parameters *alpha* and *beta*.

The beta distribution is defined by

$$P(x) = x^{\alpha-1}(1-x)^{\beta-1} \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)}$$

**property** **alpha**

Read-only property for the distribution's *alpha* parameter.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The 'distribution' key is the name of the distribution, and the remaining keys are the distribution's kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property** **beta**

Read-only property for the distribution's *beta* parameter.

**sample** (*rand*) → *float*

Sample from the distribution.

**Parameters** **rand** – The random generator used to generate the sample.

**class** `samplespace.distributions.Pareto` (*alpha*: *float*)

Represents a Pareto distribution with shape parameter *alpha* and minimum value 1.

The Pareto distribution has PDF

$$P(x) = \frac{\alpha}{x^{\alpha+1}}$$

**property** **alpha**

Read-only property for the distribution's *alpha* parameter.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The 'distribution' key is the name of the distribution, and the remaining keys are the distribution's kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**sample** (*rand*) → *float*

Sample from the distribution.

**Parameters** **rand** – The random generator used to generate the sample.

**class** `samplespace.distributions.Weibull` (*alpha*: *float*, *beta*: *float*)

Represents a Weibull distribution with scale parameter *alpha* and shape parameter *beta*.

The distribution is defined by

$$P(x) = \frac{\beta}{\alpha} \left(\frac{x}{\alpha}\right)^{\beta-1} e^{-(x/\alpha)^\beta}$$

where  $x \geq 0$ .

**property alpha**

Read-only property for the distribution's *alpha* parameter.

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The 'distribution' key is the name of the distribution, and the remaining keys are the distribution's kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property beta**

Read-only property for the distribution's *beta* parameter.

**sample** (*rand*) → float

Sample from the distribution.

**Parameters** *rand* – The random generator used to generate the sample.

**class** samplespace.distributions.**Gaussian** (*mu*: float = 0.0, *sigma*: float = 1.0)

Represents a Gaussian distribution with parameters *mu* and *sigma*.

The Gaussian, or Normal distribution is defined by

$$P(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

**as\_dict** () → Dict

Return a representation of the distribution as a dict.

The 'distribution' key is the name of the distribution, and the remaining keys are the distribution's kwargs.

**as\_list** () → List

Return a representation of the distribution as a list.

The first element if the list is the name of the distribution, and the subsequent elements are ordered parameters.

**property mu**

Read-only property for the distribution's *mu* parameter.

**sample** (*rand*)

Sample from the distribution.

**Parameters** *rand* – The random generator used to generate the sample.

**property sigma**

Read-only property for the distribution's *sigma* parameter.

## 2.4 Serialization functions

`samplespace.distributions.distribution_from_list(as_list)`

Build a distribution using a list of parameters.

Expects a list of values in the same form return by a call to `Distribution.as_list()`; i.e. `['name', arg0, arg1, ...]`.

### Examples

```
>>> tri = Triangular(2.0, 5.0)
>>> tri_as_list = tri.as_list()
>>> tri_as_list
['triangular', 2.0, 5.0]
>>> new_tri = distribution_from_list(tri_as_list)
>>> assert tri == new_tri
```

### Raises

- **KeyError** – if the distribution name is not recognized.
- **TypeError** – if the wrong number of arguments is given.

`samplespace.distributions.distribution_from_dict(as_dict)`

Build a distribution using a dictionary of keyword arguments.

Expects a dict in the same form as returned by a call to `Distribution.as_dict()`; i.e. `{'distribution': 'name', 'arg0': 'value', 'arg1': 'value', ...}`.

### Examples

```
>>> gauss = Gaussian(0.8, 5.0)
>>> gauss_as_dict = gauss.as_dict()
>>> gauss_as_dict
{'distribution': 'gaussian', 'mu': 0.8, 'sigma': 5.0}
>>> new_gauss = distribution_from_dict(gauss_as_dict)
>>> assert gauss == new_gauss
```

### Raises

- **KeyError** – if the distribution name is not recognized or provided.
- **TypeError** – if the wrong keyword arguments are provided.

**classmethod** `samplespace.distribution.Distribution.from_list()`

An alias for `distribution_from_list()`.

**classmethod** `samplespace.distribution.Distribution.from_dict()`

An alias for `distribution_from_dict()`.

## 2.5 Examples

Sampling from a distribution:

```
import random
import statistics

from samplespace.distributions import Gaussian, FiniteGeometric

gauss = Gaussian(15.0, 2.0)
samples = [gauss.sample(random) for _ in range(100)]
print('Mean:', statistics.mean(samples))
print('Standard deviation:', statistics.stdev(samples))

geo = FiniteGeometric(['one', 'two', 'three', 'four', 'five'], 0.7)
samples = [geo.sample(random) for _ in range(10)]
print(' '.join(samples))
```

Using other random generators:

```
from samplespace import distributions, RepeatableRandomSequence

exponential = distributions.Exponential(0.8)

rrs = RepeatableRandomSequence(seed=12345)
print([exponential.sample(rrs) for _ in range(5)])
# Will always print:
# [1.1959827296976795, 0.6056492468915003, 0.9155454941988664, 0.5653478889068511, 0.
↪ 6500080335986231]
```

Representations of distributions:

```
from samplespace.distributions import Pareto, DiscreteUniform, UniformCategorical

pareto = Pareto(2.5)
print('Pareto as dict:', pareto.as_dict()) # {'distribution': 'pareto', 'alpha': 2.5}
print('Pareto as list:', pareto.as_list()) # ['pareto', 2.5]

discrete = DiscreteUniform(3, 8)
print('Discrete uniform as dict:', discrete.as_dict()) # {'distribution':
↪ 'discreteuniform', 'min_val': 3, 'max_val': 8}
print('Discrete uniform as list:', discrete.as_list()) # ['discreteuniform', 3, 8]

cat = UniformCategorical(['string', 4, {'a': 'dict'}])
print('Uniform categorical as dict:', cat.as_dict()) # {'distribution':
↪ 'uniformcategorical', 'population': ['string', 4, {'a': 'dict'}]}
print('Uniform categorical as list:', cat.as_list()) # ['uniformcategorical', [
↪ 'string', 4, {'a': 'dict'}]]
```

Storing distributions as in config files as lists:

```
import random
from samplespace import distributions

...

skeleton_config = {
```

(continues on next page)

(continued from previous page)

```

    'name': 'Skeleton',
    'starting_hp': ['gaussian', 50.0, 5.0],
    'coins_dropped': ['geometric', 0.8, True],
}

...

class Skeleton(object):
    def __init__(self, name, starting_hp, coins_dropped_dist):
        self.name = name
        self.starting_hp = starting_hp
        self.coins_dropped_dist = coins_dropped_dist

    def drop_coins(self):
        return self.coins_dropped_dist.sample(random)

...

class SkeletonFactory(object):

    def __init__(self, config):
        self.name = config['name']
        self.starting_hp_dist = distributions.distribution_from_list(config['starting_
↪hp'])
        self.coins_dropped_dist = distributions.distribution_from_list(config['coins_
↪dropped'])

    def make_skeleton(self):
        return Skeleton(
            self.name,
            int(self.starting_hp_dist.sample(random)),
            self.coins_dropped_dist)

```

Storing distributions in config files as dictionaries:

```

from samplespace import distributions, RepeatableRandomSequence

city_config = {
    "building_distribution": {
        "distribution": "weightedcategorical",
        "items": [
            ["house", 0.2],
            ["store", 0.4],
            ["tree", 0.8],
            ["ground", 5.0]
        ]
    }
}

rrs = RepeatableRandomSequence()
building_dist = distributions.distribution_from_dict(city_config['building_
↪distribution'])

buildings = [[building_dist.sample(rrs) for col in range(20)] for row in range(5)]

for row in buildings:
    for building_type in row:

```

(continues on next page)



(continued from previous page)

```
if building_type == 'house':  
    print('H', end='')  
elif building_type == 'store':  
    print('S', end='')  
elif building_type == 'tree':  
    print('T', end='')  
else:  
    print('.', end='')  
print()
```



## SAMPLESPACE.ALGORITHMS - GENERAL SAMPLING ALGORITHMS

---

This module implements several general-purpose sampling algorithms.

---

```
sample.space.algorithms.sample_discrete_roulette (randfloat: Callable[[], float],  
                                                  cum_weights: Sequence[float])  
                                                  → int
```

Sample from a given discrete distribution given its cumulative weights, using binary search / the roulette wheel selection algorithm.

### Parameters

- **randfloat** (*Callable[[], float]*) – A function returning a random float in [0.0, 1.0), e.g. `random.random()`. Will only be called once.
- **cum\_weights** (*Sequence[float]*) – The list of cumulative weights. These do not need to be normalized, so `cum_weights[-1]` does not necessarily have to be 1.0.

**Returns** An index into the population from 0 to `len(cum_weights) - 1`.

```
class sample.space.algorithms.AliasTable (probability: Sequence[float], alias: Sequence[int])
```

Sample from a given discrete distribution given its relative weights, using the alias table algorithm.

Alias tables are slow to construct, but offer increased sampling speed compared to the roulette wheel algorithm.

Construct a table using `from_weights()`.

### probability

The probability row of the table

**Type** List[float]

### alias

The alias row of the table

**Type** List[int]

```
classmethod from_weights (weights: Sequence[float])
```

Construct an alias table using a list of relative weights.

The provided weights need not sum to 1.0.

```
sample (randbelow: Callable[[int], int], randchance: Callable[[float], bool]) → int
```

Sample from the alias table.

### Parameters

- **randbelow** (*Callable*[[*int*], *int*]) – A function returning a random int in [0, *arg*), e.g. `random.randrange()`. Will only be called once per sample.
- **randchance** (*Callable*[[*float*], *bool*]) – A function returning `True` with chance *arg* and `False` otherwise, e.g. `lambda arg: random.random() < arg`. Will only be called once per sample.

**Returns** An index into the population from 0 to `len(weights) - 1`.

## 3.1 Examples

Sampling from a discrete categorical distribution using the Roulette Wheel Selection algorithm:

```
import random
import itertools
from samplespace import algorithms

population = 'abcde'
weights = [0.1, 0.4, 0.2, 0.3, 0.5]

cum_weights = list(itertools.accumulate(weights))
indices = [algorithms.sample_discrete_roulette(random.random, cum_weights)
           for _ in range(25)] # [0, 4, 4, ...]
samples = [population[index] for index in indices] # ['a', 'e', 'e', ...]
```

Sampling from a discrete categorical distribution using the Alias Table algorithm and a repeatable sequence:

```
from samplespace import algorithms, RepeatableRandomSequence

population = 'abcde'
weights = [0.1, 0.4, 0.2, 0.3, 0.5]

rrs = RepeatableRandomSequence(seed=12345)
at = algorithms.AliasTable.from_weights(weights)
indices = [at.sample(rrs.randrange, rrs.chance) for _ in range(25)]
samples = [population[index] for index in indices] # ['e', 'e', 'c', ...]

# Note that rrs.index == 50 at this point, since two random values
# were required for each sample.

# It is also possible to use AliasTable within a cascade:
rrs.reset()
indices = [0] * 25
for i in range(len(indices)):
    with rrs.cascade():
        indices[i] = at.sample(rrs.randrange, rrs.chance)
samples = [population[index] for index in indices] # ['e', 'd', 'b', ...]

# Because cascades were used, rrs.index == 25.
# Note that the values produced are different than the previous
# attempt, because a different number of logical samples were made.
```

## SAMPLESPACE.PYYAML\_SUPPORT - YAML SERIALIZATION SUPPORT

---

When enabled, this module provides YAML serialization support for classes in `samplespace.repeatablerandom`, `samplespace.distributions`, and `samplespace.algorithms`.

The `yaml` module provided by `PyYaml` is required to use this module, and an exception will be thrown if it is not available.

---

**Tip:** Once the module is imported, `enable_yaml_support()` must be called to enable YAML support.

---

`samplespace.pyyaml_support.enable_yaml_support(_force=False)`  
Add YAML serialization support. This function only needs to be called once per application.

### 4.1 Examples

Repeatable random sequence:

```
>>> import yaml
>>> from samplespace import RepeatableRandomSequence
>>> import samplespace.pyyaml_support
>>> samplespace.pyyaml_support.enable_yaml_support()
>>>
>>> rrs = RepeatableRandomSequence(seed=678)
>>> [rrs.randrange(10) for _ in range(5)]
[7, 7, 0, 8, 3]
>>>
>>> # Serialize the sequence as YAML
...
>>> as_yaml = yaml.dump(rrs)
>>> as_yaml
'!samplespace.rrs\nhash_input: slenBV+SSXk=\nindex: 5\nseed: 678\n'
>>>
>>> # Generate some random values to compare against later
...
>>> [rrs.randrange(10) for _ in range(5)]
[0, 5, 1, 3, 9]
>>> [rrs.randrange(10) for _ in range(5)]
[7, 1, 1, 0, 5]
>>> [rrs.randrange(10) for _ in range(5)]
```

(continues on next page)

(continued from previous page)

```

[2, 9, 1, 4, 8]
>>>
>>> # Restore the saved sequence
...
>>> rrs2 = yaml.load(as_yaml, Loader=yaml.FullLoader)
>>>
>>> # Verify that values match those at time of serialization
...
>>> [rrs2.randrange(10) for _ in range(5)]
[0, 5, 1, 3, 9]
>>> [rrs2.randrange(10) for _ in range(5)]
[7, 1, 1, 0, 5]
>>> [rrs2.randrange(10) for _ in range(5)]
[2, 9, 1, 4, 8]

```

## Distributions:

```

>>> import yaml
>>> from samplespace import distributions
>>> import samplespace.pyyaml_support
>>> samplespace.pyyaml_support.enable_yaml_support()
>>>
>>> gamma = distributions.Gamma(5.0, 3.0)
>>> gamma_as_yaml = yaml.dump(gamma)
>>> print(gamma_as_yaml)
!samplespace.distribution
alpha: 5.0
beta: 3.0
distribution: gamma
>>> assert yaml.load(gamma_as_yaml, Loader=yaml.FullLoader) == gamma
>>>
>>> zipf = distributions.Zipf(0.9, 10)
>>> zipf_as_yaml = yaml.dump(zipf)
>>> print(zipf_as_yaml)
!samplespace.distribution
distribution: zipf
n: 10
s: 0.9
>>> assert yaml.load(zipf_as_yaml, Loader=yaml.FullLoader) == zipf
>>>
>>> wcat = distributions.WeightedCategorical(
...     population='abcde',
...     cum_weights=[0.1, 0.4, 0.6, 0.7, 1.1])
>>> wcat_as_yaml = yaml.dump(wcat)
>>> print(wcat_as_yaml)
!samplespace.distribution
distribution: weightedcategorical
items:
- - a
  - 0.1
- - b
  - 0.30000000000000004
- - c
  - 0.19999999999999996
- - d
  - 0.09999999999999998
- - e

```

(continues on next page)

(continued from previous page)

```

- 0.400000000000000013
>>> assert yaml.load(wcat_as_yaml, Loader=yaml.FullLoader) == wcat

```

Algorithms:

```

>>> import yaml
>>> from samplespace.algorithms import AliasTable
>>> import samplespace.pyyaml_support
>>> samplespace.pyyaml_support.enable_yaml_support()
>>>
>>> at = AliasTable.from_weights([0.1, 0.3, 0.5, 0.2, 0.6, 0.7])
>>> at.alias
[4, 5, 0, 5, 2, 4]
>>> at.probability
[0.25, 0.7499999999999998, 1.0, 0.5, 0.7499999999999991, 0.9999999999999996]
>>>
>>> at_as_yaml = yaml.dump(at)
>>> print(at_as_yaml)
!samplespace.aliastable
alias:
- 4
- 5
- 0
- 5
- 2
- 4
probability:
- 0.25
- 0.7499999999999998
- 1.0
- 0.5
- 0.7499999999999991
- 0.9999999999999996
>>> at2 = yaml.load(at_as_yaml, Loader=yaml.FullLoader)
>>> assert at2.alias == at.alias
>>> assert at2.probability == at.probability

```





## RANDOM GENERATOR QUALITY

`RepeatableRandomSequence` produces high-quality psuedo-random data regardless of seed or cascade size.

The script `tests/generate_random_bytes.py` can be used to generate files filled with random bytes for testing with external testing programs, or `tests/stream_random_bytes.py [seed]` can be used to stream an unlimited number of random bytes to stdout.

### 5.1 Randomness Test Results

Results from ENT:

```
$ python stream_random_bytes.py | head -c 8388608 | ent
Entropy = 7.999979 bits per byte.

Optimum compression would reduce the size
of this 8388608 byte file by 0 percent.

Chi square distribution for 8388608 samples is 244.76, and randomly
would exceed this value 66.64 percent of the times.

Arithmetic mean value of data bytes is 127.4945 (127.5 = random).
Monte Carlo value for Pi is 3.141415391 (error 0.01 percent).
Serial correlation coefficient is 0.000034 (totally uncorrelated = 0.0).
```

Results from Dieharder:

```
#####
#               dieharder version 3.31.1 Copyright 2003 Robert G. Brown               #
#####
test_name  |ntup| tsamples|psamples|  p-value |Assessment
#####
diehard_birthdays| 0|    100|    100|0.46260921| PASSED
diehard_operm5| 0| 1000000|    100|0.30499458| PASSED
diehard_rank_32x32| 0|   40000|    100|0.84557704| PASSED
diehard_rank_6x8| 0|   100000|    100|0.75986634| PASSED
diehard_bitstream| 0| 2097152|    100|0.26336497| PASSED
diehard_opso| 0| 2097152|    100|0.09991336| PASSED
diehard_oqso| 0| 2097152|    100|0.47106073| PASSED
diehard_dna| 0| 2097152|    100|0.09032458| PASSED
diehard_count_1s_str| 0|   256000|    100|0.93473326| PASSED
diehard_count_1s_byt| 0|   256000|    100|0.84493493| PASSED
diehard_parking_lot| 0|   12000|    100|0.33164281| PASSED
diehard_2dsphere| 2|    8000|    100|0.90268828| PASSED
```

(continues on next page)

(continued from previous page)

diehard_3dsphere	3	4000	100 0.87557333	PASSED
diehard_squeeze	0	100000	100 0.75415145	PASSED
diehard_sums	0	100	100 0.15694874	PASSED
diehard_runs	0	100000	100 0.19158234	PASSED
diehard_runs	0	100000	100 0.27969502	PASSED
diehard_craps	0	200000	100 0.99326664	PASSED
diehard_craps	0	200000	100 0.37802538	PASSED
marsaglia_tsang_gcd	0	40000	100 0.44158442	PASSED
marsaglia_tsang_gcd	0	40000	100 0.38455344	PASSED
sts_monobit	1	100000	100 0.90249899	PASSED
sts_runs	2	100000	100 0.77696296	PASSED
sts_serial	1	100000	100 0.90249899	PASSED
sts_serial	2	100000	100 0.59050254	PASSED
sts_serial	3	100000	100 0.90871975	PASSED
sts_serial	3	100000	100 0.99336098	PASSED
sts_serial	4	100000	100 0.34637953	PASSED
sts_serial	4	100000	100 0.23240191	PASSED
sts_serial	5	100000	100 0.65698384	PASSED
sts_serial	5	100000	100 0.20441284	PASSED
sts_serial	6	100000	100 0.11455355	PASSED
sts_serial	6	100000	100 0.10947486	PASSED
sts_serial	7	100000	100 0.33236940	PASSED
sts_serial	7	100000	100 0.90065107	PASSED
sts_serial	8	100000	100 0.71691528	PASSED
sts_serial	8	100000	100 0.72658614	PASSED
sts_serial	9	100000	100 0.05947803	PASSED
sts_serial	9	100000	100 0.35266839	PASSED
sts_serial	10	100000	100 0.25645609	PASSED
sts_serial	10	100000	100 0.96328452	PASSED
sts_serial	11	100000	100 0.26614777	PASSED
sts_serial	11	100000	100 0.75690153	PASSED
sts_serial	12	100000	100 0.72460595	PASSED
sts_serial	12	100000	100 0.59228401	PASSED
sts_serial	13	100000	100 0.43108810	PASSED
sts_serial	13	100000	100 0.39814551	PASSED
sts_serial	14	100000	100 0.93222323	PASSED
sts_serial	14	100000	100 0.10847133	PASSED
sts_serial	15	100000	100 0.97030212	PASSED
sts_serial	15	100000	100 0.74232610	PASSED
sts_serial	16	100000	100 0.30293298	PASSED
sts_serial	16	100000	100 0.74289769	PASSED
rgb_bitdist	1	100000	100 0.52609114	PASSED
rgb_bitdist	2	100000	100 0.35992730	PASSED
rgb_bitdist	3	100000	100 0.80654719	PASSED
rgb_bitdist	4	100000	100 0.84084274	PASSED
rgb_bitdist	5	100000	100 0.54266728	PASSED
rgb_bitdist	6	100000	100 0.04443687	PASSED
rgb_bitdist	7	100000	100 0.22723549	PASSED
rgb_minimum_distance	4	10000	1000 0.23472614	PASSED
rgb_permutations	5	100000	100 0.68681017	PASSED
rgb_lagged_sum	0	1000000	100 0.57837513	PASSED
rgb_kstest_test	0	10000	1000 0.52944830	PASSED

## 5.2 Distribution Validity Tests

The scripts `tests/scripts/calculate_*_metrics.py` can be used to check that Repeatable Random Sequences produce values with the correct probability distributions.

Bear in mind that as limited samples are taken from each distribution, some false negatives may occur. If anomalous results are present, try re-running the scripts.



## PYTHON MODULE INDEX

### S

`samplespace.algorithms`, [31](#)  
`samplespace.distributions`, [15](#)  
`samplespace.pyyaml_support`, [33](#)  
`samplespace.repeatablerandom`, [3](#)



## A

- `alias` (*samplespace.algorithms.AliasTable* attribute), 31
- `AliasTable` (class in *samplespace.algorithms*), 31
- `alpha()` (*samplespace.distributions.Beta* property), 24
- `alpha()` (*samplespace.distributions.Gamma* property), 21
- `alpha()` (*samplespace.distributions.Pareto* property), 24
- `alpha()` (*samplespace.distributions.Weibull* property), 25
- `as_dict()` (*samplespace.distributions.Bernoulli* method), 17
- `as_dict()` (*samplespace.distributions.Beta* method), 24
- `as_dict()` (*samplespace.distributions.Constant* method), 20
- `as_dict()` (*samplespace.distributions.DiscreteUniform* method), 15
- `as_dict()` (*samplespace.distributions.Exponential* method), 23
- `as_dict()` (*samplespace.distributions.FiniteGeometric* method), 16
- `as_dict()` (*samplespace.distributions.FiniteGeometricCategorical* method), 19
- `as_dict()` (*samplespace.distributions.Gamma* method), 21
- `as_dict()` (*samplespace.distributions.Gaussian* method), 25
- `as_dict()` (*samplespace.distributions.Geometric* method), 16
- `as_dict()` (*samplespace.distributions.LogNormal* method), 22
- `as_dict()` (*samplespace.distributions.Pareto* method), 24
- `as_dict()` (*samplespace.distributions.Triangular* method), 22
- `as_dict()` (*samplespace.distributions.Uniform* method), 21
- `as_dict()` (*samplespace.distributions.UniformCategorical* method), 18
- `as_dict()` (*samplespace.distributions.VonMises* method), 23
- `as_dict()` (*samplespace.distributions.Weibull* method), 25
- `as_dict()` (*samplespace.distributions.WeightedCategorical* method), 18
- `as_dict()` (*samplespace.distributions.ZipfMandelbrot* method), 17
- `as_dict()` (*samplespace.distributions.ZipfMandelbrotCategorical* method), 20
- `as_dict()` (*samplespace.repeatablerandom.RepeatableRandomSequence* method), 5
- `as_list()` (*samplespace.distributions.Bernoulli* method), 17
- `as_list()` (*samplespace.distributions.Beta* method), 24
- `as_list()` (*samplespace.distributions.Constant* method), 20
- `as_list()` (*samplespace.distributions.DiscreteUniform* method), 15
- `as_list()` (*samplespace.distributions.Exponential* method), 23
- `as_list()` (*samplespace.distributions.FiniteGeometric* method), 16
- `as_list()` (*samplespace.distributions.FiniteGeometricCategorical* method), 19
- `as_list()` (*samplespace.distributions.Gamma* method), 21
- `as_list()` (*samplespace.distributions.Gaussian* method), 25
- `as_list()` (*samplespace.distributions.Geometric* method), 16
- `as_list()` (*samplespace.distributions.LogNormal* method), 22
- `as_list()` (*samplespace.distributions.Pareto* method), 24
- `as_list()` (*samplespace.distributions.Triangular* method), 22
- `as_list()` (*samplespace.distributions.Uniform* method), 21
- `as_list()` (*samplespace.distributions.UniformCategorical* method), 18
- `as_list()` (*samplespace.distributions.VonMises* method), 23

method), 23

as\_list() (samplespace.distributions.Weibull method), 25

as\_list() (samplespace.distributions.WeightedCategorical method), 18

as\_list() (samplespace.distributions.ZipfMandelbrot method), 17

as\_list() (samplespace.distributions.ZipfMandelbrotCategorical method), 20

## B

Bernoulli (class in samplespace.distributions), 17

Beta (class in samplespace.distributions), 24

beta() (samplespace.distributions.Beta property), 24

beta() (samplespace.distributions.Gamma property), 21

beta() (samplespace.distributions.Weibull property), 25

betavariate() (samplespace.repeatablerandom.RepeatableRandomSequence method), 10

BLOCK\_MASK (samplespace.repeatablerandom.RepeatableRandomSequence attribute), 3

BLOCK\_SIZE\_BITS (samplespace.repeatablerandom.RepeatableRandomSequence attribute), 3

## C

cascade() (samplespace.repeatablerandom.RepeatableRandomSequence method), 4

chance() (samplespace.repeatablerandom.RepeatableRandomSequence method), 8

choice() (samplespace.repeatablerandom.RepeatableRandomSequence method), 7

choices() (samplespace.repeatablerandom.RepeatableRandomSequence method), 7

Constant (class in samplespace.distributions), 20

cum\_weights() (samplespace.distributions.FiniteGeometricCategorical property), 19

cum\_weights() (samplespace.distributions.WeightedCategorical property), 18

cum\_weights() (samplespace.distributions.ZipfMandelbrotCategorical property), 20

## D

DiscreteUniform (class in samplespace.distributions), 15

distribution\_from\_dict() (in module samplespace.distributions), 26

distribution\_from\_list() (in module samplespace.distributions), 26

## E

enable\_yaml\_support() (in module samplespace.pyyaml\_support), 33

Exponential (class in samplespace.distributions), 23

expovariate() (samplespace.repeatablerandom.RepeatableRandomSequence method), 9

## F

FiniteGeometric (class in samplespace.distributions), 16

finitegeometric() (samplespace.repeatablerandom.RepeatableRandomSequence method), 6

FiniteGeometricCategorical (class in samplespace.distributions), 19

from\_dict() (samplespace.distributions.samplespace.distribution.Distribution class method), 26

from\_dict() (samplespace.repeatablerandom.RepeatableRandomSequence class method), 5

from\_list() (samplespace.distributions.samplespace.distribution.Distribution class method), 26

from\_weights() (samplespace.algorithms.AliasTable class method), 31

## G

Gamma (class in samplespace.distributions), 21

gammaavariate() (samplespace.repeatablerandom.RepeatableRandomSequence method), 9

gauss() (samplespace.repeatablerandom.RepeatableRandomSequence method), 8

Gaussian (class in samplespace.distributions), 25

gausspair() (samplespace.repeatablerandom.RepeatableRandomSequence method), 9

Geometric (class in samplespace.distributions), 15

geometric() (samplespace.repeatablerandom.RepeatableRandomSequence method), 6

getnextblock() (samplespace.repeatablerandom.RepeatableRandomSequence method), 4

getrandbits() (samplespace.repeatablerandom.RepeatableRandomSequence method), 4

getseed() (samplespace.repeatablerandom.RepeatableRandomSequence method), 3

getstate() (samplespace.repeatablerandom.RepeatableRandomSequence method), 4

## H

high() (samplespace.distributions.Triangular property), 22



## I

include\_zero() (samplespace.distributions.Geometric property), 16  
 index() (samplespace.repeatablerandom.RepeatableRandomSequence attribute), 4  
 items() (samplespace.distributions.FiniteGeometricCategorical property), 19  
 items() (samplespace.distributions.WeightedCategorical property), 18  
 items() (samplespace.distributions.ZipfMandelbrotCategorical property), 20

## K

kappa() (samplespace.distributions.VonMises property), 23

## L

lambd() (samplespace.distributions.Exponential property), 23  
 LogNormal (class in samplespace.distributions), 22  
 lognormvariate() (samplespace.repeatablerandom.RepeatableRandomSequence method), 9  
 low() (samplespace.distributions.Triangular property), 22

## M

max\_val() (samplespace.distributions.DiscreteUniform property), 15  
 max\_val() (samplespace.distributions.Uniform property), 21  
 mean() (samplespace.distributions.Geometric property), 16  
 min\_val() (samplespace.distributions.DiscreteUniform property), 15  
 min\_val() (samplespace.distributions.Uniform property), 21  
 mode() (samplespace.distributions.Triangular property), 22  
 module  
   samplespace.algorithms, 31  
   samplespace.distributions, 15  
   samplespace.pyyaml\_support, 33  
   samplespace.repeatablerandom, 3  
 mu() (samplespace.distributions.Gaussian property), 25  
 mu() (samplespace.distributions.LogNormal property), 23  
 mu() (samplespace.distributions.VonMises property), 23

## N

n() (samplespace.distributions.FiniteGeometric property), 16

n() (samplespace.distributions.FiniteGeometricCategorical property), 19  
 n() (samplespace.distributions.ZipfMandelbrot property), 17  
 n() (samplespace.distributions.ZipfMandelbrotCategorical property), 20  
 normalvariate() (samplespace.repeatablerandom.RepeatableRandomSequence method), 10

## P

p() (samplespace.distributions.Bernoulli property), 17  
 Pareto (class in samplespace.distributions), 24  
 paretovariate() (samplespace.repeatablerandom.RepeatableRandomSequence method), 10  
 population() (samplespace.distributions.FiniteGeometricCategorical property), 19  
 population() (samplespace.distributions.UniformCategorical property), 19  
 population() (samplespace.distributions.WeightedCategorical property), 18  
 population() (samplespace.distributions.ZipfMandelbrotCategorical property), 20  
 probability (samplespace.algorithms.AliasTable attribute), 31

## Q

q() (samplespace.distributions.ZipfMandelbrot property), 17  
 q() (samplespace.distributions.ZipfMandelbrotCategorical property), 20

## R

randbytes() (samplespace.repeatablerandom.RepeatableRandomSequence method), 6  
 randint() (samplespace.repeatablerandom.RepeatableRandomSequence method), 6  
 random() (samplespace.repeatablerandom.RepeatableRandomSequence method), 8  
 randrange() (samplespace.repeatablerandom.RepeatableRandomSequence method), 6  
 RepeatableRandomSequence (class in samplespace.repeatablerandom), 3  
 RepeatableRandomSequenceState (class in samplespace.repeatablerandom), 5  
 reset() (samplespace.repeatablerandom.RepeatableRandomSequence method), 4

## S

`s()` (*samplespace.distributions.FiniteGeometric* property), 16  
`s()` (*samplespace.distributions.FiniteGeometricCategorical* property), 19  
`s()` (*samplespace.distributions.ZipfMandelbrot* property), 17  
`s()` (*samplespace.distributions.ZipfMandelbrotCategorical* property), 20  
`sample()` (*samplespace.algorithms.AliasTable* method), 31  
`sample()` (*samplespace.distributions.Bernoulli* method), 17  
`sample()` (*samplespace.distributions.Beta* method), 24  
`sample()` (*samplespace.distributions.Constant* method), 20  
`sample()` (*samplespace.distributions.DiscreteUniform* method), 15  
`sample()` (*samplespace.distributions.Exponential* method), 23  
`sample()` (*samplespace.distributions.FiniteGeometric* method), 16  
`sample()` (*samplespace.distributions.FiniteGeometricCategorical* method), 19  
`sample()` (*samplespace.distributions.Gamma* method), 21  
`sample()` (*samplespace.distributions.Gaussian* method), 25  
`sample()` (*samplespace.distributions.Geometric* method), 16  
`sample()` (*samplespace.distributions.LogNormal* method), 23  
`sample()` (*samplespace.distributions.Pareto* method), 24  
`sample()` (*samplespace.distributions.Triangular* method), 22  
`sample()` (*samplespace.distributions.Uniform* method), 21  
`sample()` (*samplespace.distributions.UniformCategorical* method), 19  
`sample()` (*samplespace.distributions.VonMises* method), 23  
`sample()` (*samplespace.distributions.Weibull* method), 25  
`sample()` (*samplespace.distributions.WeightedCategorical* method), 18  
`sample()` (*samplespace.distributions.ZipfMandelbrot* method), 17  
`sample()` (*samplespace.distributions.ZipfMandelbrotCategorical* method), 20  
`sample()` (*samplespace.repeatablerandom.RepeatableRandomSequence* method), 8  
`sample_discrete_roulette()` (in module *samplespace.algorithms*), 31  
*samplespace.algorithms* module, 31  
*samplespace.distributions* module, 15  
*samplespace.pyyaml\_support* module, 33  
*samplespace.repeatablerandom* module, 3  
`seed()` (*samplespace.repeatablerandom.RepeatableRandomSequence* method), 3  
`setstate()` (*samplespace.repeatablerandom.RepeatableRandomSequence* method), 4  
`shuffle()` (*samplespace.repeatablerandom.RepeatableRandomSequence* method), 7  
`sigma()` (*samplespace.distributions.Gaussian* property), 25  
`sigma()` (*samplespace.distributions.LogNormal* property), 23

## T

*Triangular* (class in *samplespace.distributions*), 22  
`triangular()` (*samplespace.repeatablerandom.RepeatableRandomSequence* method), 8

## U

*Uniform* (class in *samplespace.distributions*), 20  
`uniform()` (*samplespace.repeatablerandom.RepeatableRandomSequence* method), 8  
*UniformCategorical* (class in *samplespace.distributions*), 18

## V

`value()` (*samplespace.distributions.Constant* property), 20  
*VonMises* (class in *samplespace.distributions*), 23  
`vonmisesvariate()` (*samplespace.repeatablerandom.RepeatableRandomSequence* method), 9

## W

*Weibull* (class in *samplespace.distributions*), 24  
`weibullvariate()` (*samplespace.repeatablerandom.RepeatableRandomSequence* method), 10  
*WeightedCategorical* (class in *samplespace.distributions*), 18

## Z

*ZipfMandelbrot* (class in *samplespace.distributions*), 16  
`zipfmandelbrot()` (*samplespace.repeatablerandom.RepeatableRandomSequence* method), 7

ZipfMandelbrotCategorical (*class in samplespace.distributions*), [19](#)