MeTILDA

Release 1.0.0v

Dr Chen

Feb 11, 2024

CONTENTS

CHAPTER

ONE

METILDA PACKAGE

1.1 Subpackages

1.1.1 metilda.controllers package

1.1.1.1 Submodules

1.1.1.2 metilda.controllers.Postgres module

class metilda.controllers.Postgres.**Postgres**

Bases: object

Class that contains the details of Postgres database

execute_insert_query(*query*, *record*, *need_last_row_id=True*)

execute_select_query(*query*, *record=None*)

execute_update_query(*query*, *record*)

1.1.1.3 metilda.controllers.controller_firestore module

metilda.controllers.controller_firestore.**getOrCreate_Collections**(*id='0'*)

1.1.1.4 metilda.controllers.pitch_art_wizard module

metilda.controllers.pitch_art_wizard.**add_new_user_from_admin**()

metilda.controllers.pitch_art_wizard.**all_audio_pitches**(*upload_id*)

metilda.controllers.pitch_art_wizard.**all_upload_pitches**()

metilda.controllers.pitch_art_wizard.**allowed_file**(*filename*)

metilda.controllers.pitch_art_wizard.**annotationTimeSelection**(*eaffilename*, *sound*, *start*, *end*, *text0*, *text1*, *text2*, *text3*, *text4*, *text5*)

metilda.controllers.pitch_art_wizard.**api**()

metilda.controllers.pitch_art_wizard.**audio**(*upload_id*)

metilda.controllers.pitch_art_wizard.**audio_analysis_image**(*upload_id*)

metilda.controllers.pitch_art_wizard.**authorize_user**()

metilda.controllers.pitch_art_wizard.**available_files**()

metilda.controllers.pitch_art_wizard.**avg_pitch**(*upload_id*)

metilda.controllers.pitch_art_wizard.**beta**(*a*, *b*, *size=None*)

Draw samples from a Beta distribution.

The Beta distribution is a special case of the Dirichlet distribution, and is related to the Gamma distribution. It has the probability distribution function

$$f(x; a, b) = \frac{1}{B(\alpha, \beta)} x^{\alpha-1}(1-x)^{\beta-1},$$

where the normalization, B, is the beta function,

$$B(\alpha, \beta) = \int_0^1 t^{\alpha-1}(1-t)^{\beta-1} dt.$$

It is often seen in Bayesian inference and order statistics.

Note: New code should use the *~numpy.random.Generator.beta* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **a** (*float or array_like of floats*) – Alpha, positive (>0).
- **b** (*float or array_like of floats*) – Beta, positive (>0).
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` and `b` are both scalars. Otherwise, `np.broadcast(a, b).size` samples are drawn.

Returns

out – Drawn samples from the parameterized beta distribution.

Return type

ndarray or scalar

See also:

random.Generator.beta

which should be used for new code.

metilda.controllers.pitch_art_wizard.**binomial**(*n*, *p*, *size=None*)

Draw samples from a binomial distribution.

Samples are drawn from a binomial distribution with specified parameters, n trials and p probability of success where n an integer >= 0 and p is in the interval [0,1]. (n may be input as a float, but it is truncated to an integer in use)

Note: New code should use the *~numpy.random.Generator.binomial* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **n** (*int or array_like of ints*) – Parameter of the distribution, >= 0. Floats are also accepted, but they will be truncated to integers.
- **p** (*float or array_like of floats*) – Parameter of the distribution, >= 0 and <=1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

Returns

out – Drawn samples from the parameterized binomial distribution, where each sample is equal to the number of successes over the n trials.

Return type

ndarray or scalar

See also:

scipy.stats.binom

probability density function, distribution or cumulative density function, etc.

random.Generator.binomial

which should be used for new code.

Notes

The probability density for the binomial distribution is

$$P(N) = \binom{n}{N} p^N (1-p)^{n-N},$$

where n is the number of trials, p is the probability of success, and N is the number of successes.

When estimating the standard error of a proportion in a population by using a random sample, the normal distribution works well unless the product p*n <=5, where p = population proportion estimate, and n = number of samples, in which case the binomial distribution is used instead. For example, a sample of 15 people shows 4 who are left handed, and 11 who are right handed. Then p = 4/15 = 27%. 0.27*15 = 4, so the binomial distribution should be used in this case.

References

Examples

Draw samples from the distribution:

```
>>> n, p = 10, .5  # number of trials, probability of each trial
>>> s = np.random.binomial(n, p, 1000)
# result of flipping a coin 10 times, tested 1000 times.
```

A real world example. A company drills 9 wild-cat oil exploration wells, each with an estimated probability of success of 0.1. All nine wells fail. What is the probability of that happening?

Let's do 20,000 trials of the model, and count the number that generate zero positive results.

```
>>> sum(np.random.binomial(9, 0.1, 20000) == 0)/20000.
# answer = 0.38885, or 38%.
```

metilda.controllers.pitch_art_wizard.**chisquare**(*df*, *size=None*)

Draw samples from a chi-square distribution.

When *df* independent random variables, each with standard normal distributions (mean 0, variance 1), are squared and summed, the resulting distribution is chi-square (see Notes). This distribution is often used in hypothesis testing.

Note: New code should use the *~numpy.random.Generator.chisquare* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **df** (*float or array_like of floats*) – Number of degrees of freedom, must be > 0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

Returns

out – Drawn samples from the parameterized chi-square distribution.

Return type

ndarray or scalar

Raises

ValueError – When *df* <= 0 or when an inappropriate *size* (e.g. `size=-1`) is given.

See also:

random.Generator.chisquare

which should be used for new code.

Notes

The variable obtained by summing the squares of *df* independent, standard normally distributed random variables:

$$Q = \sum_{i=0}^{\mathtt{df}} X_i^2$$

is chi-square distributed, denoted

$$Q \sim \chi_k^2.$$

The probability density function of the chi-squared distribution is

$$p(x) = \frac{(1/2)^{k/2}}{\Gamma(k/2)} x^{k/2-1} e^{-x/2},$$

where Γ is the gamma function,

$$\Gamma(x) = \int_0^{-\infty} t^{x-1} e^{-t} dt.$$

References

Examples

```
>>> np.random.chisquare(2,4)
array([ 1.89920014,  9.00867716,  3.13710533,  5.62318272]) # random
```

metilda.controllers.pitch_art_wizard.**choice**(*a*, *size=None*, *replace=True*, *p=None*)

Generates a random sample from a given 1-D array

New in version 1.7.0.

Note: New code should use the *~numpy.random.Generator.choice* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **a** (*1-D array-like or int*) – If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if it were `np.arange(a)`
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.
- **replace** (*boolean, optional*) – Whether the sample is with or without replacement. Default is True, meaning that a value of `a` can be selected multiple times.
- **p** (*1-D array-like, optional*) – The probabilities associated with each entry in a. If not given, the sample assumes a uniform distribution over all entries in `a`.

Returns

samples – The generated random samples

Return type

single item or ndarray

Raises

ValueError – If a is an int and less than zero, if a or p are not 1-dimensional, if a is an array-like of size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size

See also:

randint, *shuffle*, *permutation*

random.Generator.choice

which should be used in new code

Notes

Setting user-specified probabilities through `p` uses a more general but less efficient sampler than the default. The general sampler produces a different sample than the optimized sampler even if each element of `p` is 1 / len(a).

Sampling random rows from a 2-D array is not possible with this function, but is possible with *Generator.choice* through its `axis` keyword.

Examples

Generate a uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3)
array([0, 3, 4]) # random
>>> #This is equivalent to np.random.randint(0,5,3)
```

Generate a non-uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0])
array([3, 3, 0]) # random
```

Generate a uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False)
array([3,1,0]) # random
>>> #This is equivalent to np.random.permutation(np.arange(5))[:3]
```

Generate a non-uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
array([2, 3, 0]) # random
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:

```
>>> aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']
>>> np.random.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])
array(['pooh', 'pooh', 'pooh', 'Christopher', 'piglet'], # random
      dtype='<U11')
```

metilda.controllers.pitch_art_wizard.**countVoicedFrames**(*sound*)

metilda.controllers.pitch_art_wizard.**create_analysis**()

metilda.controllers.pitch_art_wizard.**create_db_user**()

metilda.controllers.pitch_art_wizard.**create_eaf**()

metilda.controllers.pitch_art_wizard.**create_file**()

metilda.controllers.pitch_art_wizard.**create_folder**()

metilda.controllers.pitch_art_wizard.**create_image**()

metilda.controllers.pitch_art_wizard.**create_user_research_language**()

metilda.controllers.pitch_art_wizard.**create_user_research_role**()

metilda.controllers.pitch_art_wizard.**delete_eaf_file**()

metilda.controllers.pitch_art_wizard.**delete_file**()

metilda.controllers.pitch_art_wizard.**delete_folder**()

metilda.controllers.pitch_art_wizard.**delete_image**()

metilda.controllers.pitch_art_wizard.**delete_previous_user_research_language**()

metilda.controllers.pitch_art_wizard.**delete_previous_user_roles**()

metilda.controllers.pitch_art_wizard.**delete_recording**()

metilda.controllers.pitch_art_wizard.**delete_shared_user**()

metilda.controllers.pitch_art_wizard.**delete_user**()

metilda.controllers.pitch_art_wizard.**dirichlet**(*alpha*, *size=None*)

Draw samples from the Dirichlet distribution.

Draw *size* samples of dimension k from a Dirichlet distribution. A Dirichlet-distributed random variable can be seen as a multivariate generalization of a Beta distribution. The Dirichlet distribution is a conjugate prior of a multinomial distribution in Bayesian inference.

Note: New code should use the *~numpy.random.Generator.dirichlet* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **alpha** (*sequence of floats, length k*) – Parameter of the distribution (length `k` for sample of length `k`).
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n)`, then `m * n * k` samples are drawn. Default is None, in which case a vector of length `k` is returned.

Returns

samples – The drawn samples, of shape `(size, k)`.

Return type

ndarray,

Raises

ValueError – If any value in `alpha` is less than or equal to zero

See also:

`random.Generator.dirichlet`
which should be used for new code.

Notes

The Dirichlet distribution is a distribution over vectors x that fulfil the conditions $x_i > 0$ and $\sum_{i=1}^{k} x_i = 1$.

The probability density function p of a Dirichlet-distributed random vector X is proportional to

$$p(x) \propto \prod_{i=1}^{k} x_i^{\alpha_i - 1},$$

where α is a vector containing the positive concentration parameters.

The method uses the following property for computation: let Y be a random vector which has components that follow a standard gamma distribution, then $X = \frac{1}{\sum_{i=1}^{k} Y_i} Y$ is Dirichlet-distributed

References

Examples

Taking an example cited in Wikipedia, this distribution can be used if one wanted to cut strings (each of initial length 1.0) into K pieces with different lengths, where each piece had, on average, a designated average length, but allowing some variation in the relative sizes of the pieces.

```
>>> s = np.random.dirichlet((10, 5, 3), 20).transpose()
```

```
>>> import matplotlib.pyplot as plt
>>> plt.barh(range(20), s[0])
>>> plt.barh(range(20), s[1], left=s[0], color='g')
>>> plt.barh(range(20), s[2], left=s[0]+s[1], color='r')
>>> plt.title("Lengths of Strings")
```

metilda.controllers.pitch_art_wizard.**download_file**()

metilda.controllers.pitch_art_wizard.**drawSound**(*upload_id*)

metilda.controllers.pitch_art_wizard.**drawSoundWithTime**(*sound*, *startTime*, *endTime*)

metilda.controllers.pitch_art_wizard.**exponential**(*scale=1.0*, *size=None*)

Draw samples from an exponential distribution.

Its probability density function is

$$f(x; \frac{1}{\beta}) = \frac{1}{\beta} \exp(-\frac{x}{\beta}),$$

for `x > 0` and 0 elsewhere. β is the scale parameter, which is the inverse of the rate parameter $\lambda = 1/\beta$. The rate parameter is an alternative, widely used parameterization of the exponential distribution [3]_.

The exponential distribution is a continuous analogue of the geometric distribution. It describes many common situations, such as the size of raindrops measured over many rainstorms [1]_, or the time between page requests to Wikipedia [2]_.

Note: New code should use the *~numpy.random.Generator.exponential* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **scale** (*float or array_like of floats*) – The scale parameter, $\beta = 1/\lambda$. Must be non-negative.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

Returns
out – Drawn samples from the parameterized exponential distribution.

Return type
ndarray or scalar

See also:

`random.Generator.exponential`
which should be used for new code.

References

`metilda.controllers.pitch_art_wizard.f`(*dfnum*, *dfden*, *size=None*)

Draw samples from an F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters must be greater than zero.

The random variate of the F distribution (also known as the Fisher distribution) is a continuous probability distribution that arises in ANOVA tests, and is the ratio of two chi-square variates.

Note: New code should use the *~numpy.random.Generator.f* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **dfnum** (*float or array_like of floats*) – Degrees of freedom in numerator, must be > 0.
- **dfden** (*float or array_like of float*) – Degrees of freedom in denominator, must be > 0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum` and `dfden` are both scalars. Otherwise, `np.broadcast(dfnum, dfden).size` samples are drawn.

Returns
out – Drawn samples from the parameterized Fisher distribution.

Return type
ndarray or scalar

See also:

`scipy.stats.f`
probability density function, distribution or cumulative density function, etc.

random.Generator.f
which should be used for new code.

Notes

The F statistic is used to compare in-group variances to between-group variances. Calculating the distribution depends on the sampling, and so it is a function of the respective degrees of freedom in the problem. The variable *dfnum* is the number of samples minus one, the between-groups degrees of freedom, while *dfden* is the within-groups degrees of freedom, the sum of the number of samples in each group minus the number of groups.

References

Examples

An example from Glantz[1], pp 47-40:

Two groups, children of diabetics (25 people) and children from people without diabetes (25 controls). Fasting blood glucose was measured, case group had a mean value of 86.1, controls had a mean value of 82.2. Standard deviations were 2.09 and 2.49 respectively. Are these data consistent with the null hypothesis that the parents diabetic status does not affect their children's blood glucose levels? Calculating the F statistic from the data gives a value of 36.01.

Draw samples from the distribution:

```
>>> dfnum = 1. # between group degrees of freedom
>>> dfden = 48. # within groups degrees of freedom
>>> s = np.random.f(dfnum, dfden, 1000)
```

The lower bound for the top 1% of the samples is :

```
>>> np.sort(s)[-10]
7.61988120985 # random
```

So there is about a 1% chance that the F statistic will exceed 7.62, the measured value is 36, so the null hypothesis is rejected at the 1% level.

metilda.controllers.pitch_art_wizard.**formantCountAtFrame**(*sound*, *frame*)

metilda.controllers.pitch_art_wizard.**formantFrameCount**(*sound*)

metilda.controllers.pitch_art_wizard.**formantValueAtTime**(*sound*, *formantNumber*, *time*)

metilda.controllers.pitch_art_wizard.**gamma**(*shape*, *scale=1.0*, *size=None*)
Draw samples from a Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, *shape* (sometimes designated "k") and *scale* (sometimes designated "theta"), where both parameters are > 0.

Note: New code should use the *~numpy.random.Generator.gamma* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **shape** (*float or array_like of floats*) – The shape of the gamma distribution. Must be non-negative.

- **scale** (*float or array_like of floats, optional*) – The scale of the gamma distribution. Must be non-negative. Default is equal to 1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` and `scale` are both scalars. Otherwise, `np.broadcast(shape, scale).size` samples are drawn.

Returns
out – Drawn samples from the parameterized gamma distribution.

Return type
ndarray or scalar

See also:

scipy.stats.gamma
probability density function, distribution or cumulative density function, etc.

random.Generator.gamma
which should be used for new code.

Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

References

Examples

Draw samples from the distribution:

```
>>> shape, scale = 2., 2.  # mean=4, std=2*sqrt(2)
>>> s = np.random.gamma(shape, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1)*(np.exp(-bins/scale) /
...                      (sps.gamma(shape)*scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**geometric**(*p*, *size=None*)

Draw samples from the geometric distribution.

Bernoulli trials are experiments with one of two outcomes: success or failure (an example of such an experiment is flipping a coin). The geometric distribution models the number of trials that must be run in order to achieve success. It is therefore supported on the positive integers, `k = 1, 2, ...`.

The probability mass function of the geometric distribution is

$$f(k) = (1-p)^{k-1}p$$

where p is the probability of success of an individual trial.

Note: New code should use the *~numpy.random.Generator.geometric* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **p** (*float or array_like of floats*) – The probability of success of an individual trial.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

Returns

out – Drawn samples from the parameterized geometric distribution.

Return type

ndarray or scalar

See also:

random.Generator.geometric

which should be used for new code.

Examples

Draw ten thousand values from the geometric distribution, with the probability of an individual success equal to 0.35:

```
>>> z = np.random.geometric(p=0.35, size=10000)
```

How many trials succeeded after a single run?

```
>>> (z == 1).sum() / 10000.
0.34889999999999999 #random
```

metilda.controllers.pitch_art_wizard.**getBounds**(*sound*)

metilda.controllers.pitch_art_wizard.**getEnergy**(*sound*)

metilda.controllers.pitch_art_wizard.**getOrCreateWords**()

metilda.controllers.pitch_art_wizard.**get_admin**(*user_id*)

metilda.controllers.pitch_art_wizard.**get_all_images**(*user_id*)

metilda.controllers.pitch_art_wizard.**get_analyses_for_file**(*file_id*)

metilda.controllers.pitch_art_wizard.**get_analyses_for_image**(*image_id*)

metilda.controllers.pitch_art_wizard.**get_analysis_file_path**(*analysis_id*)

metilda.controllers.pitch_art_wizard.**get_eaf_file**(*audio_id*)

metilda.controllers.pitch_art_wizard.**get_eaf_file_path**(*eaf_id*)

metilda.controllers.pitch_art_wizard.**get_eafs_for_files**(*audio_id*)

metilda.controllers.pitch_art_wizard.**get_file**(*user_id*)

metilda.controllers.pitch_art_wizard.**get_files_and_folders**(*user_id*, *folder_name*)

metilda.controllers.pitch_art_wizard.**get_image_for_analysis**(*analysis_id*)

metilda.controllers.pitch_art_wizard.**get_shared_users**(*audio_id*)

metilda.controllers.pitch_art_wizard.**get_state**(*legacy=True*)

Return a tuple representing the internal state of the generator.

For more details, see *set_state*.

Parameters

legacy (*bool, optional*) – Flag indicating to return a legacy tuple state when the BitGenerator is MT19937, instead of a dict. Raises ValueError if the underlying bit generator is not an instance of MT19937.

Returns

out – If legacy is True, the returned tuple has the following items:

1. the string 'MT19937'.
2. a 1-D array of 624 unsigned integer keys.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

If *legacy* is False, or the BitGenerator is not MT19937, then state is returned as a dictionary.

Return type

{tuple(str, ndarray of 624 uints, int, int, float), dict}

See also:

set_state

Notes

set_state and *get_state* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

metilda.controllers.pitch_art_wizard.**get_student_recordings**()

metilda.controllers.pitch_art_wizard.**get_user_research_language**(*user_id*)

metilda.controllers.pitch_art_wizard.**get_user_roles**(*user_id*)

metilda.controllers.pitch_art_wizard.**get_users**()

metilda.controllers.pitch_art_wizard.**gumbel**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a Gumbel distribution.

Draw samples from a Gumbel distribution with specified location and scale. For more information on the Gumbel distribution, see Notes and References below.

Note: New code should use the *~numpy.random.Generator.gumbel* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **loc** (*float or array_like of floats, optional*) – The location of the mode of the distribution. Default is 0.
- **scale** (*float or array_like of floats, optional*) – The scale parameter of the distribution. Default is 1. Must be non- negative.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

Returns

out – Drawn samples from the parameterized Gumbel distribution.

Return type

ndarray or scalar

See also:

`scipy.stats.gumbel_l`, `scipy.stats.gumbel_r`, `scipy.stats.genextreme`, *weibull*

random.Generator.gumbel

which should be used for new code.

Notes

The Gumbel (or Smallest Extreme Value (SEV) or the Smallest Extreme Value Type I) distribution is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. The Gumbel is a special case of the Extreme Value Type I distribution for maximums from distributions with "exponential-like" tails.

The probability density for the Gumbel distribution is

$$p(x) = \frac{e^{-(x-\mu)/\beta}}{\beta} e^{-e^{-(x-\mu)/\beta}},$$

where μ is the mode, a location parameter, and β is the scale parameter.

The Gumbel (named for German mathematician Emil Julius Gumbel) was used very early in the hydrology literature, for modeling the occurrence of flood events. It is also used for modeling maximum wind speed and rainfall rates. It is a "fat-tailed" distribution - the probability of an event in the tail of the distribution is larger than if one used a Gaussian, hence the surprisingly frequent occurrence of 100-year floods. Floods were initially modeled as a Gaussian process, which underestimated the frequency of extreme events.

It is one of a class of extreme value distributions, the Generalized Extreme Value (GEV) distributions, which also includes the Weibull and Frechet.

The function has a mean of $\mu + 0.57721\beta$ and a variance of $\frac{\pi^2}{6}\beta^2$.

References

Examples

Draw samples from the distribution:

```
>>> mu, beta = 0, 0.1 # location and scale
>>> s = np.random.gumbel(mu, beta, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp( -np.exp( -(bins - mu) /beta) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Show how an extreme value distribution can arise from a Gaussian process and compare to a Gaussian:

```
>>> means = []
>>> maxima = []
>>> for i in range(0,1000) :
...    a = np.random.normal(mu, beta, 1000)
...    means.append(a.mean())
...    maxima.append(a.max())
>>> count, bins, ignored = plt.hist(maxima, 30, density=True)
>>> beta = np.std(maxima) * np.sqrt(6) / np.pi
>>> mu = np.mean(maxima) - 0.57721*beta
>>> plt.plot(bins, (1/beta)*np.exp(-(bins - mu)/beta)
...          * np.exp(-np.exp(-(bins - mu)/beta)),
```

(continues on next page)

(continued from previous page)

```
...          linewidth=2, color='r')
>>> plt.plot(bins, 1/(beta * np.sqrt(2 * np.pi))
...          * np.exp(-(bins - mu)**2 / (2 * beta**2)),
...          linewidth=2, color='g')
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**harmonicityGetMax**(*sound*, *start*, *end*)

metilda.controllers.pitch_art_wizard.**harmonicityGetMin**(*sound*, *start*, *end*)

metilda.controllers.pitch_art_wizard.**harmonicityValueAtTime**(*sound*, *time*)

metilda.controllers.pitch_art_wizard.**hypergeometric**(*ngood*, *nbad*, *nsample*, *size=None*)

Draw samples from a Hypergeometric distribution.

Samples are drawn from a hypergeometric distribution with specified parameters, *ngood* (ways to make a good selection), *nbad* (ways to make a bad selection), and *nsample* (number of items sampled, which is less than or equal to the sum `ngood + nbad`).

Note: New code should use the *~numpy.random.Generator.hypergeometric* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **ngood** (*int or array_like of ints*) – Number of ways to make a good selection. Must be nonnegative.
- **nbad** (*int or array_like of ints*) – Number of ways to make a bad selection. Must be nonnegative.
- **nsample** (*int or array_like of ints*) – Number of items sampled. Must be at least 1 and at most `ngood + nbad`.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if *ngood*, *nbad*, and *nsample* are all scalars. Otherwise, `np.broadcast(ngood, nbad, nsample).size` samples are drawn.

Returns

out – Drawn samples from the parameterized hypergeometric distribution. Each sample is the number of good items within a randomly selected subset of size *nsample* taken from a set of *ngood* good items and *nbad* bad items.

Return type

ndarray or scalar

See also:

`scipy.stats.hypergeom`
probability density function, distribution or cumulative density function, etc.

`random.Generator.hypergeometric`
which should be used for new code.

Notes

The probability density for the Hypergeometric distribution is

$$P(x) = \frac{\binom{g}{x}\binom{b}{n-x}}{\binom{g+b}{n}},$$

where $0 \leq x \leq n$ and $n - b \leq x \leq g$

for P(x) the probability of `x` good results in the drawn sample, g = *ngood*, b = *nbad*, and n = *nsample*.

Consider an urn with black and white marbles in it, *ngood* of them are black and *nbad* are white. If you draw *nsample* balls without replacement, then the hypergeometric distribution describes the distribution of black balls in the drawn sample.

Note that this distribution is very similar to the binomial distribution, except that in this case, samples are drawn without replacement, whereas in the Binomial case samples are drawn with replacement (or the sample space is infinite). As the sample space becomes large, this distribution approaches the binomial.

References

Examples

Draw samples from the distribution:

```
>>> ngood, nbad, nsamp = 100, 2, 10
# number of good, number of bad, and number of samples
>>> s = np.random.hypergeometric(ngood, nbad, nsamp, 1000)
>>> from matplotlib.pyplot import hist
>>> hist(s)
#   note that it is very unlikely to grab both bad items
```

Suppose you have an urn with 15 white and 15 black marbles. If you pull 15 marbles at random, how likely is it that 12 or more of them are one color?

```
>>> s = np.random.hypergeometric(15, 15, 15, 100000)
>>> sum(s>=12)/100000. + sum(s<=3)/100000.
#   answer = 0.003 ... pretty unlikely!
```

metilda.controllers.pitch_art_wizard.**insert_image_analysis_ids**()

metilda.controllers.pitch_art_wizard.**intensityBounds**(*sound*)

metilda.controllers.pitch_art_wizard.**laplace**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from the Laplace or double exponential distribution with specified location (or mean) and scale (decay).

The Laplace distribution is similar to the Gaussian/normal distribution, but is sharper at the peak and has fatter tails. It represents the difference between two independent, identically distributed exponential random variables.

Note: New code should use the *~numpy.random.Generator.laplace* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **loc** (*float or array_like of floats, optional*) – The position, μ, of the distribution peak. Default is 0.
- **scale** (*float or array_like of floats, optional*) – λ, the exponential decay. Default is 1. Must be non- negative.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

Returns

out – Drawn samples from the parameterized Laplace distribution.

Return type

ndarray or scalar

See also:

`random.Generator.laplace`
which should be used for new code.

Notes

It has the probability density function

$$f(x;\mu,\lambda) = \frac{1}{2\lambda}\exp\left(-\frac{|x-\mu|}{\lambda}\right).$$

The first law of Laplace, from 1774, states that the frequency of an error can be expressed as an exponential function of the absolute magnitude of the error, which leads to the Laplace distribution. For many problems in economics and health sciences, this distribution seems to model the data better than the standard Gaussian distribution.

References

Examples

Draw samples from the distribution

```
>>> loc, scale = 0., 1.
>>> s = np.random.laplace(loc, scale, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> x = np.arange(-8., 8., .01)
>>> pdf = np.exp(-abs(x-loc)/scale)/(2.*scale)
>>> plt.plot(x, pdf)
```

Plot Gaussian for comparison:

```
>>> g = (1/(scale * np.sqrt(2 * np.pi)) *
...      np.exp(-(x - loc)**2 / (2 * scale**2)))
>>> plt.plot(x,g)
```

metilda.controllers.pitch_art_wizard.**logistic**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw samples from a logistic distribution.

Samples are drawn from a logistic distribution with specified parameters, loc (location or mean, also median), and scale (>0).

Note: New code should use the *~numpy.random.Generator.logistic* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **loc** (*float or array_like of floats, optional*) – Parameter of the distribution. Default is 0.
- **scale** (*float or array_like of floats, optional*) – Parameter of the distribution. Must be non-negative. Default is 1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

Returns

out – Drawn samples from the parameterized logistic distribution.

Return type

ndarray or scalar

See also:

`scipy.stats.logistic`
probability density function, distribution or cumulative density function, etc.

`random.Generator.logistic`
which should be used for new code.

Notes

The probability density for the Logistic distribution is

$$P(x) = P(x) = \frac{e^{-(x-\mu)/s}}{s(1+e^{-(x-\mu)/s})^2},$$

where μ = location and s = scale.

The Logistic distribution is used in Extreme Value problems where it can act as a mixture of Gumbel distributions, in Epidemiology, and by the World Chess Federation (FIDE) where it is used in the Elo ranking system, assuming the performance of each player is a logistically distributed random variable.

References

Examples

Draw samples from the distribution:

```
>>> loc, scale = 10, 1
>>> s = np.random.logistic(loc, scale, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=50)
```

plot against distribution

```
>>> def logist(x, loc, scale):
...     return np.exp((loc-x)/scale)/(scale*(1+np.exp((loc-x)/scale))**2)
>>> lgst_val = logist(bins, loc, scale)
>>> plt.plot(bins, lgst_val * count.max() / lgst_val.max())
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**lognormal**(*mean=0.0*, *sigma=1.0*, *size=None*)

Draw samples from a log-normal distribution.

Draw samples from a log-normal distribution with specified mean, standard deviation, and array shape. Note that the mean and standard deviation are not the values for the distribution itself, but of the underlying normal distribution it is derived from.

Note: New code should use the *~numpy.random.Generator.lognormal* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **mean** (*float or array_like of floats, optional*) – Mean value of the underlying normal distribution. Default is 0.
- **sigma** (*float or array_like of floats, optional*) – Standard deviation of the underlying normal distribution. Must be non-negative. Default is 1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `sigma` are both scalars. Otherwise, `np.broadcast(mean, sigma).size` samples are drawn.

Returns

out – Drawn samples from the parameterized log-normal distribution.

Return type

ndarray or scalar

See also:

`scipy.stats.lognorm`
probability density function, distribution, cumulative density function, etc.

`random.Generator.lognormal`
which should be used for new code.

Notes

A variable x has a log-normal distribution if *log(x)* is normally distributed. The probability density function for the log-normal distribution is:

$$p(x) = \frac{1}{\sigma x\sqrt{2\pi}} e^{(-\frac{(ln(x)-\mu)^2}{2\sigma^2})}$$

where μ is the mean and σ is the standard deviation of the normally distributed logarithm of the variable. A log-normal distribution results if a random variable is the *product* of a large number of independent, identically-distributed variables in the same way that a normal distribution results if the variable is the *sum* of a large number of independent, identically-distributed variables.

References

Examples

Draw samples from the distribution:

```
>>> mu, sigma = 3., 1. # mean and standard deviation
>>> s = np.random.lognormal(mu, sigma, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 100, density=True, align='mid')
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, linewidth=2, color='r')
>>> plt.axis('tight')
>>> plt.show()
```

Demonstrate that taking the products of random samples from a uniform distribution can be fit well by a log-normal probability density function.

```
>>> # Generate a thousand samples: each is the product of 100 random
>>> # values, drawn from a normal distribution.
>>> b = []
>>> for i in range(1000):
...     a = 10. + np.random.standard_normal(100)
...     b.append(np.product(a))
```

```
>>> b = np.array(b) / np.min(b) # scale values to be positive
>>> count, bins, ignored = plt.hist(b, 100, density=True, align='mid')
>>> sigma = np.std(np.log(b))
>>> mu = np.mean(np.log(b))
```

```
>>> x = np.linspace(min(bins), max(bins), 10000)
>>> pdf = (np.exp(-(np.log(x) - mu)**2 / (2 * sigma**2))
...        / (x * sigma * np.sqrt(2 * np.pi)))
```

```
>>> plt.plot(x, pdf, color='r', linewidth=2)
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**logseries**(*p*, *size=None*)

Draw samples from a logarithmic series distribution.

Samples are drawn from a log series distribution with specified shape parameter, 0 <= p < 1.

Note: New code should use the *~numpy.random.Generator.logseries* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **p** (*float or array_like of floats*) – Shape parameter for the distribution. Must be in the range [0, 1).
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `p` is a scalar. Otherwise, `np.array(p).size` samples are drawn.

Returns

out – Drawn samples from the parameterized logarithmic series distribution.

Return type

ndarray or scalar

See also:

scipy.stats.logser

probability density function, distribution or cumulative density function, etc.

random.Generator.logseries

which should be used for new code.

Notes

The probability density for the Log Series distribution is

$$P(k) = \frac{-p^k}{k \ln(1-p)},$$

where p = probability.

The log series distribution is frequently used to represent species richness and occurrence, first proposed by Fisher, Corbet, and Williams in 1943 [2]. It may also be used to model the numbers of occupants seen in cars [3].

References

Examples

Draw samples from the distribution:

```
>>> a = .6
>>> s = np.random.logseries(a, 10000)
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s)
```

plot against distribution

```
>>> def logseries(k, p):
...     return -p**k/(k*np.log(1-p))
>>> plt.plot(bins, logseries(bins, a)*count.max()/
...          logseries(bins, a).max(), 'r')
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**modifyPitchOrImageDetails**(*upload_id*)

metilda.controllers.pitch_art_wizard.**move_to_folder**()

metilda.controllers.pitch_art_wizard.**multinomial**(*n*, *pvals*, *size=None*)

Draw samples from a multinomial distribution.

The multinomial distribution is a multivariate generalization of the binomial distribution. Take an experiment with one of `p` possible outcomes. An example of such an experiment is throwing a dice, where the outcome can be 1 through 6. Each sample drawn from the distribution represents *n* such experiments. Its values, `X_i = [X_0, X_1, ..., X_p]`, represent the number of times the outcome was `i`.

Note: New code should use the *~numpy.random.Generator.multinomial* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **n** (*int*) – Number of experiments.
- **pvals** (*sequence of floats, length p*) – Probabilities of each of the `p` different outcomes. These must sum to 1 (however, the last element is always assumed to account for the remaining probability, as long as `sum(pvals[:-1]) <= 1)`.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out – The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Return type
ndarray

See also:

random.Generator.multinomial
which should be used for new code.

Examples

Throw a dice 20 times:

```
>>> np.random.multinomial(20, [1/6.]*6, size=1)
array([[4, 1, 7, 5, 2, 1]]) # random
```

It landed 4 times on 1, once on 2, etc.

Now, throw the dice 20 times, and 20 times again:

```
>>> np.random.multinomial(20, [1/6.]*6, size=2)
array([[3, 4, 3, 3, 4, 3], # random
       [2, 4, 3, 4, 0, 7]])
```

For the first run, we threw 3 times 1, 4 times 2, etc. For the second, we threw 2 times 1, 4 times 2, etc.

A loaded die is more likely to land on number 6:

```
>>> np.random.multinomial(100, [1/7.]*5 + [2/7.])
array([11, 16, 14, 17, 16, 26]) # random
```

The probability inputs should be normalized. As an implementation detail, the value of the last entry is ignored and assumed to take up any leftover probability mass, but this should not be relied on. A biased coin which has twice as much weight on one side as on the other should be sampled like so:

```
>>> np.random.multinomial(100, [1.0 / 3, 2.0 / 3])  # RIGHT
array([38, 62]) # random
```

not like:

```
>>> np.random.multinomial(100, [1.0, 2.0])  # WRONG
Traceback (most recent call last):
ValueError: pvals < 0, pvals > 1 or pvals contains NaNs
```

metilda.controllers.pitch_art_wizard.**multivariate_normal**(*mean*, *cov*, *size=None*, *check_valid='warn'*, *tol=1e-8*)

Draw random samples from a multivariate normal distribution.

The multivariate normal, multinormal or Gaussian distribution is a generalization of the one-dimensional normal distribution to higher dimensions. Such a distribution is specified by its mean and covariance matrix. These parameters are analogous to the mean (average or "center") and variance (standard deviation, or "width," squared) of the one-dimensional normal distribution.

Note: New code should use the *~numpy.random.Generator.multivariate_normal* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **mean** (*1-D array_like, of length N*) – Mean of the N-dimensional distribution.

- **cov** (*2-D array_like, of shape (N, N)*) – Covariance matrix of the distribution. It must be symmetric and positive-semidefinite for proper sampling.
- **size** (*int or tuple of ints, optional*) – Given a shape of, for example, `(m,n,k)`, `m*n*k` samples are generated, and packed in an *m*-by-*n*-by-*k* arrangement. Because each sample is *N*-dimensional, the output shape is `(m,n,k,N)`. If no shape is specified, a single (*N*-D) sample is returned.
- **check_valid** (*{ 'warn', 'raise', 'ignore' }, optional*) – Behavior when the covariance matrix is not positive semidefinite.
- **tol** (*float, optional*) – Tolerance when checking the singular values in covariance matrix. cov is cast to double before the check.

Returns

out – The drawn samples, of shape *size*, if that was provided. If not, the shape is `(N,)`.

In other words, each entry `out[i,j,...,:]` is an N-dimensional value drawn from the distribution.

Return type
ndarray

See also:

`random.Generator.multivariate_normal`
which should be used for new code.

Notes

The mean is a coordinate in N-dimensional space, which represents the location where samples are most likely to be generated. This is analogous to the peak of the bell curve for the one-dimensional or univariate normal distribution.

Covariance indicates the level to which two variables vary together. From the multivariate normal distribution, we draw N-dimensional samples, $X = [x_1, x_2, ...x_N]$. The covariance matrix element C_{ij} is the covariance of x_i and x_j. The element C_{ii} is the variance of x_i (i.e. its "spread").

Instead of specifying the full covariance matrix, popular approximations include:

- Spherical covariance (*cov* is a multiple of the identity matrix)
- Diagonal covariance (*cov* has non-negative elements, and only on the diagonal)

This geometrical property can be seen in two dimensions by plotting generated data-points:

```
>>> mean = [0, 0]
>>> cov = [[1, 0], [0, 100]]  # diagonal covariance
```

Diagonal covariance means that points are oriented along x or y-axis:

```
>>> import matplotlib.pyplot as plt
>>> x, y = np.random.multivariate_normal(mean, cov, 5000).T
>>> plt.plot(x, y, 'x')
>>> plt.axis('equal')
>>> plt.show()
```

Note that the covariance matrix must be positive semidefinite (a.k.a. nonnegative-definite). Otherwise, the behavior of this method is undefined and backwards compatibility is not guaranteed.

References

Examples

```
>>> mean = (1, 2)
>>> cov = [[1, 0], [0, 1]]
>>> x = np.random.multivariate_normal(mean, cov, (3, 3))
>>> x.shape
(3, 3, 2)
```

Here we generate 800 samples from the bivariate normal distribution with mean [0, 0] and covariance matrix [[6, -3], [-3, 3.5]]. The expected variances of the first and second components of the sample are 6 and 3.5, respectively, and the expected correlation coefficient is -3/sqrt(6*3.5) -0.65465.

```
>>> cov = np.array([[6, -3], [-3, 3.5]])
>>> pts = np.random.multivariate_normal([0, 0], cov, size=800)
```

Check that the mean, covariance, and correlation coefficient of the sample are close to the expected values:

```
>>> pts.mean(axis=0)
array([ 0.0326911 , -0.01280782])  # may vary
>>> np.cov(pts.T)
array([[ 5.96202397, -2.85602287],
       [-2.85602287,  3.47613949]])  # may vary
>>> np.corrcoef(pts.T)[0, 1]
-0.6273591314603949  # may vary
```

We can visualize this data with a scatter plot. The orientation of the point cloud illustrates the negative correlation of the components of this sample.

```
>>> import matplotlib.pyplot as plt
>>> plt.plot(pts[:, 0], pts[:, 1], '.', alpha=0.5)
>>> plt.axis('equal')
>>> plt.grid()
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**negative_binomial**(*n*, *p*, *size=None*)

Draw samples from a negative binomial distribution.

Samples are drawn from a negative binomial distribution with specified parameters, *n* successes and *p* probability of success where *n* is > 0 and *p* is in the interval [0, 1].

Note: New code should use the *~numpy.random.Generator.negative_binomial* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **n** (*float or array_like of floats*) – Parameter of the distribution, > 0.
- **p** (*float or array_like of floats*) – Parameter of the distribution, >= 0 and <=1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `n` and `p` are both scalars. Otherwise, `np.broadcast(n, p).size` samples are drawn.

Returns

out – Drawn samples from the parameterized negative binomial distribution, where each sample is equal to N, the number of failures that occurred before a total of n successes was reached.

Return type

ndarray or scalar

See also:

random.Generator.negative_binomial
which should be used for new code.

Notes

The probability mass function of the negative binomial distribution is

$$P(N;n,p) = \frac{\Gamma(N+n)}{N!\Gamma(n)}p^n(1-p)^N,$$

where n is the number of successes, p is the probability of success, $N+n$ is the number of trials, and Γ is the gamma function. When n is an integer, $\frac{\Gamma(N+n)}{N!\Gamma(n)} = \binom{N+n-1}{N}$, which is the more common form of this term in the pmf. The negative binomial distribution gives the probability of N failures given n successes, with a success on the last trial.

If one throws a die repeatedly until the third time a "1" appears, then the probability distribution of the number of non-"1"s that appear before the third "1" is a negative binomial distribution.

References

Examples

Draw samples from the distribution:

A real world example. A company drills wild-cat oil exploration wells, each with an estimated probability of success of 0.1. What is the probability of having one success for each successive well, that is what is the probability of a single success after drilling 5 wells, after 6 wells, etc.?

```
>>> s = np.random.negative_binomial(1, 0.1, 100000)
>>> for i in range(1, 11):
...    probability = sum(s<i) / 100000.
...    print(i, "wells drilled, probability of one success =", probability)
```

metilda.controllers.pitch_art_wizard.**noncentral_chisquare**(*df*, *nonc*, *size=None*)

Draw samples from a noncentral chi-square distribution.

The noncentral χ^2 distribution is a generalization of the χ^2 distribution.

Note: New code should use the *~numpy.random.Generator.noncentral_chisquare* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **df** (*float or array_like of floats*) – Degrees of freedom, must be > 0.

  Changed in version 1.10.0: Earlier NumPy versions required dfnum > 1.

- **nonc** (*float or array_like of floats*) – Non-centrality, must be non-negative.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` and `nonc` are both scalars. Otherwise, `np.broadcast(df, nonc).size` samples are drawn.

Returns
out – Drawn samples from the parameterized noncentral chi-square distribution.

Return type
ndarray or scalar

See also:

`random.Generator.noncentral_chisquare`
which should be used for new code.

Notes

The probability density function for the noncentral Chi-square distribution is

$$P(x;df,nonc) = \sum_{i=0}^{\infty} \frac{e^{-nonc/2}(nonc/2)^i}{i!} P_{Y_{df+2i}}(x),$$

where Y_q is the Chi-square with q degrees of freedom.

References

Examples

Draw values from the distribution and plot the histogram

```
>>> import matplotlib.pyplot as plt
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```

Draw values from a noncentral chisquare with very small noncentrality, and compare to a chisquare.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, .0000001, 100000),
...                   bins=np.arange(0., 25, .1), density=True)
>>> values2 = plt.hist(np.random.chisquare(3, 100000),
...                    bins=np.arange(0., 25, .1), density=True)
>>> plt.plot(values[1][0:-1], values[0]-values2[0], 'ob')
>>> plt.show()
```

Demonstrate how large values of non-centrality lead to a more symmetric distribution.

```
>>> plt.figure()
>>> values = plt.hist(np.random.noncentral_chisquare(3, 20, 100000),
...                   bins=200, density=True)
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**noncentral_f**(*dfnum*, *dfden*, *nonc*, *size=None*)

Draw samples from the noncentral F distribution.

Samples are drawn from an F distribution with specified parameters, *dfnum* (degrees of freedom in numerator) and *dfden* (degrees of freedom in denominator), where both parameters > 1. *nonc* is the non-centrality parameter.

Note: New code should use the *~numpy.random.Generator.noncentral_f* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **dfnum** (*float or array_like of floats*) – Numerator degrees of freedom, must be > 0.

  Changed in version 1.14.0: Earlier NumPy versions required dfnum > 1.

- **dfden** (*float or array_like of floats*) – Denominator degrees of freedom, must be > 0.

- **nonc** (*float or array_like of floats*) – Non-centrality parameter, the sum of the squares of the numerator means, must be >= 0.

- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `dfnum`, `dfden`, and `nonc` are all scalars. Otherwise, `np.broadcast(dfnum, dfden, nonc).size` samples are drawn.

Returns

out – Drawn samples from the parameterized noncentral Fisher distribution.

Return type

ndarray or scalar

See also:

`random.Generator.noncentral_f`
which should be used for new code.

Notes

When calculating the power of an experiment (power = probability of rejecting the null hypothesis when a specific alternative is true) the non-central F statistic becomes important. When the null hypothesis is true, the F statistic follows a central F distribution. When the null hypothesis is not true, then it follows a non-central F statistic.

References

Examples

In a study, testing for a specific alternative to the null hypothesis requires use of the Noncentral F distribution. We need to calculate the area in the tail of the distribution that exceeds the value of the F distribution for the null hypothesis. We'll plot the two probability distributions for comparison.

```
>>> dfnum = 3 # between group deg of freedom
>>> dfden = 20 # within groups degrees of freedom
>>> nonc = 3.0
>>> nc_vals = np.random.noncentral_f(dfnum, dfden, nonc, 1000000)
>>> NF = np.histogram(nc_vals, bins=50, density=True)
>>> c_vals = np.random.f(dfnum, dfden, 1000000)
>>> F = np.histogram(c_vals, bins=50, density=True)
>>> import matplotlib.pyplot as plt
>>> plt.plot(F[1][1:], F[0])
>>> plt.plot(NF[1][1:], NF[0])
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**normal**(*loc=0.0*, *scale=1.0*, *size=None*)

Draw random samples from a normal (Gaussian) distribution.

The probability density function of the normal distribution, first derived by De Moivre and 200 years later by both Gauss and Laplace independently [2]_, is often called the bell curve because of its characteristic shape (see the example below).

The normal distributions occurs often in nature. For example, it describes the commonly occurring distribution of samples influenced by a large number of tiny, random disturbances, each with its own unique distribution [2]_.

Note: New code should use the *~numpy.random.Generator.normal* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **loc** (*float or array_like of floats*) – Mean ("centre") of the distribution.
- **scale** (*float or array_like of floats*) – Standard deviation (spread or "width") of the distribution. Must be non-negative.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `loc` and `scale` are both scalars. Otherwise, `np.broadcast(loc, scale).size` samples are drawn.

Returns

out – Drawn samples from the parameterized normal distribution.

Return type

ndarray or scalar

See also:

scipy.stats.norm
probability density function, distribution or cumulative density function, etc.

random.Generator.normal
which should be used for new code.

Notes

The probability density for the Gaussian distribution is

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}},$$

where μ is the mean and σ the standard deviation. The square of the standard deviation, σ^2, is called the variance.

The function has its peak at the mean, and its "spread" increases with the standard deviation (the function reaches 0.607 times its maximum at $x + \sigma$ and $x - \sigma$ [2]_). This implies that normal is more likely to return samples lying close to the mean, rather than those far away.

References

Examples

Draw samples from the distribution:

```
>>> mu, sigma = 0, 0.1 # mean and standard deviation
>>> s = np.random.normal(mu, sigma, 1000)
```

Verify the mean and the variance:

```
>>> abs(mu - np.mean(s))
0.0  # may vary
```

```
>>> abs(sigma - np.std(s, ddof=1))
0.1  # may vary
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 30, density=True)
>>> plt.plot(bins, 1/(sigma * np.sqrt(2 * np.pi)) *
...                np.exp( - (bins - mu)**2 / (2 * sigma**2) ),
...          linewidth=2, color='r')
>>> plt.show()
```

Two-by-four array of samples from the normal distribution with mean 3 and standard deviation 2.5:

```
>>> np.random.normal(3, 2.5, size=(2, 4))
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

metilda.controllers.pitch_art_wizard.**pareto**(*a*, *size=None*)

Draw samples from a Pareto II or Lomax distribution with specified shape.

The Lomax or Pareto II distribution is a shifted Pareto distribution. The classical Pareto distribution can be obtained from the Lomax distribution by adding 1 and multiplying by the scale parameter `m` (see Notes). The smallest value of the Lomax distribution is zero while for the classical Pareto distribution it is `mu`, where the standard Pareto distribution has location `mu = 1`. Lomax can also be considered as a simplified version of the Generalized Pareto distribution (available in SciPy), with the scale set to one and the location set to zero.

The Pareto distribution must be greater than zero, and is unbounded above. It is also known as the "80-20 rule". In this distribution, 80 percent of the weights are in the lowest 20 percent of the range, while the other 20 percent fill the remaining 80 percent of the range.

Note: New code should use the *~numpy.random.Generator.pareto* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **a** (*float or array_like of floats*) – Shape of the distribution. Must be positive.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

Returns

out – Drawn samples from the parameterized Pareto distribution.

Return type

ndarray or scalar

See also:

`scipy.stats.lomax`
probability density function, distribution or cumulative density function, etc.

`scipy.stats.genpareto`
probability density function, distribution or cumulative density function, etc.

`random.Generator.pareto`
which should be used for new code.

Notes

The probability density for the Pareto distribution is

$$p(x) = \frac{am^a}{x^{a+1}}$$

where a is the shape and m the scale.

The Pareto distribution, named after the Italian economist Vilfredo Pareto, is a power law probability distribution useful in many real world problems. Outside the field of economics it is generally referred to as the Bradford distribution. Pareto developed the distribution to describe the distribution of wealth in an economy. It has also found use in insurance, web page access statistics, oil field sizes, and many other problems, including the download frequency for projects in Sourceforge [1]_. It is one of the so-called "fat-tailed" distributions.

References

Examples

Draw samples from the distribution:

```
>>> a, m = 3., 2.  # shape and mode
>>> s = (np.random.pareto(a, 1000) + 1) * m
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, _ = plt.hist(s, 100, density=True)
>>> fit = a*m**a / bins**(a+1)
>>> plt.plot(bins, max(count)*fit/max(fit), linewidth=2, color='r')
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**permutation**(*x*)

Randomly permute a sequence, or return a permuted range.

If *x* is a multi-dimensional array, it is only shuffled along its first index.

Note: New code should use the *~numpy.random.Generator.permutation* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

x (*int or array_like*) – If *x* is an integer, randomly permute `np.arange(x)`. If *x* is an array, make a copy and shuffle the elements randomly.

Returns

out – Permuted sequence or array range.

Return type

ndarray

See also:

random.Generator.permutation

which should be used for new code.

Examples

```
>>> np.random.permutation(10)
array([1, 7, 4, 3, 0, 9, 2, 5, 8, 6]) # random
```

```
>>> np.random.permutation([1, 4, 9, 12, 15])
array([15,  1,  9,  4, 12]) # random
```

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.permutation(arr)
array([[6, 7, 8], # random
```

(continues on next page)

(continued from previous page)

```
       [0, 1, 2],
       [3, 4, 5]])
```

metilda.controllers.pitch_art_wizard.**pitchValueAtTime**(*sound*, *time*)

metilda.controllers.pitch_art_wizard.**pitchValueInFrame**(*sound*, *frame*)

metilda.controllers.pitch_art_wizard.**pointProcessGetJitter**(*sound*, *start*, *end*)

metilda.controllers.pitch_art_wizard.**pointProcessGetNumPeriods**(*sound*, *start*, *end*)

metilda.controllers.pitch_art_wizard.**pointProcessGetNumPoints**(*sound*)

metilda.controllers.pitch_art_wizard.**poisson**(*lam=1.0*, *size=None*)

Draw samples from a Poisson distribution.

The Poisson distribution is the limit of the binomial distribution for large N.

Note: New code should use the *~numpy.random.Generator.poisson* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **lam** (*float or array_like of floats*) – Expected number of events occurring in a fixed-time interval, must be >= 0. A sequence must be broadcastable over the requested size.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `lam` is a scalar. Otherwise, `np.array(lam).size` samples are drawn.

Returns

out – Drawn samples from the parameterized Poisson distribution.

Return type

ndarray or scalar

See also:

random.Generator.poisson

which should be used for new code.

Notes

The Poisson distribution

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!}$$

For events with an expected separation λ the Poisson distribution $f(k;\lambda)$ describes the probability of k events occurring within the observed interval λ.

Because the output is limited to the range of the C int64 type, a ValueError is raised when *lam* is within 10 sigma of the maximum representable value.

References

Examples

Draw samples from the distribution:

```
>>> import numpy as np
>>> s = np.random.poisson(5, 10000)
```

Display histogram of the sample:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 14, density=True)
>>> plt.show()
```

Draw each 100 values for lambda 100 and 500:

```
>>> s = np.random.poisson(lam=(100., 500.), size=(100, 2))
```

metilda.controllers.pitch_art_wizard.**power**(*a*, *size=None*)

Draws samples in [0, 1] from a power distribution with positive exponent a - 1.

Also known as the power function distribution.

Note: New code should use the *~numpy.random.Generator.power* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **a** (*float or array_like of floats*) – Parameter of the distribution. Must be non-negative.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

Returns

out – Drawn samples from the parameterized power distribution.

Return type

ndarray or scalar

Raises

ValueError – If a <= 0.

See also:

random.Generator.power

which should be used for new code.

Notes

The probability density function is

$$P(x; a) = ax^{a-1}, 0 \leq x \leq 1, a > 0.$$

The power function distribution is just the inverse of the Pareto distribution. It may also be seen as a special case of the Beta distribution.

It is used, for example, in modeling the over-reporting of insurance claims.

References

Examples

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> samples = 1000
>>> s = np.random.power(a, samples)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, bins=30)
>>> x = np.linspace(0, 1, 100)
>>> y = a*x**(a-1.)
>>> normed_y = samples*np.diff(bins)[0]*y
>>> plt.plot(x, normed_y)
>>> plt.show()
```

Compare the power function distribution to the inverse of the Pareto.

```
>>> from scipy import stats
>>> rvs = np.random.power(5, 1000000)
>>> rvsp = np.random.pareto(5, 1000000)
>>> xx = np.linspace(0,1,100)
>>> powpdf = stats.powerlaw.pdf(xx,5)
```

```
>>> plt.figure()
>>> plt.hist(rvs, bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('np.random.power(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of 1 + np.random.pareto(5)')
```

```
>>> plt.figure()
>>> plt.hist(1./(1.+rvsp), bins=50, density=True)
>>> plt.plot(xx,powpdf,'r-')
>>> plt.title('inverse of stats.pareto(5)')
```

metilda.controllers.pitch_art_wizard.**rand**(*d0*, *d1*, ..., *dn*)

Random values in a given shape.

Note: This is a convenience function for users porting code from Matlab, and wraps *random_sample*. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like *numpy.zeros* and *numpy.ones*.

Create an array of the given shape and populate it with random samples from a uniform distribution over `[0, 1)`.

Parameters

- **d0** (*int, optional*) – The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.
- **d1** (*int, optional*) – The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.
- **...** (*int, optional*) – The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.
- **dn** (*int, optional*) – The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

Returns

out – Random values.

Return type

ndarray, shape `(d0, d1, ..., dn)`

See also:

random

Examples

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618],  #random
       [ 0.37601032,  0.25528411],  #random
       [ 0.49313049,  0.94909878]]) #random
```

metilda.controllers.pitch_art_wizard.**randint**(*low*, *high=None*, *size=None*, *dtype=int*)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

Note: New code should use the *~numpy.random.Generator.randint* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **low** (*int or array-like of ints*) – Lowest (signed) integers to be drawn from the distribution (unless `high=None`, in which case this parameter is one above the *highest* such integer).

- **high** (*int or array-like of ints, optional*) – If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`). If array-like, must contain integer values
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.
- **dtype** (*dtype, optional*) – Desired dtype of the result. Byteorder must be native. The default value is int.

  New in version 1.11.0.

Returns

out – *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

Return type

int or ndarray of ints

See also:

random_integers

similar to *randint*, only for the closed interval [*low*, *high*], and 1 is the lowest value if *high* is omitted.

random.Generator.integers

which should be used for new code.

Examples

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0]) # random
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1], # random
       [3, 2, 2, 0]])
```

Generate a 1 x 3 array with 3 different upper bounds

```
>>> np.random.randint(1, [3, 5, 10])
array([2, 2, 9]) # random
```

Generate a 1 by 3 array with 3 different lower bounds

```
>>> np.random.randint([1, 5, 7], 10)
array([9, 8, 7]) # random
```

Generate a 2 by 4 array using broadcasting with dtype of uint8

```
>>> np.random.randint([1, 3, 5, 7], [[10], [20]], dtype=np.uint8)
array([[ 8,  6,  9,  7], # random
       [ 1, 16,  9, 12]], dtype=uint8)
```

metilda.controllers.pitch_art_wizard.**randn**(*d0*, *d1*, ..., *dn*)

Return a sample (or samples) from the "standard normal" distribution.

Note: This is a convenience function for users porting code from Matlab, and wraps *standard_normal*. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like *numpy.zeros* and *numpy.ones*.

Note: New code should use the *~numpy.random.Generator.standard_normal* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

If positive int_like arguments are provided, *randn* generates an array of shape `(d0, d1, ..., dn)`, filled with random floats sampled from a univariate "normal" (Gaussian) distribution of mean 0 and variance 1. A single float randomly sampled from the distribution is returned if no argument is provided.

Parameters

- **d0** (*int, optional*) – The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.
- **d1** (*int, optional*) – The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.
- **...** (*int, optional*) – The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.
- **dn** (*int, optional*) – The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

Returns

Z – A `(d0, d1, ..., dn)`-shaped array of floating-point samples from the standard normal distribution, or a single such float if no parameters were supplied.

Return type

ndarray or float

See also:

standard_normal

Similar, but takes a tuple as its argument.

normal

Also accepts mu and sigma arguments.

random.Generator.standard_normal

which should be used for new code.

Notes

For random samples from the normal distribution with mean `mu` and standard deviation `sigma`, use:

```
sigma * np.random.randn(...) + mu
```

Examples

```
>>> np.random.randn()
2.1923875335537315  # random
```

Two-by-four array of samples from the normal distribution with mean 3 and standard deviation 2.5:

```
>>> 3 + 2.5 * np.random.randn(2, 4)
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

metilda.controllers.pitch_art_wizard.**random**(*size=None*)

Return random floats in the half-open interval [0.0, 1.0). Alias for *random_sample* to ease forward-porting to the new random API.

metilda.controllers.pitch_art_wizard.**random_integers**(*low*, *high=None*, *size=None*)

Random integers of type *np.int_* between *low* and *high*, inclusive.

Return random integers of type *np.int_* from the "discrete uniform" distribution in the closed interval [*low*, *high*]. If *high* is None (the default), then results are from [1, *low*]. The *np.int_* type translates to the C long integer type and its precision is platform dependent.

This function has been deprecated. Use randint instead.

Deprecated since version 1.11.0.

Parameters

- **low** (*int*) – Lowest (signed) integer to be drawn from the distribution (unless `high=None`, in which case this parameter is the *highest* such integer).
- **high** (*int, optional*) – If provided, the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`).
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out – *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

Return type

int or ndarray of ints

See also:

randint

Similar to *random_integers*, only for the half-open interval [*low*, *high*), and 0 is the lowest value if *high* is omitted.

Notes

To sample from N evenly spaced floating-point numbers between a and b, use:

```
a + (b - a) * (np.random.random_integers(N) - 1) / (N - 1.)
```

Examples

```
>>> np.random.random_integers(5)
4 # random
>>> type(np.random.random_integers(5))
<class 'numpy.int64'>
>>> np.random.random_integers(5, size=(3,2))
array([[5, 4], # random
       [3, 3],
       [4, 5]])
```

Choose five random numbers from the set of five evenly-spaced numbers between 0 and 2.5, inclusive (*i.e.*, from the set $0, 5/8, 10/8, 15/8, 20/8$):

```
>>> 2.5 * (np.random.random_integers(5, size=(5,)) - 1) / 4.
array([ 0.625,  1.25 ,  0.625,  0.625,  2.5  ]) # random
```

Roll two six sided dice 1000 times and sum the results:

```
>>> d1 = np.random.random_integers(1, 6, 1000)
>>> d2 = np.random.random_integers(1, 6, 1000)
>>> dsums = d1 + d2
```

Display results as a histogram:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(dsums, 11, density=True)
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**random_sample**(*size=None*)

Return random floats in the half-open interval [0.0, 1.0).

Results are from the "continuous uniform" distribution over the stated interval. To sample $Unif[a, b), b > a$ multiply the output of *random_sample* by *(b-a)* and add *a*:

```
(b - a) * random_sample() + a
```

Note: New code should use the *~numpy.random.Generator.random* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

size (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns
out – Array of random floats of shape *size* (unless `size=None`, in which case a single float is returned).

Return type
float or ndarray of floats

See also:

`random.Generator.random`
which should be used for new code.

Examples

```
>>> np.random.random_sample()
0.47108547995356098 # random
>>> type(np.random.random_sample())
<class 'float'>
>>> np.random.random_sample((5,))
array([ 0.30220482,  0.86820401,  0.1654503 ,  0.11659149,  0.54323428]) # random
```

Three-by-two array of random numbers from [-5, 0):

```
>>> 5 * np.random.random_sample((3, 2)) - 5
array([[-3.99149989, -0.52338984], # random
       [-2.99091858, -0.79479508],
       [-1.23204345, -1.75224494]])
```

`metilda.controllers.pitch_art_wizard.`**`rayleigh`**(*scale=1.0*, *size=None*)

Draw samples from a Rayleigh distribution.

The χ and Weibull distributions are generalizations of the Rayleigh.

Note: New code should use the *~numpy.random.Generator.rayleigh* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **`scale`** (*`float or array_like of floats, optional`*) – Scale, also equals the mode. Must be non-negative. Default is 1.
- **`size`** (*`int or tuple of ints, optional`*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `scale` is a scalar. Otherwise, `np.array(scale).size` samples are drawn.

Returns
out – Drawn samples from the parameterized Rayleigh distribution.

Return type
ndarray or scalar

See also:

`random.Generator.rayleigh`
which should be used for new code.

Notes

The probability density function for the Rayleigh distribution is

$$P(x; scale) = \frac{x}{scale^2} e^{\frac{-x^2}{2 \cdot scale^2}}$$

The Rayleigh distribution would arise, for example, if the East and North components of the wind velocity had identical zero-mean Gaussian distributions. Then the wind speed would have a Rayleigh distribution.

References

Examples

Draw values from the distribution and plot the histogram

```
>>> from matplotlib.pyplot import hist
>>> values = hist(np.random.rayleigh(3, 100000), bins=200, density=True)
```

Wave heights tend to follow a Rayleigh distribution. If the mean wave height is 1 meter, what fraction of waves are likely to be larger than 3 meters?

```
>>> meanvalue = 1
>>> modevalue = np.sqrt(2 / np.pi) * meanvalue
>>> s = np.random.rayleigh(modevalue, 1000000)
```

The percentage of waves larger than 3 meters is:

```
>>> 100.*sum(s>3)/1000000.
0.087300000000000003 # random
```

metilda.controllers.pitch_art_wizard.**set_state**(*state*)

Set the internal state of the generator from a tuple.

For use if one has reason to manually (re-)set the internal state of the bit generator used by the RandomState instance. By default, RandomState uses the "Mersenne Twister"[1]_ pseudo-random number generating algorithm.

Parameters

state (*{tuple(str, ndarray of 624 uints, int, int, float), dict}*) – The *state* tuple has the following items:

1. the string 'MT19937', specifying the Mersenne Twister algorithm.
2. a 1-D array of 624 unsigned integers `keys`.
3. an integer `pos`.
4. an integer `has_gauss`.
5. a float `cached_gaussian`.

If state is a dictionary, it is directly set using the BitGenerators *state* property.

Returns

out – Returns 'None' on success.

Return type

None

See also:

get_state

Notes

set_state and *get_state* are not needed to work with any of the random distributions in NumPy. If the internal state is manually altered, the user should know exactly what he/she is doing.

For backwards compatibility, the form (str, array of 624 uints, int) is also accepted although it is missing some information about the cached Gaussian value: `state = ('MT19937', keys, pos)`.

References

metilda.controllers.pitch_art_wizard.**share_file**()

metilda.controllers.pitch_art_wizard.**shuffle**(*x*)

Modify a sequence in-place by shuffling its contents.

This function only shuffles the array along the first axis of a multi-dimensional array. The order of sub-arrays is changed but their contents remains the same.

Note: New code should use the *~numpy.random.Generator.shuffle* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

x (*ndarray or MutableSequence*) – The array, list or mutable sequence to be shuffled.

Return type

None

See also:

random.Generator.shuffle
which should be used for new code.

Examples

```
>>> arr = np.arange(10)
>>> np.random.shuffle(arr)
>>> arr
[1 7 5 2 9 4 3 6 0 8] # random
```

Multi-dimensional arrays are only shuffled along the first axis:

```
>>> arr = np.arange(9).reshape((3, 3))
>>> np.random.shuffle(arr)
>>> arr
array([[3, 4, 5], # random
       [6, 7, 8],
       [0, 1, 2]])
```

metilda.controllers.pitch_art_wizard.**sound_length**(*upload_id*)

metilda.controllers.pitch_art_wizard.**spectrumFrequencyBounds**(*sound*)

metilda.controllers.pitch_art_wizard.**standard_cauchy**(*size=None*)

Draw samples from a standard Cauchy distribution with mode = 0.

Also known as the Lorentz distribution.

Note: New code should use the *~numpy.random.Generator.standard_cauchy* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

size (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

samples – The drawn samples.

Return type

ndarray or scalar

See also:

random.Generator.standard_cauchy
which should be used for new code.

Notes

The probability density function for the full Cauchy distribution is

$$P(x; x_0, \gamma) = \frac{1}{\pi\gamma\left[1 + (\frac{x-x_0}{\gamma})^2\right]}$$

and the Standard Cauchy distribution just sets $x_0 = 0$ and $\gamma = 1$

The Cauchy distribution arises in the solution to the driven harmonic oscillator problem, and also describes spectral line broadening. It also describes the distribution of values at which a line tilted at a random angle will cut the x axis.

When studying hypothesis tests that assume normality, seeing how the tests perform on data from a Cauchy distribution is a good indicator of their sensitivity to a heavy-tailed distribution, since the Cauchy looks very much like a Gaussian distribution, but with heavier tails.

References

Examples

Draw samples and plot the distribution:

```
>>> import matplotlib.pyplot as plt
>>> s = np.random.standard_cauchy(1000000)
>>> s = s[(s>-25) & (s<25)]  # truncate distribution so it plots well
>>> plt.hist(s, bins=100)
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**standard_exponential**(*size=None*)

Draw samples from the standard exponential distribution.

standard_exponential is identical to the exponential distribution with a scale parameter of 1.

Note: New code should use the *~numpy.random.Generator.standard_exponential* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

size (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.

Returns

out – Drawn samples.

Return type

float or ndarray

See also:

random.Generator.standard_exponential

which should be used for new code.

Examples

Output a 3x8000 array:

```
>>> n = np.random.standard_exponential((3, 8000))
```

metilda.controllers.pitch_art_wizard.**standard_gamma**(*shape*, *size=None*)

Draw samples from a standard Gamma distribution.

Samples are drawn from a Gamma distribution with specified parameters, shape (sometimes designated "k") and scale=1.

Note: New code should use the *~numpy.random.Generator.standard_gamma* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **shape** (*float or array_like of floats*) – Parameter, must be non-negative.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `shape` is a scalar. Otherwise, `np.array(shape).size` samples are drawn.

Returns
out – Drawn samples from the parameterized standard gamma distribution.

Return type
ndarray or scalar

See also:

scipy.stats.gamma
probability density function, distribution or cumulative density function, etc.

random.Generator.standard_gamma
which should be used for new code.

Notes

The probability density for the Gamma distribution is

$$p(x) = x^{k-1}\frac{e^{-x/\theta}}{\theta^k \Gamma(k)},$$

where k is the shape and θ the scale, and Γ is the Gamma function.

The Gamma distribution is often used to model the times to failure of electronic components, and arises naturally in processes for which the waiting times between Poisson distributed events are relevant.

References

Examples

Draw samples from the distribution:

```
>>> shape, scale = 2., 1. # mean and width
>>> s = np.random.standard_gamma(shape, 1000000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> import scipy.special as sps
>>> count, bins, ignored = plt.hist(s, 50, density=True)
>>> y = bins**(shape-1) * ((np.exp(-bins/scale))/
...                        (sps.gamma(shape) * scale**shape))
>>> plt.plot(bins, y, linewidth=2, color='r')
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**standard_normal**(*size=None*)

Draw samples from a standard Normal distribution (mean=0, stdev=1).

Note: New code should use the *~numpy.random.Generator.standard_normal* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters
size (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., (m,

n, k), then m * n * k samples are drawn. Default is None, in which case a single value is returned.

Returns

out – A floating-point array of shape size of drawn samples, or a single sample if size was not specified.

Return type

float or ndarray

See also:

normal

Equivalent function with additional loc and scale arguments for setting the mean and standard deviation.

random.Generator.standard_normal

which should be used for new code.

Notes

For random samples from the normal distribution with mean mu and standard deviation sigma, use one of:

```
mu + sigma * np.random.standard_normal(size=...)
np.random.normal(mu, sigma, size=...)
```

Examples

```
>>> np.random.standard_normal()
2.1923875335537315 #random
```

```
>>> s = np.random.standard_normal(8000)
>>> s
array([ 0.6888893 ,  0.78096262, -0.89086505, ...,  0.49876311,  # random
       -0.38672696, -0.4685006 ])                                # random
>>> s.shape
(8000,)
>>> s = np.random.standard_normal(size=(3, 4, 2))
>>> s.shape
(3, 4, 2)
```

Two-by-four array of samples from the normal distribution with mean 3 and standard deviation 2.5:

```
>>> 3 + 2.5 * np.random.standard_normal(size=(2, 4))
array([[-4.49401501,  4.00950034, -1.81814867,  7.29718677],   # random
       [ 0.39924804,  4.68456316,  4.99394529,  4.84057254]])  # random
```

metilda.controllers.pitch_art_wizard.**standard_t**(*df*, *size=None*)

Draw samples from a standard Student's t distribution with *df* degrees of freedom.

A special case of the hyperbolic distribution. As *df* gets large, the result resembles that of the standard normal distribution (*standard_normal*).

Note: New code should use the *~numpy.random.Generator.standard_t* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **df** (*float or array_like of floats*) – Degrees of freedom, must be > 0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `df` is a scalar. Otherwise, `np.array(df).size` samples are drawn.

Returns

out – Drawn samples from the parameterized standard Student's t distribution.

Return type

ndarray or scalar

See also:

`random.Generator.standard_t`
which should be used for new code.

Notes

The probability density function for the t distribution is

$$P(x, df) = \frac{\Gamma(\frac{df+1}{2})}{\sqrt{\pi df}\,\Gamma(\frac{df}{2})}\left(1 + \frac{x^2}{df}\right)^{-(df+1)/2}$$

The t test is based on an assumption that the data come from a Normal distribution. The t test provides a way to test whether the sample mean (that is the mean calculated from the data) is a good estimate of the true mean.

The derivation of the t-distribution was first published in 1908 by William Gosset while working for the Guinness Brewery in Dublin. Due to proprietary issues, he had to publish under a pseudonym, and so he used the name Student.

References

Examples

From Dalgaard page 83 [1]_, suppose the daily energy intake for 11 women in kilojoules (kJ) is:

```
>>> intake = np.array([5260., 5470, 5640, 6180, 6390, 6515, 6805, 7515, \
...                    7515, 8230, 8770])
```

Does their energy intake deviate systematically from the recommended value of 7725 kJ? Our null hypothesis will be the absence of deviation, and the alternate hypothesis will be the presence of an effect that could be either positive or negative, hence making our test 2-tailed.

Because we are estimating the mean and we have N=11 values in our sample, we have N-1=10 degrees of freedom. We set our significance level to 95% and compute the t statistic using the empirical mean and empirical standard deviation of our intake. We use a ddof of 1 to base the computation of our empirical standard deviation on an unbiased estimate of the variance (note: the final estimate is not unbiased due to the concave nature of the square root).

```
>>> np.mean(intake)
6753.636363636364
>>> intake.std(ddof=1)
1142.1232221373727
>>> t = (np.mean(intake)-7725)/(intake.std(ddof=1)/np.sqrt(len(intake)))
>>> t
-2.8207540608310198
```

We draw 1000000 samples from Student's t distribution with the adequate degrees of freedom.

```
>>> import matplotlib.pyplot as plt
>>> s = np.random.standard_t(10, size=1000000)
>>> h = plt.hist(s, bins=100, density=True)
```

Does our t statistic land in one of the two critical regions found at both tails of the distribution?

```
>>> np.sum(np.abs(t) < np.abs(s)) / float(len(s))
0.018318  #random < 0.05, statistic is in critical region
```

The probability value for this 2-tailed test is about 1.83%, which is lower than the 5% pre-determined significance threshold.

Therefore, the probability of observing values as extreme as our intake conditionally on the null hypothesis being true is too low, and we reject the null hypothesis of no deviation.

metilda.controllers.pitch_art_wizard.**triangular**(*left*, *mode*, *right*, *size=None*)

Draw samples from the triangular distribution over the interval `[left, right]`.

The triangular distribution is a continuous probability distribution with lower limit left, peak at mode, and upper limit right. Unlike the other distributions, these parameters directly define the shape of the pdf.

Note: New code should use the *~numpy.random.Generator.triangular* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **left** (*float or array_like of floats*) – Lower limit.
- **mode** (*float or array_like of floats*) – The value where the peak of the distribution occurs. The value must fulfill the condition `left <= mode <= right`.
- **right** (*float or array_like of floats*) – Upper limit, must be larger than *left*.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `left`, `mode`, and `right` are all scalars. Otherwise, `np.broadcast(left, mode, right).size` samples are drawn.

Returns

out – Drawn samples from the parameterized triangular distribution.

Return type

ndarray or scalar

See also:

random.Generator.triangular
which should be used for new code.

Notes

The probability density function for the triangular distribution is

$$P(x;l,m,r) = \begin{cases} \frac{2(x-l)}{(r-l)(m-l)} & \text{for } l \leq x \leq m, \\ \frac{2(r-x)}{(r-l)(r-m)} & \text{for } m \leq x \leq r, \\ 0 & \text{otherwise.} \end{cases}$$

The triangular distribution is often used in ill-defined problems where the underlying distribution is not known, but some knowledge of the limits and mode exists. Often it is used in simulations.

References

Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.triangular(-3, 0, 8, 100000), bins=200,
...              density=True)
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**uniform**(*low=0.0*, *high=1.0*, *size=None*)

Draw samples from a uniform distribution.

Samples are uniformly distributed over the half-open interval `[low, high)` (includes low, but excludes high). In other words, any value within the given interval is equally likely to be drawn by *uniform*.

Note: New code should use the *~numpy.random.Generator.uniform* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **low** (*float or array_like of floats, optional*) – Lower boundary of the output interval. All values generated will be greater than or equal to low. The default value is 0.
- **high** (*float or array_like of floats*) – Upper boundary of the output interval. All values generated will be less than or equal to high. The high limit may be included in the returned array of floats due to floating-point rounding in the equation `low + (high-low) * random_sample()`. The default value is 1.0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `low` and `high` are both scalars. Otherwise, `np.broadcast(low, high).size` samples are drawn.

Returns
out – Drawn samples from the parameterized uniform distribution.

Return type
ndarray or scalar

See also:

randint
Discrete uniform distribution, yielding integers.

random_integers
Discrete uniform distribution over the closed interval `[low, high]`.

random_sample
Floats uniformly distributed over `[0, 1)`.

random
Alias for *random_sample*.

rand
Convenience function that accepts dimensions as input, e.g., `rand(2,2)` would generate a 2-by-2 array of floats, uniformly distributed over `[0, 1)`.

random.Generator.uniform
which should be used for new code.

Notes

The probability density function of the uniform distribution is

$$p(x) = \frac{1}{b - a}$$

anywhere within the interval `[a, b)`, and zero elsewhere.

When `high == low`, values of `low` will be returned. If `high < low`, the results are officially undefined and may eventually raise an error, i.e. do not rely on this function to behave when passed arguments satisfying that inequality condition. The `high` limit may be included in the returned array of floats due to floating-point rounding in the equation `low + (high-low) * random_sample()`. For example:

```
>>> x = np.float32(5*0.99999999)
>>> x
5.0
```

Examples

Draw samples from the distribution:

```
>>> s = np.random.uniform(-1,0,1000)
```

All values are within the given interval:

```
>>> np.all(s >= -1)
True
>>> np.all(s < 0)
True
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> count, bins, ignored = plt.hist(s, 15, density=True)
>>> plt.plot(bins, np.ones_like(bins), linewidth=2, color='r')
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**update_analysis**()

metilda.controllers.pitch_art_wizard.**update_db_user**()

metilda.controllers.pitch_art_wizard.**update_user_from_admin**()

metilda.controllers.pitch_art_wizard.**vonmises**(*mu*, *kappa*, *size=None*)

Draw samples from a von Mises distribution.

Samples are drawn from a von Mises distribution with specified mode (mu) and dispersion (kappa), on the interval [-pi, pi].

The von Mises distribution (also known as the circular normal distribution) is a continuous probability distribution on the unit circle. It may be thought of as the circular analogue of the normal distribution.

Note: New code should use the *~numpy.random.Generator.vonmises* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **mu** (*float or array_like of floats*) – Mode ("center") of the distribution.
- **kappa** (*float or array_like of floats*) – Dispersion of the distribution, has to be >=0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mu` and `kappa` are both scalars. Otherwise, `np.broadcast(mu, kappa).size` samples are drawn.

Returns

out – Drawn samples from the parameterized von Mises distribution.

Return type

ndarray or scalar

See also:

`scipy.stats.vonmises`
probability density function, distribution, or cumulative density function, etc.

`random.Generator.vonmises`
which should be used for new code.

Notes

The probability density for the von Mises distribution is

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

where μ is the mode and κ the dispersion, and $I_0(\kappa)$ is the modified Bessel function of order 0.

The von Mises is named for Richard Edler von Mises, who was born in Austria-Hungary, in what is now the Ukraine. He fled to the United States in 1939 and became a professor at Harvard. He worked in probability theory, aerodynamics, fluid mechanics, and philosophy of science.

References

Examples

Draw samples from the distribution:

```
>>> mu, kappa = 0.0, 4.0 # mean and dispersion
>>> s = np.random.vonmises(mu, kappa, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy.special import i0
>>> plt.hist(s, 50, density=True)
>>> x = np.linspace(-np.pi, np.pi, num=51)
>>> y = np.exp(kappa*np.cos(x-mu))/(2*np.pi*i0(kappa))
>>> plt.plot(x, y, linewidth=2, color='r')
>>> plt.show()
```

metilda.controllers.pitch_art_wizard.**wald**(*mean*, *scale*, *size=None*)

Draw samples from a Wald, or inverse Gaussian, distribution.

As the scale approaches infinity, the distribution becomes more like a Gaussian. Some references claim that the Wald is an inverse Gaussian with mean equal to 1, but this is by no means universal.

The inverse Gaussian distribution was first studied in relationship to Brownian motion. In 1956 M.C.K. Tweedie used the name inverse Gaussian because there is an inverse relationship between the time to cover a unit distance and distance covered in unit time.

Note: New code should use the *~numpy.random.Generator.wald* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **mean** (*float or array_like of floats*) – Distribution mean, must be > 0.
- **scale** (*float or array_like of floats*) – Scale parameter, must be > 0.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `mean` and `scale` are both scalars. Otherwise, `np.broadcast(mean, scale).size` samples are drawn.

Returns
out – Drawn samples from the parameterized Wald distribution.

Return type
ndarray or scalar

See also:

`random.Generator.wald`
which should be used for new code.

Notes

The probability density function for the Wald distribution is

$$P(x; mean, scale) = \sqrt{\frac{scale}{2\pi x^3}} e^{\frac{-scale(x-mean)^2}{2 \cdot mean^2 x}}$$

As noted above the inverse Gaussian distribution first arise from attempts to model Brownian motion. It is also a competitor to the Weibull for use in reliability modeling and modeling stock returns and interest rate processes.

References

Examples

Draw values from the distribution and plot the histogram:

```
>>> import matplotlib.pyplot as plt
>>> h = plt.hist(np.random.wald(3, 2, 100000), bins=200, density=True)
>>> plt.show()
```

`metilda.controllers.pitch_art_wizard.`**`weibull`**(*a*, *size=None*)

Draw samples from a Weibull distribution.

Draw samples from a 1-parameter Weibull distribution with the given shape parameter a.

$$X = (-ln(U))^{1/a}$$

Here, U is drawn from the uniform distribution over (0,1].

The more common 2-parameter Weibull, including a scale parameter λ is just $X = \lambda(-ln(U))^{1/a}$.

Note: New code should use the *~numpy.random.Generator.weibull* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **a** (*float or array_like of floats*) – Shape parameter of the distribution. Must be nonnegative.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

Returns
out – Drawn samples from the parameterized Weibull distribution.

Return type
ndarray or scalar

See also:

`scipy.stats.weibull_max`, `scipy.stats.weibull_min`, `scipy.stats.genextreme`, *gumbel*

random.Generator.weibull
which should be used for new code.

Notes

The Weibull (or Type III asymptotic extreme value distribution for smallest values, SEV Type III, or Rosin-Rammler distribution) is one of a class of Generalized Extreme Value (GEV) distributions used in modeling extreme value problems. This class includes the Gumbel and Frechet distributions.

The probability density for the Weibull distribution is

$$p(x) = \frac{a}{\lambda}(\frac{x}{\lambda})^{a-1}e^{-(x/\lambda)^a},$$

where a is the shape and λ the scale.

The function has its peak (the mode) at $\lambda(\frac{a-1}{a})^{1/a}$.

When `a = 1`, the Weibull distribution reduces to the exponential distribution.

References

Examples

Draw samples from the distribution:

```
>>> a = 5. # shape
>>> s = np.random.weibull(a, 1000)
```

Display the histogram of the samples, along with the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> x = np.arange(1,100.)/50.
>>> def weib(x,n,a):
...     return (a / n) * (x / n)**(a - 1) * np.exp(-(x / n)**a)
```

```
>>> count, bins, ignored = plt.hist(np.random.weibull(5.,1000))
>>> x = np.arange(1,100.)/50.
>>> scale = count.max()/weib(x, 1., 5.).max()
>>> plt.plot(x, weib(x, 1., 5.)*scale)
>>> plt.show()
```

`metilda.controllers.pitch_art_wizard.`**zipf**(*a*, *size=None*)

Draw samples from a Zipf distribution.

Samples are drawn from a Zipf distribution with specified parameter $a > 1$.

The Zipf distribution (also known as the zeta distribution) is a discrete probability distribution that satisfies Zipf's law: the frequency of an item is inversely proportional to its rank in a frequency table.

Note: New code should use the *~numpy.random.Generator.zipf* method of a *~numpy.random.Generator* instance instead; please see the random-quick-start.

Parameters

- **a** (*float or array_like of floats*) – Distribution parameter. Must be greater than 1.
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. If size is `None` (default), a single value is returned if `a` is a scalar. Otherwise, `np.array(a).size` samples are drawn.

Returns

out – Drawn samples from the parameterized Zipf distribution.

Return type

ndarray or scalar

See also:

scipy.stats.zipf
probability density function, distribution, or cumulative density function, etc.

random.Generator.zipf
which should be used for new code.

Notes

The probability density for the Zipf distribution is

$$p(k) = \frac{k^{-a}}{\zeta(a)},$$

for integers $k \geq 1$, where ζ is the Riemann Zeta function.

It is named for the American linguist George Kingsley Zipf, who noted that the frequency of any word in a sample of a language is inversely proportional to its rank in the frequency table.

References

Examples

Draw samples from the distribution:

```
>>> a = 4.0
>>> n = 20000
>>> s = np.random.zipf(a, n)
```

Display the histogram of the samples, along with the expected histogram based on the probability density function:

```
>>> import matplotlib.pyplot as plt
>>> from scipy.special import zeta
```

bincount provides a fast histogram for small integers.

```
>>> count = np.bincount(s)
>>> k = np.arange(1, s.max() + 1)
```

```
>>> plt.bar(k, count[1:], alpha=0.5, label='sample count')
>>> plt.plot(k, n*(k**-a)/zeta(a), 'k.-', alpha=0.5,
...          label='expected count')
>>> plt.semilogy()
>>> plt.grid(alpha=0.4)
>>> plt.legend()
>>> plt.title(f'Zipf sample, a={a}, size={n}')
>>> plt.show()
```

1.1.1.5 metilda.controllers.tempCodeRunnerFile module

1.1.1.6 Module contents

1.1.2 metilda.services package

1.1.2.1 Submodules

1.1.2.2 metilda.services.audio_analysis module

Contains utilities for audio analysis. Some of the visualization functions are based on examples from the parselmouth-praat library: https://github.com/YannickJadoul/Parselmouth

metilda.services.audio_analysis.**audio_analysis_image**(*upload_path*, *tmin=-1*, *tmax=-1*, *min_pitch=75*, *max_pitch=500*, *output_path=None*)

metilda.services.audio_analysis.**draw_pitch**(*ax*, *pitch*, *min_pitch*, *max_pitch*)

metilda.services.audio_analysis.**draw_spectrogram**(*ax*, *spectrogram*, *dynamic_range=70*)

metilda.services.audio_analysis.**get_all_pitches**(*time_range*, *upload_path*, *min_pitch=75*, *max_pitch=500*)

metilda.services.audio_analysis.**get_audio**(*upload_path*, *tmin=-1*, *tmax=-1*)

metilda.services.audio_analysis.**get_avg_pitch**(*time_range*, *upload_path*, *min_pitch=75*, *max_pitch=500*)

metilda.services.audio_analysis.**get_pitches_in_range**(*tmin*, *tmax*, *snd_pitch*)

metilda.services.audio_analysis.**get_sound_length**(*upload_path*)

metilda.services.audio_analysis.**test**(*upload_path*)

1.1.2.3 metilda.services.file_io module

metilda.services.file_io.**available_files**(*dir*)

1.1.2.4 metilda.services.praat module

metilda.services.praat.**runScript**(*scriptName*, *args*)

1.1.2.5 metilda.services.utils module

metilda.services.utils.**deleteCachedImages**(*directory*, *prefix*)

Delete cached images starting with prefix

metilda.services.utils.**fileType**(*fileName*)

Return file extension

metilda.services.utils.**isSound**(*fileName*)

Checks if fileName has a valid sound file extension

metilda.services.utils.**resizeImage**(*image*)

Down-scaling the image to 500x500 pixels

1.1.2.6 Module contents

1.2 Submodules

1.3 metilda.debug module

1.4 metilda.default module

1.5 metilda.local_server module

1.6 metilda.main module

1.7 Module contents

class metilda.**CustomJSONEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

Bases: JSONEncoder

default(*obj*)

Convert o to a JSON serializable type. See json.JSONEncoder.default(). Python does not support overriding how basic types like str or list are serialized, they are handled before this method.

metilda.**get_app**()

metilda.**react_app**(*path=None*)

PYTHON MODULE INDEX

m