simplestatistics Documentation

Release 0.4.0

Sherif Soliman

Nov 17, 2017

Contents

CHAPTER 1

Goal and rules

The goal of simplestatistics is to provide statistical methods that are implemented in readable Python.

simplestatistics is compatible with Python 2 & 3.

- Everything should be implemented in raw, organic, locally sourced python.
- Use libraries only if you have to and only when unrelated to the math/statistics. For example, `from functools import reduce` to make `reduce` available for those using python3. That's okay, because it's about making python work and not about making the stats easier.
- It's okay to use operators and functions if they correspond to regular calculator buttons. For example, all calculators have a built-in square root function, so there is no need to implement that ourselves, we can use `math.sqrt()`. Anything beyond that, like `mean`, `median`, we have to write ourselves.

Pull requests are welcome! Please visit the project on GitHub.

CHAPTER 2

Installation

```
pip install simplestatistics
```

CHAPTER 3

Usage

```
>>> import simplestatistics as ss
>>> ss.mean([1, 2, 3])
2.0
>>> ss.t_test([1, 2, 2.4, 3, 0.9], 2)
-0.3461277235039042
```

CHAPTER 4

Tests

To get coverage reports with tests:

```
pip install coverage
nosetests --with-coverage --cover-package=simplestatistics --with-doctest
```

Otherwise, to just run the tests:

```
nosetests --with-doctest
```

CHAPTER 5

Functions

5.1 Descriptive statistics

5.1.1 Min

simplestatistics.**min**(*data*)

This function returns the smallest numerical value in a data set.

Parameters **data** – A numeric built-in object or list of numeric objects.

Returns A numeric object.

Examples

```
>>> min([1, 2, 3])
1
>>> min([-3, 0, 3])
-3
>>> min([0, 1, 3, -5, 6])
-5
>>> min([-2])
-2
>>> min(-3)
-3
```

5.1.2 Max

simplestatistics.**max**(*data*)

This function returns the maximum numerical value in a data set.

Parameters **data** – A numeric built-in object or list of numeric objects.

Returns A numeric object.

Examples

```
>>> max([1, 2, 3])
3
>>> max([-3, 0, 3])
3
>>> max([-2])
-2
>>> max(-3)
-3
```

5.1.3 Sum

simplestatistics.**sum**(*data*)

This function returns the sum of numerical values in a data set.

To reduce floating-point errors, I am using an implementation of the Kahan summation algorithm.

The implementation is modeled after that of simple-statistics.

Parameters **data** – A numeric built-in object or list of numeric objects.

Returns A numeric object.

Examples

```
>>> sum([1, 2, 3])
6.0
>>> sum([-1, 0, 1])
0.0
>>> sum([2.3, 0, -1.1])
1.2
>>> sum(4)
4
>>> sum((3, 2.5))
5.5
>>> sum('abc')
Traceback (most recent call last):
    ...
TypeError: sum() expects an int, list, or tuple.
```

5.1.4 Quantiles

simplestatistics.**quantile**(*data, p=[0, 0.25, 0.5, 0.75, 1]*)

Quantiles are "are cutpoints dividing the range of a probability distribution into contiguous intervals with equal probabilities, or dividing the observations in a sample in the same way". This function assumes the data provided is a statistical population, not sample.

Consider the first example below:

```
>>> quantile([3, 6, 7, 8, 8, 9, 10, 13, 15, 16, 20], .25)
7
```

Given the probability 0.25, 7 is the value below which you can find 25% of the values after the data is sorted.

Parameters

- **data** – The sample. A list of numerical objects.
- **p** – Can be a numerical object (int or float) indicating one quantile, or a list of numerical objects indicating several quantiles. p is *[0, 0.25, 0.5, 0.75, 1]* by default.

Returns A numerical object if provided p was a single value, or a list of quantiles if provided p was a list. The list will be returned in the order of quantiles provided.

Examples

```
>>> quantile([3, 6, 7, 8, 8, 9, 10, 13, 15, 16, 20], .25)
7
>>> quantile([3, 6, 7, 8, 8, 9, 10, 13, 15, 16, 20], .5)
9
>>> quantile([3, 6, 7, 8, 8, 9, 10, 13, 15, 16, 20], .75)
15
>>> quantile([3, 6, 7, 8, 8, 9, 10, 13, 15, 16, 20], 1)
20
>>> quantile([3, 6, 7, 8, 8, 9, 10, 13, 15, 16, 20])
[3, 7, 9, 15, 20]
>>> quantile([3, 6, 7, 8, 8, 9, 10, 13, 15, 16, 20], [.75, .25])
[15, 7]
```

```
>>> quantile(4, .5)
Traceback (most recent call last):
    ...
TypeError: quantile expects a list of numerical objects.
```

5.1.5 Product

simplestatistics.**product**(*x*, *y*)

This function calculates the product of two vectors or lists of numerical objects, like so:

$$[x_1, x_2, x_3, x_4] \times [y_1, y_2, y_3, y_4] = [x_1 y_1, x_2 y_2, x_3 y_3, x_4 y_4]$$

Parameters

- **x** – An int or a float, or a list or tuple of numerical objects.
- **y** – An int or a float, or a list or tuple of numerical objects. If a list or tuple, must have the same length as x.

Returns A list of numerical objects of the same length as x and y.

Examples

```
>>> product([1, 2, 3], [1, 2, 3])
[1, 4, 9]
>>> product((2, 3), (3, -1))
[6, -3]
>>> product([1.25, 2.75], [2.5, 3.40])
[3.125, 9.35]
>>> product(2, [1, 2, 3])
[2, 4, 6]
>>> product([-3, -2, -1], 3)
[-9, -6, -3]
>>> product(5.5, 6.4)
35.2
```

```
>>> product('a', [2, 4])
Traceback (most recent call last):
    ...
TypeError: product() expects ints, floats, lists, or tuples of numbers.
>>> product([1, 2], [3, 4, 5])
Traceback (most recent call last):
    ...
ValueError: The two lists or tuples have to have equal lengths
```

5.2 Measures of central tendency

5.2.1 Mean

simplestatistics.**mean**(*data*)

The mean, or *average*, is "the sum of values divided by the number of values".

Equation:

$$\bar{X} = \frac{\sum X}{n}$$

Parameters **data** – A numeric built-in object, a tuple, or list of numeric objects.

Returns A float object.

Examples

```
>>> mean([1])
1.0
>>> mean([1, 2])
1.5
>>> mean([1, 2, 3])
2.0
>>> mean([-1, 0, 1, 2, 3])
1.0
```

5.2.2 Median

simplestatistics.**median**(*data*)

The median is "the number separating the higher half of a data sample... from the lower half."

If the sample has an odd number of values, the median is the value in the middle. If the sample has an even number of values, the median is the mean of the two middle values.

Parameters **data** – A numeric built-in object or list of numeric objects.

Returns A float object.

Examples

```
>>> median([1, 2, 3])
2.0
>>> median([1, 2, 3, 4])
2.5
>>> median([10, 2, -5, -1])
0.5
>>> median([-2])
-2.0
>>> median(-3)
-3.0
>>> median("90")
Traceback (most recent call last):
    ...
TypeError: median() expects an int or a list.
```

5.2.3 Mode

simplestatistics.**mode**(*data*)

The mode is "the value that appears most often in a set of data."

If more than one value appear equally frequently more than the rest of the values, then the mode is not unique and includes all the values that appear the same maximum number of times.

In the most extreme of cases, of each value in the dataset appears the same number of times, then the mode is all the values.

Parameters **data** – A numeric built-in object, a tuple, or list of numeric objects.

Returns A list containing one or more modes if the function received a list as input. Or, a numeric object if the function received a numeric object.

Examples

```
>>> mode([1, 2, 3, 1])
[1]
>>> mode([2, 3, 1, 1, 2])
[1, 2]
>>> mode([-1, 1, 0])
[-1, 0, 1]
>>> mode(4)
```

```
4
>>> mode([])
```

5.2.4 Geometric mean

simplestatistics.**geometric_mean**(*data*)

The geometric mean uses the product of a set of numbers to determine their central tendency, as opposed to the regular arithmetic mean which uses their sum.

Equation:

$$\sqrt[n]{x_1 x_2 \dots x_n}$$

Parameters **data** – A list of numeric objects.

Returns A float object.

Examples

```
>>> geometric_mean([1])
1.0
>>> geometric_mean([1, 10])
3.1622776601683795
```

5.2.5 Harmonic mean

simplestatistics.**harmonic_mean**(*x*)

The harmonic mean is a kind of average that is calculated as the reciprocal of the arithmetic mean of the reciprocals. It is appropriate when calculating averages of rates.

Equation:

$$H = \frac{n}{\frac{1}{x_1} + \frac{1}{x_2} + \dots + \frac{1}{x_n}} = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

Parameters **x** – A list or tuple of numerical objects.

Returns A numerical object.

Raises TypeError – If the user passes something other than list or tuple.

Examples

```
>>> harmonic_mean([1, 2, 4])
1.7142857142857142
>>> harmonic_mean(7)
Traceback (most recent call last):
    ...
TypeError: harmonic_mean() expects a list or a tuple.
```

5.2.6 Root mean square

simplestatistics.**root_mean_square**(*x*)

Root mean square (RMS) is the square root of the sum of the squares of values in a list divided by the length of the list. It is a mean function that measures the magnitude of values in the list regardless of their sign.

Parameters **x** – A list or tuple of numerical objects.

Returns A float of the root mean square of the list.

Examples

```
>>> root_mean_square([-1, 1, -1, 1])
1.0
>>> root_mean_square((9, 4))
6.96419413859206
>>> root_mean_square(9)
Traceback (most recent call last):
    ...
TypeError: root_mean_square() expects a list or a tuple.
```

5.2.7 Add to mean

simplestatistics.**add_to_mean**(*current_mean*, *n*, *new_value*, *decimals=2*)

This function can be used when wanting to know the new mean of a list after adding one or more values to it.

One could use this function instead of recalculating the new mean by adding all the old values with the new one and dividing by n (where n is the new total number of items in the list).

Parameters

- **current_mean** – Mean of the list pre-new value.
- **n** – Number of items in the list pre-new value.
- **new_value** – An int, float, or list of ints and/or flots of the new value(s)
- **decimals** – (optional) number of decimal points (default is 2)

Returns A float object denoting the new mean.

Examples

```
>>> add_to_mean(20, 4, 10)
18.0
>>> add_to_mean(40, 4, (10, 12))
30.33
>>> add_to_mean(0, 3, (10, -10, 0))
0.0
>>> add_to_mean(50, 0, 5)
Traceback (most recent call last):
    ...
ValueError: Current n must be an integer greater than 0.
>>> add_to_mean(16, 8, ('5'))
Traceback (most recent call last):
```

```
    ...
TypeError: add_to_mean() requires the new value(s) ... ints and/or floats.
```

5.2.8 Skewness

simplestatistics.**skew**(*x*)

Skew or Skewness is a measure of the asymmetry of the probability distribution of a variable around its mean.

A positive (+ve) skewness value indicates a long tail to the **right** of the mean.

A negative (-ve) skewness value indicates a long tail to the **left** of the mean.

There are several equations to calculate skewness. The one used in this function is Pearson's moment coefficient of skewness.

Equation:

$$\gamma_1 = E\left[\left(\frac{(X-\mu)^3}{\sigma}\right)\right] = \frac{\mu^3}{\sigma^3} = \frac{E[(X-\mu)^3]}{(E[(X-\mu)^3])^{3/2}} = \frac{\kappa_3}{\kappa_2^{3/2}}$$

which can be rewritten as

$$\gamma_1 = \frac{\frac{1}{n}\sum(X-\mu)^3}{\frac{1}{n}(\sum(X-\mu)^2)^{3/2}}$$

Parameters **x** – A list of numerical objects.

Returns A numerical object.

Examples

```
>>> skew([1, 2, 3])
0.0
>>> skew([1, 2, 5])
0.5279896038431618
>>> skew([20, 30]) # skew of a variable that contains two observations is always 0
0.0
>>> skew(9) # no skew for a single number/value
```

5.2.9 Kurtosis

simplestatistics.**kurtosis**(*x*)

Kurtosis is a descriptor of the shape of a probability distribution. It is a measure of the "tailedness" of the probability distribution of a variable.

There are several ways of calculating kurtosis. See this page for a reference.

This function is an implementation of the formula found in Sheskin (2000), which is the one used by SPSS and SAS by default.

Sheskin, D.J. (2000) *Handbook of Parametric and Nonparametric Statistical Procedures, Second Edition*. Boca Raton, Florida: Chapman & Hall/CRC.

Equation:

$$\frac{n(n+1)}{(n-1)(n-2)(n-3)}\left(\frac{s4}{V(x)^2}\right) - 3\frac{(n-1)^2}{(n-2)(n-3)}$$

Where

$s2 = \sum(X - \bar{X})^2$

$s4 = \sum(X - \bar{X})^4$

$V(x) = \frac{s2}{n-1}$

Parameters **x** – A list of numerical objects. This calculation requires **at least 4 observations**.

Returns A numerical object.

Examples

```
>>> kurtosis([1, 2, 3, 4, 5])
-1.1999999999999993
>>> kurtosis([1987, 1987, 1991, 1992, 1992, 1992, 1992, 1993, 1994, 1994, 1995])
0.4466257157411544
>>> kurtosis(2) # no kurtosis for a single number/value
>>> kurtosis([1, 2, 3]) # this kurtosis calculation requires at least 4␣
↪observations
```

5.3 Measures of dispersion

5.3.1 Coefficient of variation

simplestatistics.**coefficient_of_variation**(*data*, *sample=True*)

The coefficient of variation is the ratio of the standard deviation to the mean.

Parameters

- **data** – A list of numerical objects.
- **sample** – A boolean value. If True, calculates coefficient of variation for sample. If False, calculates coefficient of variation for population.

Returns A float object.

Examples

```
>>> coefficient_of_variation([1, 2, 3])
0.5
>>> coefficient_of_variation([1, 2, 3], False)
0.408248290463863
>>> coefficient_of_variation([1, 2, 3, 4])
0.5163977794943222
>>> coefficient_of_variation([-1, 0, 1, 2, 3, 4])
1.247219128924647
```

5.3.2 Variance

simplestatistics.**variance**(*data*, *sample=True*)

Variance is the sum of squared deviations from the mean. It is a general measurement of how far from the mean the values are.

Parameters

- **data** – A list of numeric objects.
- **sample** – A boolean value. If True, calculates sample variance. If False, calculates population variance.

Returns A float object.

Examples

```
>>> variance([1, 2, 3, 4])
1.6666666666666667
>>> variance([1, 2, 3, 4], sample = False)
1.25
>>> variance([1, 2, 3, 4, 5, 6])
3.5
>>> variance([-2, -1, 0, 1, 2])
2.5
>>> variance([1]) # variance of one value is not defined
>>> variance([4]) # variance of one value is not defined
```

5.3.3 Standard deviation

simplestatistics.**standard_deviation**(*data*, *sample=True*)

The standard deviation is the square root of variance (the sum of squared deviations from the mean). The standard deviation is a commonly used measure of the variation and distance of a set of values in a sample from the mean of the sample.

Equation:

$$\sigma = \sqrt{\frac{\sum(x-\mu)^2}{N-1}}$$

In English:

- Obtain the difference between each value and the mean.
- Square those values.
- Sum the squared values.
- Divide by the number of values - 1 (to correct for the sampling).
- Obtain the square root of the result.

Parameters

- **data** – A list of numerical objects.
- **sample** – A boolean value. If True, calculates standard deviation for sample. If False, calculates standard deviation for population.

Returns A float object.

Examples

```
>>> standard_deviation([1, 2, 3])
1.0
>>> standard_deviation([1, 2, 3], False)
0.816496580927726
>>> standard_deviation([1, 2, 3, 4])
1.2909944487358056
>>> standard_deviation([-1, 0, 1, 2, 3, 4])
1.8708286933869707
```

5.3.4 Interquartile range

simplestatistics.**interquartile_range**(*x*)

The interquartile range is the difference between the third and first quartiles of a numerical list. It is a measure of dispersion, or the spread of a distribution.

Parameters **x** – A list or tuple of numerical objects.

Returns An int or float of the difference between third and first quartiles.

Examples

```
>>> interquartile_range([1, 2, 3, 4])
2
>>> interquartile_range([1, 3, 5, 7])
4
>>> interquartile_range(4)
Traceback (most recent call last):
    ...
TypeError: interquartile_range() expects a list or tuple.
```

5.3.5 Sum of Nth power deviations

simplestatistics.**sum_nth_power_deviations**(*x*, *n*)

The sum of the deviations of each value (from the mean) to the Nth power.

Parameters

- **x** – List or tuple of numerical objects.
- **n** – Power to raise each deviation from the mean.

Returns Float of the sum of the deviations from the mean raised to Nth power.

Examples

```
>>> sum_nth_power_deviations([1, 2, 3], 2)
2.0
>>> sum_nth_power_deviations([-1, 0, 2, 4], 3)
7.875
>>> sum_nth_power_deviations(4, 3)
Traceback (most recent call last):
    ...
TypeError: sum_nth_power_deviations() expects the data to be a list or tuple.
```

5.3.6 Standard scores (z scores)

simplestatistics.**z_scores**(*data*, *sample=True*)

Standardizing a variable or set of data is transforming the data such that it has a mean of 0 and standard deviation of 1.

Each converted value equals how many standard deviations the value is above or below the mean. These converted values are known as "z scores".

Equation:

$$z_i = \frac{X_i - \bar{X}}{s_X}$$

In English:

- Subract the value from the mean.
- Divide the result by the standard deviation.

Parameters

- **data** – A list of numerical objects.
- **sample** – A boolean value. If True, calculates z scores for sample. If False, calculates z scores for population.

Returns A list of float objects.

Examples

```
>>> z_scores([-2, -1, 0, 1, 2])
[1.2649110640673518, 0.6324555320336759, 0.0, -0.6324555320336759, -1.
↪2649110640673518]
>>> z_scores([-2, -1, 0, 1, 2], False)
[1.414213562373095, 0.7071067811865475, 0.0, -0.7071067811865475, -1.
↪414213562373095]
>>> z_scores([1, 2])
[0.7071067811865475, -0.7071067811865475]
>>> z_scores([1, 2], False)
[1.0, -1.0]
>>> z_scores([90]) # a z score for one value is not defined
>>> z_scores(4) # a z score for one value is not defined
```

5.4 Linear regression

5.4.1 Linear regression

simplestatistics.**linear_regression**(*x*, *y*, *decimals=2*)

This is a simple linear regression that finds the line of best fit based on a set of points. It uses the least sum of squares to find the slope (m) and y-intercept (b). Maximum number of decimals can be set with optional argument decimals.

Equation:

$$m = \frac{\bar{X}\bar{Y} - \bar{XY}}{(\bar{X})^2 - \bar{X^2}}$$
$$b = \bar{Y} - m\bar{X}$$

Where:

- m is the slope.
- b is the y intercept.

Returns (m, b), where m is the slope and b is the y intercept.

Return type A tuple of two values

Examples

```
>>> linear_regression([1, 2, 3, 4, 5], [4, 4.5, 5.5, 5.3, 6])
(0.48, 3.62)
>>> linear_regression([1, 2, 3, 4, 5], [2, 2.9, 3.95, 5.1, 5.9])
(1.0, 0.97)
>>> linear_regression([0, 1, 2, 3, 4], [1.429, 4.554, 7.679, 1.804, 13.929],␣
↪decimals=3)
(2.225, 1.429)
>>> linear_regression((1, 2), (3, 3.5))
(0.5, 2.5)
>>> linear_regression([1], [2])
(None, 2)
>>> linear_regression(4, 5)
>>> linear_regression([1, 2], [5])
Traceback (most recent call last):
    ...
ValueError: The two variables have to have the same length.
```

5.4.2 Linear regression line function

simplestatistics.**linear_regression_line**(*mb*)

Given the output of `linear_regression()` function, or provided with a tuple of `(m, b)`, where `m` is the slope and `b` is the intercept, `inear_regression_line()` returns a function that calculates y values based on given x values.

Parameters **mb** – A list or tuple of [m, b] or (m, b) where m is the slope and b is the y intercept.

Returns A function that accepts ints, floats, lists, or tuples of x values and returns y values.

Examples

```
>>> linear_regression_line(linear_regression([0, 1], [0, 1]))(1)
1.0
>>> linear_regression_line(linear_regression([1,3,5,7,9], [10,11,12,13,14]))([1,␣
↪2, 3])
[10.0, 10.5, 11.0]
>>> linear_regression_line([.5, 9.5])([1, 2, 3])
[10.0, 10.5, 11.0]
```

```
>>> linear_regression_line(9.5)
Traceback (most recent call last):
    ...
TypeError: linear_regression_line() expects a list or tuple of (slope, intercept).
↪..
>>> linear_regression_line([2, 3, 4])
Traceback (most recent call last):
    ...
ValueError: The list or tuple containing the slope and intercept needs to be of␣
↪length = 2.
```

5.5 Similarity

5.5.1 Correlation

simplestatistics.**correlate**(*x*, *y*)

Correlation refers to "the extent to which two variables have a linear relationship with each other".

A correlation (usually denoted as r) can range from 1.0 to -1.0. r of 1.0 is the strongest positive correlation between two variables, and an r of -1.0 is the strongest negative correlation.

This Cross Validated answer provides a good explanation of the difference between covariance and correlation. Covariance is understood in the context of the units and scales involved. You cannot compare covariances across those contexts. A correlation is a "normalized" covariance that will always be a value between -1 and 1 and takes into account the scale of the variables.

Equation:

$$r_{x,y} = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{ns_x s_y}$$

In English:

- Get the z (standardized) scores of x.
- Get the z (standardized) scores of y.
- Get the product of the two lists of standardized scores.
- Sum the product of standardized scores.
- Divide by the length of x or y $-$ 1 (to correct for sampling).

Parameters

- **x** – A list of numerical objects.

- **y** – A list of numerical objects that has the same length as x.

Returns A numerical object.

Examples

```
>>> correlate([1, 2, 3, 4], [1, 3, 3, 5])
0.9486666666666667
>>> correlate([2, 1, 0, -1, -2, -3, -4, -5], [0, 1, 1, 2, 3, 2, 4, 5])
-0.9434285714285714
```

```
>>> correlate(2, 3)
Traceback (most recent call last):
    ...
ValueError: To calculate correlation you need lists or tuples of equal length...
>>> correlate([2, 4], [6, 6.5, 7])
Traceback (most recent call last):
    ...
ValueError: To calculate correlation you need lists or tuples of equal length...
>>> correlate([1], [-1])
Traceback (most recent call last):
    ...
ValueError: Correlation requires lists of equal length where length is > 1.
```

5.5.2 Covariance

simplestatistics.**covariance**(*x*, *y*)

(Sample) Covariance is a measure of how two random variables vary together. When the greater values of one variable correspond to the greater values of the other variable, this is a positive covariance. Whereas when the greater values of one variable correspond to the lesser values of the other variable, this is negative covariance.

This Cross Validated answer provides a good explanation of the difference between covariance and correlation. Covariance is understood in the context of the units and scales involved. You cannot compare covariances across those contexts. A correlation is a "normalized" covariance that will always be a value between -1 and 1 and takes into account the scale of the variables.

Equation:

$$q_{jk} = \frac{1}{N-1}\sum_{i=1}^{N}(X_{ij} - \bar{X}_j)(X_{ik} - \bar{X}_k)$$

Parameters

- **x** – A list of numerical objects.
- **y** – A list of numerical objects that has the same length as x.

Returns A numerical object.

Examples

```
>>> covariance([1,2,3,4,5,6], [6,5,4,3,2,1])
-3.5
```

```
>>> covariance([1,2,3], [4, 4.5, 5])
0.5
```

```
>>> covariance(2, 3)
Traceback (most recent call last):
    ...
ValueError: To calculate covariance you need lists or tuples of equal length...
>>> covariance([2, 4], [6, 6.5, 7])
Traceback (most recent call last):
    ...
ValueError: To calculate covariance you need lists or tuples of equal length...
>>> covariance([1], [-1])
Traceback (most recent call last):
    ...
ValueError: covariance requires lists of equal length where length is > 1.
```

5.6 Distributions

5.6.1 Factorial

simplestatistics.**factorial**(*x*)

The factorial (denoted $n!$) is the product of all positive integers less than or equal to a positive integer (n).

Equation:

$$n! = \prod_{k=1}^{n} k$$

Parameters **x** – A numeric built-in object, or a list of numeric built-in objects for n

Returns A numerical (int) object.

Examples

```
>>> factorial(5)
120
>>> factorial(1)
1
>>> factorial(20)
2432902008176640000
>>> factorial([1, 5, 20])
[1, 120, 2432902008176640000]
```

```
>>> factorial(-2)
Traceback (most recent call last):
    ...
ValueError: factorial() expects a positive integer.
>>> factorial(4.5)
Traceback (most recent call last):
    ...
TypeError: factorial() expects a positive integer or list of positive integers.
>>> factorial([2, 3, 'a'])
```

```
Traceback (most recent call last):
    ...
ValueError: Perhaps you provided a list that contains non-integers.
```

5.6.2 Choose

simplestatistics.**choose**(*n*, *k*)

The choose function calculates the binomial coefficient as used in combinatorics and other counting problems.

The binomical coefficient, written as $\binom{n}{k}$ and often read aloud as 'n choose k' is the answer to the question "how many ways are there to choose k elements, regardless of their order, from a set of n elements?".

Equation:

$$\frac{n!}{k!(n-k)!}$$

Parameters

- **n** – An integer.
- **k** – An integer.

Returns An integer.

Examples

```
>>> choose(5, 3)
10
```

```
>>> choose(2.1, 5)
Traceback (most recent call last):
    ...
TypeError: choose() expects both n and k to be integers
```

5.6.3 Normal distribution

simplestatistics.**normal**(*x*, *mean*, *standard_deviation*)

The Normal Distribution is, quoting from the Wikipedia page:

> A very common continuous probability distribution. Normal distributions are important in statistics and are often used in the natural and social sciences to represent real-valued random variables whose distributions are not known.
>
> The normal distribution is useful because of the central limit theorem. In its most general form, under some conditions (which include finite variance), it states that averages of random variables independently drawn from independent distributions converge in distribution to the normal, that is, become normally distributed when the number of random variables is sufficiently large. Physical quantities that are expected to be the sum of many independent processes (such as measurement errors) often have distributions that are nearly normal. Moreover, many results and methods (such as propagation of uncertainty and least squares parameter fitting) can be derived analytically in explicit form when the relevant variables are normally distributed.

Probability density function equation:

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

Parameters

- **x** – Int or float or list of ints or floats representing values for which
- **normal distribution density is desired.** (*the*) –
- **mean** – Mean of the sample or population.
- **standard_deviation** – Standard deviation of the sample or population.

Returns Float or (or list of floats if provided x was a list) representing the normal distribution density function for x or each value in x.

Examples

```
>>> normal(4, 8, 2)
0.02699548325659403
>>> normal([1, 4], 8, 2)
[0.00043634134752288024, 0.02699548325659403]
>>> normal(4, 8, 0)
0
```

5.6.4 Binomial distribution

simplestatistics.**binomial**(*k*, *n*, *p*)

The Binomial Distribution is, quoting from the Wikipedia page:

> In probability theory and statistics, the binomial distribution with parameters n and p is the discrete probability distribution of the number of successes in a sequence of n independent yes/no experiments, each of which yields success with probability p.

Probability mass function equation:

$$f(k; n, p) = Pr(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

Parameters

- **k** – Int or list of ints representing number of choices or successes.
- **n** – Int representing total number of trials.
- **p** – Float representing probability of success per trial.

Returns Float (or list of floats, if provided k was a list) representing probabilities of obtaining each k according to the binomial distribution.

Examples

```
>>> binomial(4, 12, 0.2)
0.13287555072
>>> binomial([1, 2, 3], 10, 0.5)
[0.009765625, 0.0439453125, 0.1171875]
```

```
>>> binomial(4, 10, 1.5)
Traceback (most recent call last):
    ...
ValueError: probability cannot be greater than 1 or smaller than 0
```

5.6.5 Bernoulli distribution

simplestatistics.**bernoulli**(*p*, *decimals=2*)

The Bernoulli distribution is, quoting from the Wikipedia page:

> The Bernoulli distribution is the probability discrete distribution of a random variable which takes value 1 with success probability p and value 0 with failure probability q = 1 - p. It can be used, for example, to represent the toss of a coin, where "1" is defined to mean "heads" and "0" is defined to mean "tails" (or vice versa). It is a special case of a Binomial Distribution where n = 1.

Parameters

- **p** – Int or float representing *p* (probability)
- **decimals** – (optional) number of decimal points (default is 2)

Returns List of the distribution values for the variable taking values 0, 1.

Examples

```
>>> bernoulli(0.5)
[0.5, 0.5]
>>> bernoulli(0.25)
[0.75, 0.25]
>>> bernoulli(0.3)
[0.7, 0.3]
>>> bernoulli(1.1)
Traceback (most recent call last):
    ...
ValueError: Probability (p) must be between 0 and 1.
```

5.6.6 Poisson distribution

simplestatistics.**poisson**(*lam*, *k=[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]*, *decimals=4*)

The Poisson distribution is, quoting from the Wikipedia page:

> ... a discrete probability distribution that expresses the probability of a given number of events occurring in a fixed interval of time and/or space if these events occur with a known average rate and independently of the time since the last event.

This function calculates the probability of events for a Poisson distribution.

An event can occur 0, 1, 2,... times in an interval. The average number of events in an interval is designated λ (lambda). Lambda is the event rate, also called the rate parameter. The probability of observing k events in an interval is given by the equation:

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

λ is the average number of events per interval

e is the number 2.71828... (Euler's number)

k is the number of events, takes values 0, 1, 2...

If provided with a k value or list of values, the function will return the probabilities for those values. Otherwise, the function will return the probabilities for k = [0, 1, 2... 20].

Parameters

- **lam** – Value for *lambda*. Has to be a positive value
- **k** – Value for *k*. Default is [0, 1, 2... 20]
- **decimals** – (optional) number of decimal points (default is 4)

Returns A value or list of values for the probability(ies) of observing the one or more provided *k* values given provided *lamda*.

Examples

```
>>> poisson(3, 3)
0.224
>>> poisson(3, [0, 1, 2, 3])
[0.0498, 0.1494, 0.224, 0.224]
>>> poisson(5.5, 7)
0.1234
```

```
>>> poisson(-3)
Traceback (most recent call last):
    ...
ValueError: lambda has to be a positive value.
```

5.6.7 Gamma function

simplestatistics.**gamma_function**(*x*, *decimals=6*)

The gamma function is an extension of the factorial function for all positive integers. It is a common component in many probability distributions (e.g., the beta distribution).

The probability density function (PDF) for integers is defined as:

$$\Gamma(n) = (n-1)!$$

Whereas for decimals/fractions we use Stirling's approximation:

$$\Gamma(x) \approx \sqrt{\frac{2\pi}{x}} \left(\frac{1}{e} \left(x + \frac{1}{12z - \frac{1}{10z}} \right) \right)^x$$

Parameters

- **x** – Non-negative int or float or list of ints or floats representing values for which the Gamma function probability density function values are desired.
- **decimals** – (optional) number of decimal points (default is 6)

Returns Int or float (or list of ints or floats) representing the Gamma function probability density function values for x or each value in x.

Examples

```
>>> gamma_function(6)
120
>>> gamma_function(1.2, decimals=2)
0.92
>>> gamma_function([6, 8, 10, 12])
[120, 5040, 362880, 39916800]
>>> gamma_function((3, 4))
[2, 6]
>>> gamma_function([3, .5], decimals=2)
[2, 1.76]
>>> gamma_function(-.5)
Traceback (most recent call last):
    ...
ValueError: gamma_function is only defined for non-negative values.
```

5.6.8 Beta distribution

simplestatistics.**beta**(*x*, *_alpha*, *_beta*, *decimals=6*)

Returns probability density function for beta distribution

$$\text{PDF} = \frac{x^{\alpha-1}(1-x)^{\beta-1}}{\text{B}(\alpha,\beta)}$$

$$\text{where B}(\alpha,\beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha+\beta)}$$

Parameters

- **x** – Float or list of floats between 0 and 1 representing values for which the Beta distribution probability density function is desired.
- **_alpha** – Int or float representing α
- **_beta** – Int of float representing β
- **decimals** – (optional) number of decimal points (default is 6)

Returns Float or list of floats representing Beta distribution probability density function values fo x or each value in x.

Examples

```
>>> beta(.5, 1, 3)
0.75
>>> beta([.01, .02, .03], 2, 5)
[0.288179, 0.553421, 0.796764]
>>> beta(-.5, 3, 3)
Traceback (most recent call last):
    ...
ValueError: The beta distribution is only defined for non-negative values.
```

5.6.9 One-sample t test

simplestatistics.**t_test**(*sample*, *x*)

A one-sample t-test is a test that compares the mean of a sample to a known value, x.

In this case, we're trying to determine whether the sample mean is equal to the value that we know, which is *x*. Usually the results are used to look up a p-value, which, for a certain level of significance, will let you determine whether the null hypothesis (that there is no real difference between the mean of the sample and provided *x*) can be rejected or not.

Equation:

$$t = \frac{\bar{X} - \mu}{sd_X}$$

$\bar{X}$ is the sample mean

μ is the provided value

sd_X is the sample standard deviation

Parameters

- **sample** – A list of numerical objects (the sample)
- **x** – The provided value to compare the mean of the sample to.

Returns A numerical object.

Example

```
>>> t_test([1, 2, 3, 4, 5, 6], 3.385)
0.150570344262835
```

5.6.10 Chi Squared Distribution Table

simplestatistics.**chi_squared_dist_table**(*k=0*, *p=0*)

From Wikipedia.

The p-value is the probability of observing a test statistic at least as extreme in a chi-squared distribution. Accordingly, since the cumulative distribution function (CDF) for the appropriate degrees of freedom (df) gives the probability of having obtained a value less extreme than this point, subtracting the CDF value from 1 gives the p-value. The table below gives a number of p-values matching to χ^2 for the first 10 degrees of freedom.

The `chi_squared_dist_table` function can either return the full table of χ^2 vs. p values, or it can perform a lookup for a certain value if given k (degrees of freedom: df) or k and p values.

k has to be one of the following values:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 40, 50, 60, 70, 80, 90, 100

p has to be one of the following values:

.005, .01, .025, .05, .1, .5, .9, .95, .975, .99, .995

Parameters

- **k** – (optional) int of k representing degrees of freedom.
- **p** – (optiona, but k is required) float of p representing probability.

Returns Entire table as dict where the key is degree of freedom and value is another dict of keys representing p and values representing test statistic. If only k is provided, the function returns a dict of p: test statistic pairs. If both k and p are provided, the function will return the specific test statistic.

Examples

```
>>> chi_squared_dist_table(5, .01)
15.09
>>> x = chi_squared_dist_table()
>>> sorted(x.keys())
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, ..., 100]
```

```
>>> k60 = chi_squared_dist_table(60)
>>> for p in sorted(k60.keys()): print({p: k60[p]})
{0.005: 91.95}
{0.01: 88.38}
{0.025: 83.3}
{0.05: 79.08}
{0.1: 74.4}
{0.5: 59.33}
{0.9: 46.46}
{0.95: 43.19}
{0.975: 40.48}
{0.99: 37.48}
{0.995: 35.53}
```

```
>>> chi_squared_dist_table(5, 0.4)
Traceback (most recent call last):
    ...
ValueError: If you provide a p value, it has to be one of 11 values...
>>> chi_squared_dist_table(95)
Traceback (most recent call last):
    ...
ValueError: If you are providing k and/or p values, they have to be chosen...
```

5.7 Utilities

5.7.1 Decimalize

Implements unexposed helper function decimalize() used by other exposed functions to arrive at better numerical accuracy and avoid floating point errors.

simplestatistics.statistics.decimalize.**decimalize**(*data*)

Utility function converting all inputs to Decimal, streamlines input types for other statistics functions.

Parameters **data** – A numeric built-in object, a tuple or list of numeric objects.

Returns A Decimal object in the case of a single numeric built-in, or a list of Decimal objects when supplied a list or tuple of built-in numerics.

Raises

- TypeError – An object other than a built-in numeric or a list or tuple
- of numerics was supplied as data.

Examples

```
>>> decimalize(1)
Decimal('1')
>>> decimalize([1,2,3])
[Decimal('1'), Decimal('2'), Decimal('3')]
>>> decimalize((1,2,3))
[Decimal('1'), Decimal('2'), Decimal('3')]
```

```
>>> decimalize('abc')
Sorry, the decimalize function accepts lists or tuples of numerics
```

5.8 Classifiers

5.8.1 Bayesian classifier

class simplestatistics.**bayesian_classifier**

A naive Bayesian classifier.

The implemention of this classifier is very closely modeled after the implementation in simple-statistics, the javascript analogue of simplestatistics. You can find the javascript implementation of the Bayesian classifier here.

Examples

Making a seventy-five/twenty-five classification.

```
>>> model1 = bayesian_classifier()
>>> model1.train({'species': 'cat'}, 'animal')
>>> model1.train({'species': 'cat'}, 'animal')
>>> model1.train({'species': 'cat'}, 'animal')
```

```
>>> model1.train({'species': 'cat'}, 'feline')
>>> model1.count
4
>>> model1.score({'species': 'cat'})
[('animal', 0.75), ('feline', 0.25)]
```

Classifying multiple things

```
>>> model2 = bayesian_classifier()
>>> model2.train({'species': 'cat'}, 'animal')
>>> model2.train({'species': 'dog'}, 'animal')
>>> model2.train({'species': 'dog'}, 'animal')
>>> model2.train({'species': 'cat'}, 'chair')
>>> model2.score({'species': 'cat'})
[('chair', 0.25), ('animal', 0.25)]
>>> model2.score({'species': 'dog'})
[('animal', 0.5), ('chair', 0)]
```

Testing multiple properties

```
>>> model3 = bayesian_classifier()
>>> model3.train({'species': 'cat'}, 'animal')
>>> model3.train({'species': 'cat'}, 'animal')
>>> model3.train({'species': 'cat'}, 'animal')
>>> model3.train({'species': 'cat'}, 'chair')
>>> model3.train({'species': 'cat', 'color': 'white'}, 'chair')
>>> model3.score({'color': 'white'})
[('chair', 0.2), ('animal', 0)]
```

```
>>> mod = bayesian_classifier()
>>> mod.score({'color': 'purple'})
Traceback (most recent call last):
    ...
RuntimeError: The model has not been trained yet. Train the model ... item.
```

score (*item*)

Scores a certain item based on the learning the model has done on previous items.

Parameters **item** – A dict in the form {property: value} of the item you want to score.

Returns A list containing tuples of properties and scores. The list is ordered in descending order of scores.

train (*item*, *category*)

The method to train the instance of the Bayesian classifier on an item.

Parameters

- **item** – A dict of property-value pairs in the form {property_1: value1,
- **property2** – value2, . . .} for the item. category: A string of the
- **of the item.** (*category*) –

Returns null

5.8.2 Perceptron

class simplestatistics.**perceptron**

A perceptron model.

The implementation of the perceptron model is closely modeled after the implementation in simple-statistics, the javascript analogue of simplestatistics. You can find the javascript implementation of the perceptron model here.

Examples

```
>>> mod = perceptron()
>>> for ii in range(10):
...     mod.train([0,1], 0)
...     mod.train([1,0], 0)
...     mod.train([1,1], 1)
...     mod.train([0,0], 0)
>>> mod.predict([1, 0])
0
>>> mod.predict([1, 1])
1
```

You cannot predict an item with a model that hasn't been trained yet.

```
>>> mod2 = perceptron()
>>> mod2.predict([0, 0])
Traceback (most recent call last):
    ...
RuntimeError: The model has not been trained yet.
```

When training, labels need to be 0 or 1.

```
>>> mod3 = perceptron()
>>> mod3.train([1, 1], 4)
Traceback (most recent call last):
    ...
ValueError: Labels need to be either 0 or 1.
```

Once trained on an item, the rest of the training items need to have features of the same length.

```
>>> mod4 = perceptron()
>>> mod4.train([1, 0], 1)
>>> mod4.train([1, 1], 1)
>>> mod4.train([1, 1, 0], 1)
Traceback (most recent call last):
    ...
ValueError: The length of features is different ... to use new feature lengths.
```

predict(*features*)

Classifies an item based on the learning the instance of the model has done on previous items.

Parameters

- **features** – A list of the features of the item to classify. The length
- **the list has to be the same as that of the lists of features the**(*of*) –

- **trained on.** (*model*) –

Returns 0 or 1 denoting the predicted category/classification.

train (*features*, *label*)
The method to train an instance of the perceptron model on an item.

Parameters

- **features** – A list of numerical features in the form [feature_1,
- **..]The length of the list needs to be the same for each** (*feature_2,*) –
- **given to the same model/instance. label** (*item*) – An integer of value 0
- **1 to denote category of the item.** (*or*) –

Returns null

5.9 Errors

5.9.1 Error function

class simplestatistics.**error_function**
The error function, or Gauss error function.

The function returns the probability that a value in a normal distribution is between $\frac{x}{sd\sqrt{2}}$ and $\frac{-x}{sd\sqrt{2}}$.

This implementation is closely modeled after the impementation in simple-statistics, and returns a numerical approximation of the exact value.

Parameters

- **x** – A numerical object denoting sd
- **decimals** – number of decimal points (default is 2)

Returns Probability between 0 and 1.

Examples

```
>>> error_function(1)
0.84
>>> error_function(.8, decimals = 1)
0.7
>>> error_function(1.01)
0.85
>>> error_function(-0.4)
-0.43
>>> error_function('.75')
Traceback (most recent call last):
    ...
ValueError: error_function() only accepts values of type int or float.
```

5.10 Hyperbolic functions

5.10.1 sinh

class simplestatistics.**sinh**

The hyperbolic sin, analogous to the trigonometric sin.

Hyperbolic functions are defined on a parabola whereas trigonometric functions are defined on a circle.

One of the uses of $sinh$ is in calculating Stirling's approximation for factorials, which in turn is useful for calculating the gamma function and the beta distribution probability density function.

I've come across two different equations that produce the same results when calculating $sinh$. One found on the Wikipedia page for hyperbolic functions:

$$\sinh(x) = \frac{1 - e^{-2x}}{2e^{-x}}$$

And a simpler equation found on this reference page by Erik Max Francis:

$$\sinh(x) = \frac{e^{x} - e^{-x}}{2}$$

I implemented the second version for simplicity.

Parameters **x** – int or float

Returns A float object denoting the hyperbolic sin of input

Examples

```
>>> sinh(2)
3.62686
>>> sinh(2.2)
4.45711
>>> sinh('3')
Traceback (most recent call last):
    ...
TypeError: sinh expects an integer or float.
```

5.10.2 cosh

class simplestatistics.**cosh**

The hyperbolic cos, analogous to the trigonometric cos.

Hyperbolic functions are defined on a parabola whereas trigonometric functions are defined on a circle.

There are – at least – two different equations that produce the same results when calculating $cosh$. One found on the Wikipedia page for hyperbolic functions:

$$\cosh(x) = \frac{1 + e^{-2x}}{2e^{-x}}$$

And a simpler equation found on this reference page by Erik Max Francis:

$$\cosh(x) = \frac{e^x + e^{-x}}{2}$$

I implemented the second version for simplicity.

Parameters **x** – int or float

Returns A float object denoting the hyperbolic cos of input.

Examples

```
>>> cosh(2)
3.7622
>>> cosh(2.5)
6.13229
>>> cosh('c')
Traceback (most recent call last):
    ...
TypeError: cosh expects an integer or float.
```

5.10.3 tanh

class simplestatistics.**tanh**

The hyperbolic tan, analogous to the trigonometric tan.

Hyperbolic functions are defined on a parabola whereas trigonometric functions are defined on a circle.

There are – at least – two different equations that produce the same results when calculating $tanh$. One found on the Wikipedia page for hyperbolic functions:

$$\tanh(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

And a simpler equation found on this reference page by Erik Max Francis:

$$\tanh(x) = \frac{sinh(x)}{cosh(x)}$$

Parameters **x** – int or float

Returns A float object denoting the hyperbolic tan of input.

Examples

```
>>> tanh(3)
0.99505
>>> tanh(0.2)
0.19738
```

```
>>> tanh('c')
Traceback (most recent call last):
    ...
TypeError: tanh expects an integer or float.
```

CHAPTER 6

Indices and tables

- genindex
- modindex

Python Module Index

S

Index

V

Z