9.7. `statistics` — Mathematical statistics functions

New in version 3.4.

Source code: Lib/statistics.py

This module provides functions for calculating mathematical statistics of numeric (`Real`-valued) data.

Note: Unless explicitly noted otherwise, these functions support `int`, `float`, `decimal.Decimal` and `fractions.Fraction`. Behaviour with other types (whether in the numeric tower or not) is currently unsupported. Mixed types are also undefined and implementation-dependent. If your input data consists of mixed types, you may be able to use `map()` to ensure a consistent result, e.g. `map(float, input_data)`.

9.7.1. Averages and measures of central location

These functions calculate an average or typical value from a population or sample.

`mean()`	Arithmetic mean ("average") of data.
`median()`	Median (middle value) of data.
`median_low()`	Low median of data.
`median_high()`	High median of data.
`median_grouped()`	Median, or 50th percentile, of grouped data.
`mode()`	Mode (most common value) of discrete data.

9.7.2. Measures of spread

These functions calculate a measure of how much the population or sample tends to deviate from the typical or average values.

`pstdev()`	Population standard deviation of data.
`pvariance()`	Population variance of data.
`stdev()`	Sample standard deviation of data.
`variance()`	Sample variance of data.

9.7.3. Function details

Note: The functions do not require the data given to them to be sorted. However, for reading convenience, most of the examples show sorted sequences.

`statistics.`**`mean`**(*data*)

Return the sample arithmetic mean of *data*, a sequence or iterator of real-valued numbers.

The arithmetic mean is the sum of the data divided by the number of data points. It is commonly called “the average”, although it is only one of many different mathematical averages. It is a measure of the central location of the data.

If *data* is empty, `StatisticsError` will be raised.

Some examples of use:

```
>>> mean([1, 2, 3, 4, 4])
2.8
>>> mean([-1.0, 2.5, 3.25, 5.75])
2.625

>>> from fractions import Fraction as F
>>> mean([F(3, 7), F(1, 21), F(5, 3), F(1, 3)])
Fraction(13, 21)

>>> from decimal import Decimal as D
>>> mean([D("0.5"), D("0.75"), D("0.625"), D("0.375")])
Decimal('0.5625')
```

Note: The mean is strongly affected by outliers and is not a robust estimator for central location: the mean is not necessarily a typical example of the data points. For more robust, although less efficient, measures of central location, see `median()` and `mode()`. (In this case, “efficient” refers to statistical efficiency rather than computational efficiency.)
The sample mean gives an unbiased estimate of the true population mean, which means that, taken on average over all the possible samples, `mean(sample)` converges on the true mean of the entire population. If *data* represents the entire population rather than a sample, then `mean(data)` is equivalent to calculating the true population mean μ.

`statistics.`**`median`**(*data*)

Return the median (middle value) of numeric data, using the common “mean of middle two” method. If *data* is empty, `StatisticsError` is raised.

The median is a robust measure of central location, and is less affected by the presence of outliers in your data. When the number of data points is odd, the middle data point is returned:

```
>>> median([1, 3, 5])
3
```

When the number of data points is even, the median is interpolated by taking the average of the two middle values:

```
>>> median([1, 3, 5, 7])
4.0
```

This is suited for when your data is discrete, and you don't mind that the median may not be an actual data point.

See also: `median_low()`, `median_high()`, `median_grouped()`

`statistics.`**`median_low`**`(`*data*`)`

Return the low median of numeric data. If *data* is empty, `StatisticsError` is raised.

The low median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the smaller of the two middle values is returned.

```
>>> median_low([1, 3, 5])
3
>>> median_low([1, 3, 5, 7])
3
```

Use the low median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

`statistics.`**`median_high`**`(`*data*`)`

Return the high median of data. If *data* is empty, `StatisticsError` is raised.

The high median is always a member of the data set. When the number of data points is odd, the middle value is returned. When it is even, the larger of the two middle values is returned.

```
>>> median_high([1, 3, 5])
3
>>> median_high([1, 3, 5, 7])
5
```

Use the high median when your data are discrete and you prefer the median to be an actual data point rather than interpolated.

`statistics.`**`median_grouped`**`(`*data*, *interval=1*`)`

Return the median of grouped continuous data, calculated as the 50th percentile, using interpolation. If *data* is empty, `StatisticsError` is raised.

```
>>> median_grouped([52, 52, 53, 54])
52.5
```

In the following example, the data are rounded, so that each value represents the midpoint of data classes, e.g. 1 is the midpoint of the class 0.5-1.5, 2 is the midpoint of

1.5-2.5, 3 is the midpoint of 2.5-3.5, etc. With the data given, the middle value falls somewhere in the class 3.5-4.5, and interpolation is used to estimate it:

```
>>> median_grouped([1, 2, 2, 3, 4, 4, 4, 4, 4, 5])
3.7
```

Optional argument *interval* represents the class interval, and defaults to 1. Changing the class interval naturally will change the interpolation:

```
>>> median_grouped([1, 3, 3, 5, 7], interval=1)
3.25
>>> median_grouped([1, 3, 3, 5, 7], interval=2)
3.5
```

This function does not check whether the data points are at least *interval* apart.

CPython implementation detail: Under some circumstances, `median_grouped()` may coerce data points to floats. This behaviour is likely to change in the future.

See also:

- "Statistics for the Behavioral Sciences", Frederick J Gravetter and Larry B Wallnau (8th Edition).
- Calculating the median.
- The SSMEDIAN function in the Gnome Gnumeric spreadsheet, including this discussion.

`statistics.`**`mode`**(*data*)

Return the most common data point from discrete or nominal *data*. The mode (when it exists) is the most typical value, and is a robust measure of central location.

If *data* is empty, or if there is not exactly one most common value, `StatisticsError` is raised.

`mode` assumes discrete data, and returns a single value. This is the standard treatment of the mode as commonly taught in schools:

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

The mode is unique in that it is the only statistic which also applies to nominal (non-numeric) data:

```
>>> mode(["red", "blue", "blue", "red", "green", "red", "red"])
'red'
```

`statistics.`**`pstdev`**(*data*, *mu=None*)

Return the population standard deviation (the square root of the population variance). See `pvariance()` for arguments and other details.

```
>>> pstdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
0.986893273527251
```

`statistics.`**`pvariance`**(*data*, *mu=None*)

Return the population variance of *data*, a non-empty iterable of real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *mu* is given, it should be the mean of *data*. If it is missing or `None` (the default), the mean is automatically calculated.

Use this function to calculate the variance from the entire population. To estimate the variance from a sample, the `variance()` function is usually a better choice.

Raises `StatisticsError` if *data* is empty.

Examples:

```
>>> data = [0.0, 0.25, 0.25, 1.25, 1.5, 1.75, 2.75, 3.25]
>>> pvariance(data)
1.25
```

If you have already calculated the mean of your data, you can pass it as the optional second argument *mu* to avoid recalculation:

```
>>> mu = mean(data)
>>> pvariance(data, mu)
1.25
```

This function does not attempt to verify that you have passed the actual mean as *mu*. Using arbitrary values for *mu* may lead to invalid or impossible results.

Decimals and Fractions are supported:

```
>>> from decimal import Decimal as D
>>> pvariance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('24.815')

>>> from fractions import Fraction as F
>>> pvariance([F(1, 4), F(5, 4), F(1, 2)])
Fraction(13, 72)
```

Note: When called with the entire population, this gives the population variance σ^2. When called on a sample instead, this is the biased sample variance s^2, also known as variance with N degrees of freedom.
If you somehow know the true population mean μ, you may use this function to calculate the variance of a sample, giving the known population mean as the second argument. Provided the data points are representative (e.g. independent and identically distributed), the result will be an unbiased estimate of the population variance.

statistics.**stdev**(*data*, *xbar=None*)

Return the sample standard deviation (the square root of the sample variance). See `variance()` for arguments and other details.

```
>>> stdev([1.5, 2.5, 2.5, 2.75, 3.25, 4.75])
1.0810874155219827
```

statistics.**variance**(*data*, *xbar=None*)

Return the sample variance of *data*, an iterable of at least two real-valued numbers. Variance, or second moment about the mean, is a measure of the variability (spread or dispersion) of data. A large variance indicates that the data is spread out; a small variance indicates it is clustered closely around the mean.

If the optional second argument *xbar* is given, it should be the mean of *data*. If it is missing or `None` (the default), the mean is automatically calculated.

Use this function when your data is a sample from a population. To calculate the variance from the entire population, see `pvariance()`.

Raises `StatisticsError` if *data* has fewer than two values.

Examples:

```
>>> data = [2.75, 1.75, 1.25, 0.25, 0.5, 1.25, 3.5]
>>> variance(data)
1.3720238095238095
```

If you have already calculated the mean of your data, you can pass it as the optional second argument *xbar* to avoid recalculation:

```
>>> m = mean(data)
>>> variance(data, m)
1.3720238095238095
```

This function does not attempt to verify that you have passed the actual mean as *xbar*. Using arbitrary values for *xbar* can lead to invalid or impossible results.

Decimal and Fraction values are supported:

```
>>> from decimal import Decimal as D
>>> variance([D("27.5"), D("30.25"), D("30.25"), D("34.5"), D("41.75")])
Decimal('31.01875')

>>> from fractions import Fraction as F
>>> variance([F(1, 6), F(1, 2), F(5, 3)])
Fraction(67, 108)
```

Note: This is the sample variance s^2 with Bessel's correction, also known as variance with N-1 degrees of freedom. Provided that the data points are representative (e.g.

independent and identically distributed), the result should be an unbiased estimate of the true population variance.

If you somehow know the actual population mean μ you should pass it to the `pvariance()` function as the *mu* parameter to get the variance of a sample.

9.7.4. Exceptions

A single exception is defined:

exception `statistics.`**`StatisticsError`**

Subclass of `ValueError` for statistics-related exceptions.